Experiment-10

Transformers in Vision

1. Aim

To understand the core principles of Vision Transformers (ViT) by exploring self-attention mechanisms and patch-based image representations, and to apply a pretrained or lightly fine-tuned ViT model on a CIFAR-10 subset, with detailed visualization of patch embeddings and self-attention maps for selected images.

2. Theory

Vision Transformers represent an important shift in computer vision by replacing convolution-based feature extraction with attention-based learning. Unlike Convolutional Neural Networks (CNNs), which mainly capture local patterns through convolutional filters and receptive fields, Vision Transformers model relationships among all regions of an image using the self-attention mechanism. This allows the model to capture both local visual details and global contextual information across the entire image.

A Vision Transformer processes an image through a sequence of stages. First, the image is divided into small fixed-size patches. These patches are then flattened and converted into patch embeddings. Positional embeddings are added to preserve spatial information, and the resulting token sequence is then passed through a stack of Transformer encoder blocks. Finally, the MLP head uses a classification token or a pooled representation to predict the output class.

The major stages in a ViT model are:

1. Image patching
2. Patch embedding
3. Positional encoding
4. Transformer encoder block
5. Classification token and MLP classification head

2.1 Image Patching

Image patching is the first step in the Vision Transformer pipeline. In this process, an input image is divided into a set of fixed-size, non-overlapping square patches. Instead of processing the image as a whole, ViT treats each patch as an individual visual token, similar to how words are treated as tokens in natural language processing.

Each image patch is flattened into a one-dimensional vector and then passed through a linear projection layer to convert it into an embedding. These embedded patches form the input sequence for the Transformer encoder.

Fig. 1 illustrates the patching process applied to an automobile image from the CIFAR-10 dataset. The original image is divided into smaller image regions, where each patch represents a localized visual component. These patches are later embedded and processed by the Transformer encoder to learn meaningful visual representations.

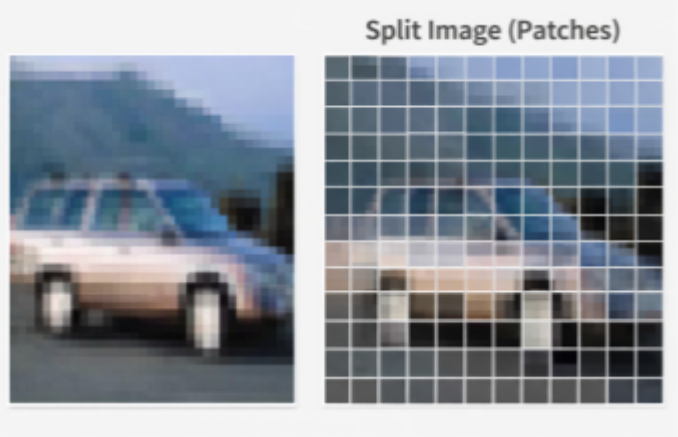


Fig. 1. Original automobile image and its corresponding patch-wise representation

2.2 Patch Embedding

After the image is divided into patches, each patch is flattened into a vector. However, the raw flattened patch vectors cannot be directly processed by the Transformer encoder. Therefore, each flattened patch is linearly projected into a fixed-dimensional embedding space.

If an image has dimensions $H \times W \times C$, and each patch has size $P \times P$, then the number of patches is:

$$N = \frac{H \times W}{P^2}$$

Each patch is flattened into a vector of size:

$$P^2 \cdot C$$

This vector is then projected into a D-dimensional embedding space, where $D$is the embedding dimension of the model. The output is a sequence of patch embeddings that serves as the input to the Transformer.

Since the Transformer architecture was originally designed for sequential data, it does not naturally understand the spatial arrangement of image patches. Therefore, positional embeddings are added to the patch embeddings to preserve information about each patch's location in the original image.

2.3 Positional Encoding in Vision Transformers

Transformer models process input tokens as a sequence but do not inherently understand the order or position of those tokens. For images, this becomes a major issue because the spatial arrangement of patches is essential for visual understanding. For example, the position of edges, shapes, and object parts strongly affects how an image is interpreted.

To solve this problem, Vision Transformers add positional embeddings to the patch embeddings. These positional embeddings provide information about the location of each patch in the original image. As a result, the model can distinguish patches that appear in different spatial positions.

In the original ViT architecture, learnable positional embeddings are commonly used. These embeddings are trained along with the rest of the model and help the Transformer learn spatial relationships among image patches. The classification token also receives its own positional embedding because it is included as part of the input token sequence.

The input to the Transformer encoder can therefore be represented as:

$$Z_0 = [x_{cls}; x_1E; x_2E; ...; x_NE] + E_{pos}$$

where x_{cls} is the learnable classification token, $x_1, x_2, ..., x_N$ are the flattened image patches, E is the linear projection matrix, and E_{pos} represents the positional embeddings.

There are two common types of positional encoding:

Learnable Positional Embeddings

In this approach, positional vectors are treated as trainable parameters. The model learns the most suitable positional representation during training. This is the approach used in the original ViT model.

Fixed Sinusoidal Positional Embeddings

Fixed sinusoidal positional encoding was originally used in the Transformer architecture for natural language processing. It uses sine and cosine functions to encode position information in a deterministic way. The encoding is given as:

$$PE(pos, 2i) = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) PE(pos, 2i + 1) = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where pos is the token position, i is the dimension index, and d_{model} is the embedding dimension. Even dimensions use sine functions, while odd dimensions use cosine functions.

This type of encoding is fixed, deterministic, and does not require additional learnable parameters.

2.4 Self-Attention Mechanism

The self-attention mechanism is the central component of the Vision Transformer. It allows each image patch to interact with every other patch in the image. This is different from convolutional operations, which typically focus on local neighborhoods. Through self-attention, the model can capture long-range dependencies and global relationships across the entire image.

For each input token, three vectors are generated using learned linear projections:

- **Query (Q):** represents what a token is looking for
- **Key (K):** represents what information a token provides
- **Value (V):** represents the actual content of the token

The attention score between tokens is calculated using the dot product between the query and key vectors. These scores indicate how strongly one token should attend to another. The scaled dot-product attention is computed as:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimension of the key vectors. The division by $\sqrt{d_k}$ prevents the dot-product values from becoming too large, which helps avoid softmax saturation and stabilizes training.

The softmax operation converts the attention scores into normalized attention weights. These weights are then multiplied by the value vectors to produce a weighted combination of information from all tokens. In this way, each patch token is updated based on its relationship with every other patch in the image.

2.5 Multi-Head Self-Attention

Instead of using a single attention operation, Vision Transformers use Multi-Head Self-Attention. Multi-head attention allows the model to learn different types of relationships among image patches simultaneously. For example, one attention head may focus on local texture, another may focus on object shape, while another may capture global structure.

A single attention head can only learn one type of attention pattern at a time. Multi-head attention overcomes this limitation by running several attention operations in parallel. Each head works in a lower-dimensional representation space, and the outputs of all heads are later combined.

The multi-head attention operation is expressed as:

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \ldots, head_h)W^O$$

where each attention head is computed as:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Here, W_i^Q, W_i^K, and W_i^V are learned projection matrices for the query, key, and value vectors of the i^{th} attention head. W^O is the final output projection matrix that combines the outputs of all attention heads.

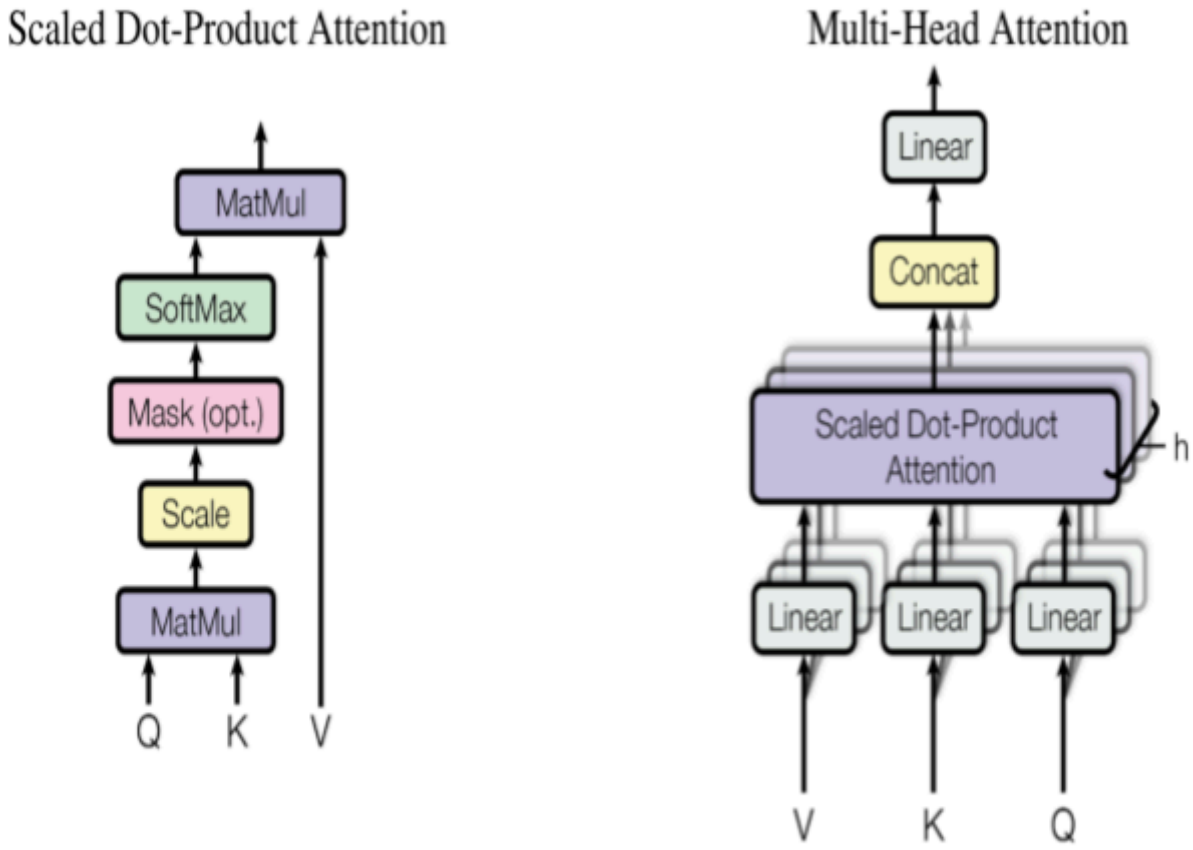


Fig. 2. Scaled dot-product attention and multi-head attention architecture. The left diagram shows scaled dot-product attention, while the right diagram shows multi-head attention, where multiple attention heads operate in parallel, followed by concatenation and a final linear projection. Source: Vaswani *et al.*, "Attention Is All You Need" (2017).

As shown in Fig. 2, the input tokens are first projected into Query, Key, and Value vectors. These projections are then divided into multiple parallel attention heads. Each head

independently applies scaled dot-product attention. The outputs from all heads are concatenated and passed through a final linear layer. This enables the model to learn richer and more diverse visual representations than a single attention head.

2.6 Residual Connections and Layer Normalization

Each Transformer encoder block uses residual connections and layer normalization to improve training stability and information flow.

Residual connections, also called skip connections, add the input of a sub-layer back to its output. This helps preserve information from earlier layers and reduces the risk of gradient degradation in deep networks. Instead of forcing each layer to learn a completely new representation, residual connections allow the model to learn refinements over the previous representation.

Layer normalization is used to stabilize the training process. It normalizes the token features across the embedding dimension, helping maintain well-conditioned gradients and reducing instability during optimization. In modern Vision Transformers, Layer Normalization is commonly applied before the self-attention and MLP sub-layers. This is known as the Pre-LN formulation.

Together, residual connections and layer normalization allow Vision Transformers to be trained effectively even when multiple encoder layers are stacked deeply.

2.7 Classification Token

A learnable classification token, commonly called the CLS token, is added to the beginning of the patch embedding sequence before it enters the Transformer encoder. Unlike patch tokens, the CLS token does not correspond to any specific region of the image. Instead, it is a trainable vector that learns to collect global information from all image patches.

As the token sequence passes through the Transformer encoder, the CLS token attends to all patch tokens through the self-attention mechanism. Gradually, it aggregates information from the entire image. After the final encoder layer, the output representation of the CLS token is passed to the MLP classification head, which produces the final class prediction.

This idea is similar to the CLS token used in BERT for natural language processing. However, in Vision Transformers, the CLS token summarizes visual information rather than textual information.

Some ViT variants use global average pooling instead of a CLS token. In that approach, the final representations of all patch tokens are averaged to obtain a single image-level representation. Both methods are valid, but the original ViT architecture uses the CLS token for classification.

2.8 Model Architecture of Vision Transformer

The Vision Transformer architecture begins by dividing an image into fixed-size patches. Each patch is flattened and projected into an embedding space using a linear projection layer. Positional embeddings are then added to the patch embeddings, allowing the model to retain spatial information about the original image structure.

A learnable CLS token is prepended to the sequence, and the complete sequence is passed through a stack of Transformer encoder blocks. Each encoder block consists of Multi-Head Self-Attention, residual connections, Layer Normalization, and a position-wise Multilayer Perceptron.

Although the Transformer encoder structure is similar to that used in natural language processing, the key difference is that ViT processes image patches instead of word tokens. This allows the Transformer architecture to be applied directly to visual data.

In the original ViT model proposed by Dosovitskiy et al., the ViT-Base configuration uses an embedding dimension of:

$$d_{model} = 768$$

It contains:

$$L = 12$$

Transformer encoder layers and:

$$h = 12$$

attention heads. Larger ViT variants increase the embedding dimension, number of layers, and number of attention heads to improve model capacity.

Fig. 3 shows the overall ViT architecture. The image is first converted into patch embeddings, positional information is added, and the resulting sequence is processed through Transformer encoder layers. The final CLS token representation is then used for classification.

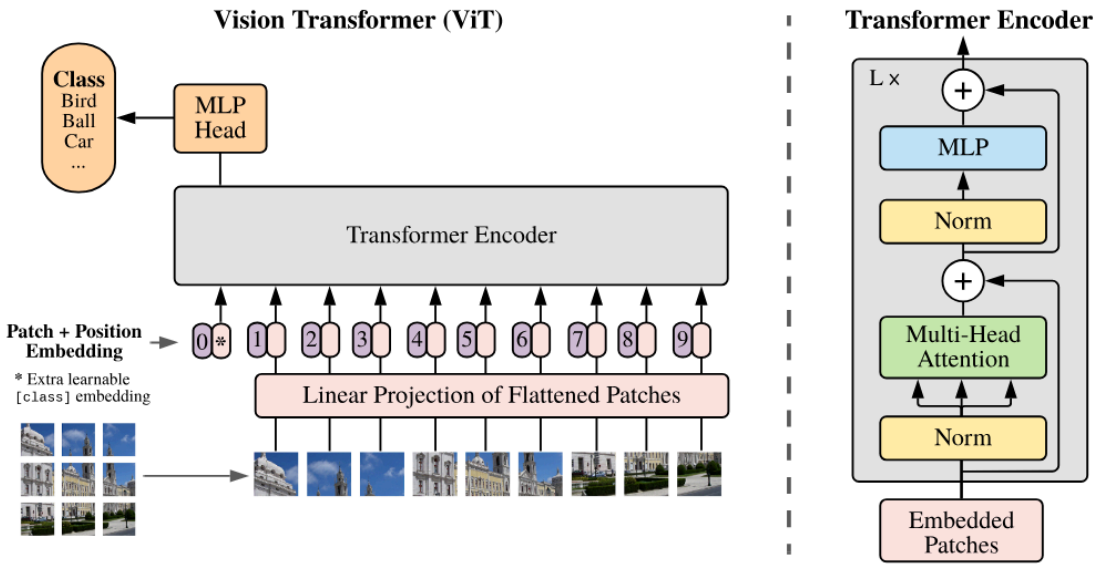


Figure 3. Overview of the Vision Transformer architecture. Source: Dosovitskiy et al., *An Image is Worth 16×16 Words* (2021). The Transformer encoder structure is based on the original Transformer architecture introduced by Vaswani et al., *Attention Is All You Need* (2017).

2.9 Transformer Encoder Block

Each Vision Transformer encoder block applies two main operations to the input token embeddings. The first operation is Multi-Head Self-Attention, and the second is a position-wise MLP. Both operations are combined with residual connections and Layer Normalization.

In the Pre-LN formulation, the first step applies Layer Normalization followed by Multi-Head Self-Attention:

$$Z_l^{'} = MSA(LN(Z_{l-1})) + Z_{l-1}$$

The second step applies Layer Normalization, followed by the MLP:

$$Z_l = MLP(LN(Z_l^{'})) + Z_l^{'}$$

where Z_{l-1} is the input to the l^{th} encoder block, $Z_l^{'}$ is the intermediate output after self-attention, and Z_l is the final output of the encoder block.

The MLP sub-layer usually consists of two fully connected layers with a GELU activation function between them. In the original ViT architecture, the hidden dimension of the MLP is typically four times larger than the embedding dimension:

$$MLP(x) = W_2\, GELU(W_1 x + b_1) + b_2$$

The Multi-Head Self-Attention module captures global dependencies among image patches, while the MLP independently refines each token's representation. Residual connections preserve earlier information, and Layer Normalization stabilizes training. Together, these components form the standard Transformer encoder block used in Vision Transformers.

2.10 Working of the ViT Encoder

The working of the Vision Transformer encoder can be summarized in three main stages.

First, the image is divided into fixed-size patches. Each patch is flattened and projected into a token embedding. A CLS token is added at the beginning of the sequence, and positional embeddings are added to retain spatial information.

Second, the token sequence is passed through a stack of Transformer encoder blocks. Each block applies Layer Normalization, Multi-Head Self-Attention, residual addition, another Layer Normalization, an MLP, and another residual addition. This process is repeated for Llayers, such as 12 layers in ViT-Base.

Finally, after all encoder blocks have processed the token sequence, the final representation of the CLS token is extracted. This representation contains global image-level information and is passed to the MLP classification head to predict the final class label.

Because Vision Transformers do not have the same inductive biases as CNNs, such as local connectivity and translation equivariance, they usually require large amounts of training data when trained from scratch. For this reason, pretrained ViT models are often fine-tuned on smaller datasets such as CIFAR-10 to achieve strong performance with limited data.

2.11 Summary of Core Components

Component	Purpose	Key Insight
Patch Embedding	Converts image patches into token vectors	Bridges the image data and the transformer input format
CLS Token	Aggregates global image representation	Enables classification without spatial pooling
Positional Encoding	Adds spatial location information to patch tokens	Introduces spatial awareness without convolution
Self-Attention	Captures relationships between all patch tokens	Enables global contextual understanding

Multi-Head Attention	Learns multiple relation types in parallel	Captures diverse visual features simultaneously
MLP / Feed-Forward Network	Applies a nonlinear transformation per token	Enhances per-token representation depth
Residual + Layer Norm	Stabilizes training and gradients	Enables deep stacking without degradation

2.12 Use Cases of Vision Transformers

Vision Transformers have been successfully applied across a wide range of computer vision tasks:

- Image Classification: ViT was originally proposed for image classification and achieved highly competitive performance when pretrained on large-scale datasets such as ImageNet-21k and JFT-300M.
- Object Detection: Transformer-based detectors such as DETR formulate object detection as a direct set-prediction problem. DETR combines a CNN backbone with a Transformer encoder-decoder and learned object queries, enabling end-to-end object detection without hand-designed anchor boxes.
- Semantic and Instance Segmentation: Models such as SegFormer and Mask2Former adapt Transformer-based architectures for pixel-level prediction tasks using specialized decoder heads. These models are effective because self-attention captures global spatial context, which is useful for understanding object boundaries and scene layout.
- Medical Image Analysis: ViTs have been applied to histopathology, radiology, and ophthalmology, where modeling long-range spatial relationships across images, scans, or whole-slide images can be valuable.
- Video Understanding: By extending image patches across the temporal dimension, ViT variants such as TimeSformer process videos as sequences of spatiotemporal patches.
- Multi-Modal and Generative Tasks: ViT-based encoders are used in multi-modal models such as CLIP, where image and text representations are aligned in a shared embedding space. In generative AI, Transformer-based and ViT-inspired architectures have also influenced modern image-generation systems, although models such as Stable Diffusion primarily use latent diffusion with U-Net-based denoising and CLIP-based text conditioning.

2.13 Merits of Vision Transformers:

- **Ability to model global context:** Self-attention connects every patch to every other patch, enabling ViTs to capture long-range spatial dependencies that CNNs cannot model directly.

- **Resolution flexibility:** Unlike CNNs with fixed kernel sizes, ViTs can be adapted to different input resolutions by adjusting the number of patches.

- **Strong transfer learning:** When pretrained on large datasets (e.g., ImageNet-21k or JFT-300M), ViTs fine-tune effectively on smaller datasets, achieving competitive or superior accuracy to CNNs.

- **Interpretability**: Attention maps provide a natural way to visualize which image regions the model focuses on when making a prediction, offering greater transparency than CNN feature maps.

2.14 Demerits of Vision Transformers:

- **High computational cost:** The self-attention operation scales quadratically with the number of patches ($O(N^2)$), making ViTs computationally expensive for high-resolution images.

- **Data hungry:** Due to the lack of CNN-style inductive biases (local connectivity, translation equivariance), ViTs require substantially more training data to learn effective representations from scratch.

- **Slower convergence on small datasets:** Without pretraining, ViTs typically underperform CNNs on datasets with fewer than ~100,000 images, as CNNs benefit from their built-in spatial priors.

3. Code and Result

1 STEP 1: Import Libraries

[1]:
```
# Core
import os
import random
from pathlib import Path
import numpy as np

# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset

# Vision / Data
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Metrics
from sklearn.metrics import confusion_matrix, accuracy_score

# Model zoo
import timm

print("All libraries imported successfully.")
```

```
All libraries imported successfully.
```

2 STEP 2: Set Device (CPU / GPU)

[2]:
```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"Using device: {DEVICE}")
if DEVICE.type == "cuda":
    print(f"GPU: {torch.cuda.get_device_name(0)}")
```

```
Using device: cuda
GPU: NVIDIA GeForce RTX 5090
```

3 STEP 3: Define Class Names

[3]:
```
CLASSES = (
    "airplane", "automobile", "bird", "cat", "deer",
    "dog", "frog", "horse", "ship", "truck"
)

NUM_CLASSES = len(CLASSES)
print(f"Number of classes: {NUM_CLASSES}")
for i, name in enumerate(CLASSES):
    print(f" {i} → {name}")
```

```
Number of classes: 10
 0 → airplane
 1 → automobile
 2 → bird
 3 → cat
 4 → deer
 5 → dog
 6 → frog
 7 → horse
 8 → ship
 9 → truck
```

4 STEP 4: Load Raw CIFAR-10 Dataset (No Transforms)

[4]:
```
DATA_ROOT = "./data"

train_dataset_raw = CIFAR10(
    root=DATA_ROOT,
    train=True,
    download=True,
    transform=None
)

test_dataset_raw = CIFAR10(
```

```
    root=DATA_ROOT,
    train=False,
    download=True,
    transform=None
)

print(f"Full training samples: {len(train_dataset_raw)}")
print(f"Full test samples: {len(test_dataset_raw)}")
```

```
Full training samples: 50000
Full test samples: 10000
```

5 STEP 5: Visualize Sample Images

[5]:
```
def show_samples(dataset, classes, n=10):
    plt.figure(figsize=(15, 4))
    for i in range(n):
        img, label = dataset[i]
        plt.subplot(1, n, i + 1)
        plt.imshow(img)
        plt.title(classes[label])
        plt.axis("off")
    plt.show()

show_samples(train_dataset_raw, CLASSES, n=10)
```

6 STEP 6: Check Class Distribution (Raw Dataset)

[6]:
```
from collections import Counter

train_labels = [label for _, label in train_dataset_raw]
label_counts = Counter(train_labels)

for cls_idx, count in label_counts.items():
    print(f"{CLASSES[cls_idx]:10s}: {count}")
```

```
frog      : 5000
truck     : 5000
deer      : 5000
```

```
automobile: 5000
bird      : 5000
horse     : 5000
ship      : 5000
cat       : 5000
dog       : 5000
airplane  : 5000
```

7 STEP 7: Define Normalization Values

[7]:
```
CIFAR_MEAN = (0.4914, 0.4822, 0.4465)
CIFAR_STD  = (0.2023, 0.1994, 0.2010)

print("Normalization values defined.")
```

```
Normalization values defined.
```

8 STEP 8: Define Strong Data Augmentation

[8]:
```
train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(CIFAR_MEAN, CIFAR_STD),
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(CIFAR_MEAN, CIFAR_STD),
])
```

9 STEP 9: Visualize Augmentation Effects

[9]:
```
def show_augmented_samples(dataset, transform, classes, n=8):
    plt.figure(figsize=(15, 4))
    for i in range(n):
        img, label = dataset[i]
        img_aug = transform(img)
        img_aug = img_aug.permute(1, 2, 0)
        img_aug = img_aug * torch.tensor(CIFAR_STD) + torch.tensor(CIFAR_MEAN)
        img_aug = img_aug.clamp(0, 1)

        plt.subplot(1, n, i + 1)
        plt.imshow(img_aug)
        plt.title(classes[label])
        plt.axis("off")
```

```
    plt.show()

show_augmented_samples(train_dataset_raw, train_transform, CLASSES)
```

10 STEP 10: Create Class-Balanced Stratified Subset + DataLoaders

Fix Random Seed (Reproducibility)

[10]:
```
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
```

[10]:
```
<torch._C.Generator at 0x71dafc07ba50>
```

Define Subset Sizes (Per Class)

[11]:
```
TRAIN_PER_CLASS = 2000
VAL_PER_CLASS   = 500
TEST_PER_CLASS  = 500
```

Stratified Sampling Function

[12]:
```
def stratified_split(dataset, train_n, val_n, test_n):
    class_indices = {i: [] for i in range(NUM_CLASSES)}

    for idx, (_, label) in enumerate(dataset):
        class_indices[label].append(idx)

    train_idx, val_idx, test_idx = [], [], []

    for cls, indices in class_indices.items():
        random.shuffle(indices)
        train_idx.extend(indices[:train_n])
        val_idx.extend(indices[train_n:train_n + val_n])
        test_idx.extend(indices[train_n + val_n:train_n + val_n + test_n])

    return train_idx, val_idx, test_idx
```

Create Subset Indices

[13]:
```
train_idx, val_idx, subset_test_idx = stratified_split(
    train_dataset_raw,
    TRAIN_PER_CLASS,
    VAL_PER_CLASS,
    TEST_PER_CLASS
)

print(f"Train subset size: {len(train_idx)}")
print(f"Val subset size: {len(val_idx)}")
print(f"Subset test size: {len(subset_test_idx)}")
```

```
Train subset size: 20000
Val subset size: 5000
Subset test size: 5000
```

Create Dataset Objects with Transforms

[14]:
```
print("\n" + "="*70)
print("CREATING DATASET SPLITS")
print("="*70)

# ------------------------------
# Training Dataset
# ------------------------------
train_dataset = Subset(
    CIFAR10(root=DATA_ROOT, train=True, transform=train_transform),
    train_idx
)

print("\n[TRAIN DATASET]")
print(f"• Source           : CIFAR-10 (train split)")
print(f"• Samples          : {len(train_dataset)}")
print(f"• Augmentation     : ENABLED")
print(f"• Purpose          : Model learning")

# ------------------------------
# Validation Dataset
# ------------------------------
val_dataset = Subset(
    CIFAR10(root=DATA_ROOT, train=True, transform=test_transform),
    val_idx
)

print("\n[VALIDATION DATASET]")
print(f"• Source           : CIFAR-10 (train split)")
print(f"• Samples          : {len(val_dataset)}")
print(f"• Augmentation     : DISABLED (only normalization)")
print(f"• Purpose          : Early stopping & hyperparameter tuning")
```

```python
# ----------------------------
# Subset Test Dataset
# ----------------------------
subset_test_dataset = Subset(
    CIFAR10(root=DATA_ROOT, train=True, transform=test_transform),
    subset_test_idx
)

print("\n[SUBSET TEST DATASET]")
print(f"• Source           : CIFAR-10 (train split)")
print(f"• Samples          : {len(subset_test_dataset)}")
print(f"• Augmentation     : DISABLED (only normalization)")
print(f"• Purpose          : Fair evaluation on balanced subset")

# ----------------------------
# Full CIFAR-10 Test Dataset
# ----------------------------
full_test_dataset = CIFAR10(
    root=DATA_ROOT,
    train=False,
    transform=test_transform
)

print("\n[FULL TEST DATASET]")
print(f"• Source           : CIFAR-10 (official test split)")
print(f"• Samples          : {len(full_test_dataset)}")
print(f"• Augmentation     : DISABLED (only normalization)")
print(f"• Purpose          : Final generalization evaluation")

print("\n" + "="*70)
print("DATASET SETUP COMPLETE")
print("="*70)
```

```
======================================================================
CREATING DATASET SPLITS
======================================================================


[TRAIN DATASET]
• Source           : CIFAR-10 (train split)
• Samples          : 20000
• Augmentation     : ENABLED
• Purpose          : Model learning

[VALIDATION DATASET]
• Source           : CIFAR-10 (train split)
```

```
• Samples          : 5000
• Augmentation     : DISABLED (only normalization)
• Purpose          : Early stopping & hyperparameter tuning

[SUBSET TEST DATASET]
• Source           : CIFAR-10 (train split)
• Samples          : 5000
• Augmentation     : DISABLED (only normalization)
• Purpose          : Fair evaluation on balanced subset

[FULL TEST DATASET]
• Source           : CIFAR-10 (official test split)
• Samples          : 10000
• Augmentation     : DISABLED (only normalization)
• Purpose          : Final generalization evaluation

======================================================================
DATASET SETUP COMPLETE
======================================================================
```

11 Create DataLoaders

[15]:
```python
print("\n" + "="*70)
print("CREATING DATALOADERS")
print("="*70)

# ----------------------------
# DataLoader Configuration
# ----------------------------
BATCH_SIZE = 128
NUM_WORKERS = 4

print("\n[DATALOADER SETTINGS]")
print(f"• Batch size        : {BATCH_SIZE}")
print(f"• Num workers       : {NUM_WORKERS}")
print(f"• Pin memory        : Enabled (faster GPU transfer)")

# ----------------------------
# Training DataLoader
# ----------------------------
train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=NUM_WORKERS,
    pin_memory=True
)
```

```
print("\n[TRAIN DATALOADER]")
print(f"• Total samples     : {len(train_dataset)}")
print(f"• Total batches     : {len(train_loader)}")
print(f"• Shuffling         : Enabled")
print(f"• Purpose           : Model learning (each epoch sees data in new␣
 ↪order)")

# ----------------------------
# Validation DataLoader
# ----------------------------
val_loader = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=NUM_WORKERS,
    pin_memory=True
)

print("\n[VALIDATION DATALOADER]")
print(f"• Total samples     : {len(val_dataset)}")
print(f"• Total batches     : {len(val_loader)}")
print(f"• Shuffling         : Disabled")
print(f"• Purpose           : Stable validation & early stopping")

# ----------------------------
# Subset Test DataLoader
# ----------------------------
subset_test_loader = DataLoader(
    subset_test_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=NUM_WORKERS,
    pin_memory=True
)

print("\n[SUBSET TEST DATALOADER]")
print(f"• Total samples     : {len(subset_test_dataset)}")
print(f"• Total batches     : {len(subset_test_loader)}")
print(f"• Shuffling         : Disabled")
print(f"• Purpose           : Fair evaluation on balanced subset")

# ----------------------------
# Full CIFAR-10 Test DataLoader
# ----------------------------
full_test_loader = DataLoader(
    full_test_dataset,
```

```
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=NUM_WORKERS,
    pin_memory=True
)

print("\n[FULL TEST DATALOADER]")
print(f"• Total samples     : {len(full_test_dataset)}")
print(f"• Total batches     : {len(full_test_loader)}")
print(f"• Shuffling         : Disabled")
print(f"• Purpose           : Final generalization evaluation")

print("\n" + "="*70)
print("DATALOADERS READY")
print("="*70)
```

```
======================================================================
CREATING DATALOADERS
======================================================================

[DATALOADER SETTINGS]
• Batch size        : 128
• Num workers       : 4
• Pin memory        : Enabled (faster GPU transfer)

[TRAIN DATALOADER]
• Total samples     : 20000
• Total batches     : 157
• Shuffling         : Enabled
• Purpose           : Model learning (each epoch sees data in new order)

[VALIDATION DATALOADER]
• Total samples     : 5000
• Total batches     : 40
• Shuffling         : Disabled
• Purpose           : Stable validation & early stopping

[SUBSET TEST DATALOADER]
• Total samples     : 5000
• Total batches     : 40
• Shuffling         : Disabled
• Purpose           : Fair evaluation on balanced subset

[FULL TEST DATALOADER]
• Total samples     : 10000
• Total batches     : 79
• Shuffling         : Disabled
```

```
• Purpose           : Final generalization evaluation

======================================================================
DATALOADERS READY
======================================================================
```

[16]:
```python
print("\nSANITY CHECK: ONE TRAINING BATCH")

images, labels = next(iter(train_loader))
print(f"• Image batch shape : {images.shape}")
print(f"• Label batch shape : {labels.shape}")
print("Expected image shape: (batch_size, 3, 32, 32)")
print("Expected label shape: (batch_size,)")

if images.shape[1:] == (3, 32, 32):
    print("Image tensor shape is correct")
else:
    print("Image tensor shape is incorrect")
```

```
SANITY CHECK: ONE TRAINING BATCH
• Image batch shape : torch.Size([128, 3, 32, 32])
• Label batch shape : torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Expected label shape: (batch_size,)
Image tensor shape is correct
```

12 STEP 11: Build the Pretrained ViT-Small-Patch8 Model

**What this cell does (student view)

Loads a Vision Transformer pretrained on ImageNet and prepares it for CIFAR-10 classification.**

[17]:
```python
MODEL_NAME = "vit_small_patch8_224"
```

[18]:
```python
print("\n" + "="*80)
print("STEP 11 (FIXED): BUILDING PRETRAINED ViT-SMALL FOR CIFAR-10")
print("="*80)


model = timm.create_model(
    MODEL_NAME,
    pretrained=True,
    num_classes=NUM_CLASSES,
    img_size=32
)

model = model.to(DEVICE)
```

```
print("\n[MODEL LOADING CONFIRMATION]")
print(f"• Model name           : {MODEL_NAME}")
print("• Pretrained weights  : ImageNet")
print("• Image size override : 32×32 (CIFAR-10)")
print("• Patch size          : 8×8")
print("• Number of classes   : 10")
print(f"• Model device         : {DEVICE}")
```

```
================================================================================
STEP 11 (FIXED): BUILDING PRETRAINED ViT-SMALL FOR CIFAR-10
================================================================================


[MODEL LOADING CONFIRMATION]
• Model name           : vit_small_patch8_224
• Pretrained weights  : ImageNet
• Image size override : 32×32 (CIFAR-10)
• Patch size          : 8×8
• Number of classes   : 10
• Model device         : cuda
```

[19]:
```
print("\n" + "="*70)
print("VERIFYING MODEL INPUT COMPATIBILITY")
print("="*70)

dummy_input = torch.randn(1, 3, 32, 32).to(DEVICE)

try:
    dummy_output = model(dummy_input)
    print("  Model successfully accepted 32×32 input")
    print(f"• Output shape: {dummy_output.shape}")
    print("Expected shape: (1, 10)")
except Exception as e:
    print("  Model failed on 32×32 input")
    print(e)
```

```
======================================================================
VERIFYING MODEL INPUT COMPATIBILITY
======================================================================
  Model successfully accepted 32×32 input
• Output shape: torch.Size([1, 10])
Expected shape: (1, 10)
```

13 Verify Patch Embedding Details (VERY IMPORTANT)

[20]:
```
print("\n[PATCH EMBEDDING VERIFICATION]")

patch_size = model.patch_embed.patch_size
embed_dim = model.embed_dim

print(f"• Patch size from model      : {patch_size}")
print(f"• Embedding dimension        : {embed_dim}")

expected_patches = (32 // patch_size[0]) ** 2
print(f"• Expected patches per image : {expected_patches}")
print("• Expected tokens (+ CLS)    :", expected_patches + 1)

print("\nExplanation:")
print("• CIFAR-10 image size = 32×32")
print("• Patch size = 8×8 → (32/8)² = 16 patches")
print("• +1 CLS token → total 17 tokens")

if expected_patches == 16:
    print(" Patch-to-token mapping is correct")
else:
    print(" Patch-to-token mapping is incorrect")
```

```
[PATCH EMBEDDING VERIFICATION]
• Patch size from model      : (8, 8)
• Embedding dimension        : 384
• Expected patches per image : 16
• Expected tokens (+ CLS)    : 17

Explanation:
• CIFAR-10 image size = 32×32
• Patch size = 8×8 → (32/8)² = 16 patches
• +1 CLS token → total 17 tokens
 Patch-to-token mapping is correct
```

14 Verify CLS Token Exists

The CLS token learns a global image representation used for classification.

[21]:
```
print("\n[CLS TOKEN CHECK]")

if hasattr(model, "cls_token"):
    print(" CLS token is present in the model")
    print(f"CLS token shape: {model.cls_token.shape}")
else:
```

```
    print(" CLS token is missing (this should NOT happen)")
```

```
[CLS TOKEN CHECK]
 CLS token is present in the model
CLS token shape: torch.Size([1, 1, 384])
```

15 STEP 12: Count Model Parameters

[22]:
```
print("\n" + "="*80)
print("STEP 12: COUNTING MODEL PARAMETERS")
print("="*80)

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:,}")

print("\nExpected:")
print("• ViT-Small   22 million parameters")

if total_params > 20_000_000:
    print(" Parameter count matches ViT-Small")
else:
    print("Unexpected parameter count")
```

```
================================================================================
STEP 12: COUNTING MODEL PARAMETERS
================================================================================
• Total parameters      : 21,379,210
• Trainable parameters  : 21,379,210

Expected:
• ViT-Small   22 million parameters
 Parameter count matches ViT-Small
```

16 STEP 13: Print Model Architecture

[23]:
```
print("\n" + "="*80)
print("STEP 13: MODEL ARCHITECTURE OVERVIEW")
print("="*80)

print(model)
```

```
================================================================================
STEP 13: MODEL ARCHITECTURE OVERVIEW
================================================================================
VisionTransformer(
  (patch_embed): PatchEmbed(
    (proj): Conv2d(3, 384, kernel_size=(8, 8), stride=(8, 8))
    (norm): Identity()
  )
  (pos_drop): Dropout(p=0.0, inplace=False)
  (patch_drop): Identity()
  (norm_pre): Identity()
  (blocks): Sequential(
    (0): Block(
      (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
      (attn): Attention(
        (qkv): Linear(in_features=384, out_features=1152, bias=True)
        (q_norm): Identity()
        (k_norm): Identity()
        (attn_drop): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (proj): Linear(in_features=384, out_features=384, bias=True)
        (proj_drop): Dropout(p=0.0, inplace=False)
      )
      (ls1): Identity()
      (drop_path1): Identity()
      (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=384, out_features=1536, bias=True)
        (act): GELU(approximate='none')
        (drop1): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (fc2): Linear(in_features=1536, out_features=384, bias=True)
        (drop2): Dropout(p=0.0, inplace=False)
      )
      (ls2): Identity()
      (drop_path2): Identity()
    )
    (1): Block(
      (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
      (attn): Attention(
        (qkv): Linear(in_features=384, out_features=1152, bias=True)
        (q_norm): Identity()
        (k_norm): Identity()
        (attn_drop): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (proj): Linear(in_features=384, out_features=384, bias=True)
        (proj_drop): Dropout(p=0.0, inplace=False)
      )
```

```
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(2): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(3): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
```

```
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(4): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(5): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
```

```
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(6): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(7): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
```

```
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(8): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(9): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
```

```
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(10): Block(
  (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (attn): Attention(
    (qkv): Linear(in_features=384, out_features=1152, bias=True)
    (q_norm): Identity()
    (k_norm): Identity()
    (attn_drop): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (proj): Linear(in_features=384, out_features=384, bias=True)
    (proj_drop): Dropout(p=0.0, inplace=False)
  )
  (ls1): Identity()
  (drop_path1): Identity()
  (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (mlp): Mlp(
    (fc1): Linear(in_features=384, out_features=1536, bias=True)
    (act): GELU(approximate='none')
    (drop1): Dropout(p=0.0, inplace=False)
    (norm): Identity()
    (fc2): Linear(in_features=1536, out_features=384, bias=True)
    (drop2): Dropout(p=0.0, inplace=False)
  )
  (ls2): Identity()
  (drop_path2): Identity()
)
(11): Block(
```

```
      (norm1): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
      (attn): Attention(
        (qkv): Linear(in_features=384, out_features=1152, bias=True)
        (q_norm): Identity()
        (k_norm): Identity()
        (attn_drop): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (proj): Linear(in_features=384, out_features=384, bias=True)
        (proj_drop): Dropout(p=0.0, inplace=False)
      )
      (ls1): Identity()
      (drop_path1): Identity()
      (norm2): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=384, out_features=1536, bias=True)
        (act): GELU(approximate='none')
        (drop1): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (fc2): Linear(in_features=1536, out_features=384, bias=True)
        (drop2): Dropout(p=0.0, inplace=False)
      )
      (ls2): Identity()
      (drop_path2): Identity()
    )
  )
  (norm): LayerNorm((384,), eps=1e-06, elementwise_affine=True)
  (fc_norm): Identity()
  (head_drop): Dropout(p=0.0, inplace=False)
  (head): Linear(in_features=384, out_features=10, bias=True)
)
```

17 STEP 14: Define Loss Function and Optimizer

[24]:
```python
print("\n" + "="*80)
print("STEP 14: DEFINING LOSS FUNCTION & OPTIMIZER")
print("="*80)

LABEL_SMOOTHING = 0.1

criterion = nn.CrossEntropyLoss(label_smoothing=LABEL_SMOOTHING)

print("\n[LOSS FUNCTION]")
print("• Type              : CrossEntropyLoss")
print(f"• Label smoothing   : {LABEL_SMOOTHING}")
print("• Purpose           : Prevent overconfident predictions")
print("• Benefit           : Better generalization for Transformers")
```

```
================================================================================
STEP 14: DEFINING LOSS FUNCTION & OPTIMIZER
================================================================================

[LOSS FUNCTION]
• Type              : CrossEntropyLoss
• Label smoothing   : 0.1
• Purpose           : Prevent overconfident predictions
• Benefit           : Better generalization for Transformers
```

Optimizer (AdamW)

[25]:
```
LEARNING_RATE = 3e-4
WEIGHT_DECAY = 0.05

optimizer = optim.AdamW(
    model.parameters(),
    lr=LEARNING_RATE,
    weight_decay=WEIGHT_DECAY
)

print("\n[OPTIMIZER]")
print("• Optimizer         : AdamW")
print(f"• Learning rate     : {LEARNING_RATE}")
print(f"• Weight decay      : {WEIGHT_DECAY}")
print("• Why AdamW?        : Correct decoupled weight decay (better for ViT)")
```

```
[OPTIMIZER]
• Optimizer         : AdamW
• Learning rate     : 0.0003
• Weight decay      : 0.05
• Why AdamW?        : Correct decoupled weight decay (better for ViT)
```

[26]:
```
print("\n[OPTIMIZER SANITY CHECK]")
print(f"• Number of parameter groups: {len(optimizer.param_groups)}")
print("Expected: 1 parameter group")

if len(optimizer.param_groups) == 1:
    print("  Optimizer correctly attached to model parameters")
else:
    print("  Unexpected optimizer configuration")
```

```
[OPTIMIZER SANITY CHECK]
• Number of parameter groups: 1
Expected: 1 parameter group
  Optimizer correctly attached to model parameters
```

18 STEP 15: Define Learning Rate Scheduler

Controls how fast the model learns over time.

19 Cosine Annealing Scheduler

[27]:
```
print("\n" + "="*80)
print("STEP 15: DEFINING LEARNING RATE SCHEDULER")
print("="*80)

MAX_EPOCHS = 100  # upper bound (early stopping will stop earlier)

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

print("\n[LEARNING RATE SCHEDULER]")
print("• Scheduler type    : CosineAnnealingLR")
print(f"• Maximum epochs    : {MAX_EPOCHS}")
print("• LR behavior       : Gradually decays from initial LR to near zero")
print("• Why cosine?       : Smooth decay improves transformer convergence")
```

```
================================================================================
STEP 15: DEFINING LEARNING RATE SCHEDULER
================================================================================

[LEARNING RATE SCHEDULER]
• Scheduler type    : CosineAnnealingLR
• Maximum epochs    : 100
• LR behavior       : Gradually decays from initial LR to near zero
• Why cosine?       : Smooth decay improves transformer convergence
```

20 STEP 16: Define Training Function

[28]:
```
print("\n" + "="*80)
print("STEP 16: DEFINING TRAINING FUNCTION (MixUp + CutMix)")
print("="*80)

def mixup_data(x, y, alpha=1.0):
    lam = np.random.beta(alpha, alpha)
    batch_size = x.size(0)
    index = torch.randperm(batch_size).to(x.device)

    mixed_x = lam * x + (1 - lam) * x[index]
    y_a, y_b = y, y[index]
```

```
    return mixed_x, y_a, y_b, lam


def cutmix_data(x, y, alpha=1.0):
    lam = np.random.beta(alpha, alpha)
    batch_size, _, H, W = x.size()
    index = torch.randperm(batch_size).to(x.device)

    cx = np.random.randint(W)
    cy = np.random.randint(H)
    w = int(W * np.sqrt(1 - lam))
    h = int(H * np.sqrt(1 - lam))

    x1 = np.clip(cx - w // 2, 0, W)
    x2 = np.clip(cx + w // 2, 0, W)
    y1 = np.clip(cy - h // 2, 0, H)
    y2 = np.clip(cy + h // 2, 0, H)

    x[:, :, y1:y2, x1:x2] = x[index, :, y1:y2, x1:x2]
    lam = 1 - ((x2 - x1) * (y2 - y1) / (W * H))

    y_a, y_b = y, y[index]
    return x, y_a, y_b, lam
```

```
================================================================================
STEP 16: DEFINING TRAINING FUNCTION (MixUp + CutMix)
================================================================================
```

[29]:
```
def train_one_epoch(model, loader, optimizer, criterion, device, epoch):
    model.train()

    running_loss = 0.0
    correct = 0
    total = 0

    print("\n" + "-"*60)
    print(f"Training Epoch {epoch}")
    print("-"*60)

    for batch_idx, (images, labels) in enumerate(loader):
        images, labels = images.to(device), labels.to(device)

        # Randomly choose MixUp or CutMix
        use_cutmix = np.random.rand() < 0.5

        if use_cutmix:
```

```
            images, y_a, y_b, lam = cutmix_data(images, labels)
            aug_type = "CutMix"
        else:
            images, y_a, y_b, lam = mixup_data(images, labels)
            aug_type = "MixUp"

        optimizer.zero_grad()
        outputs = model(images)

        loss = lam * criterion(outputs, y_a) + (1 - lam) * criterion(outputs,␣
↪y_b)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = outputs.max(1)

        total += labels.size(0)
        correct += (lam * predicted.eq(y_a).sum().item() +
                    (1 - lam) * predicted.eq(y_b).sum().item())

        if batch_idx == 0:
            print(f"First batch augmentation used: {aug_type}")
            print(f"• Batch image shape: {images.shape}")
            print(f"• Batch label shape: {labels.shape}")
            print("Expected image shape: (batch_size, 3, 32, 32)")

    epoch_loss = running_loss / len(loader)
    epoch_acc = 100. * correct / total

    print(f"Epoch {epoch} Training Loss: {epoch_loss:.4f}")
    print(f"Epoch {epoch} Training Accuracy: {epoch_acc:.2f}%")

    return epoch_loss, epoch_acc
```

21 STEP 17: Define Evaluation Function

[30]:
```
print("\n" + "="*80)
print("STEP 17: DEFINING EVALUATION FUNCTION")
print("="*80)

def evaluate(model, loader, criterion, device, split_name="Validation"):
    model.eval()

    running_loss = 0.0
    correct = 0
```

```
    total = 0

    all_preds = []
    all_labels = []

    print(f"\nEvaluating on {split_name} set...")

    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item()
            _, predicted = outputs.max(1)

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

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

    avg_loss = running_loss / len(loader)
    acc = 100. * correct / total

    print(f"{split_name} Loss     : {avg_loss:.4f}")
    print(f"{split_name} Accuracy : {acc:.2f}%")

    return avg_loss, acc, all_labels, all_preds
```

```
================================================================================
STEP 17: DEFINING EVALUATION FUNCTION
================================================================================
```

22 STEP 18: Define Early Stopping Logic

[31]:
```
print("\n" + "="*80)
print("STEP 18: DEFINING EARLY STOPPING")
print("="*80)

class EarlyStopping:
    def __init__(self, patience=8, min_delta=0.0):
        self.patience = patience
        self.min_delta = min_delta
        self.best_acc = 0.0
```

```
        self.counter = 0
        self.should_stop = False

        print("\n[EARLY STOPPING CONFIGURATION]")
        print(f"• Patience   : {patience} epochs")
        print(f"• Min delta  : {min_delta}")
        print("• Monitored  : Validation accuracy")

    def step(self, val_acc):
        if val_acc > self.best_acc + self.min_delta:
            print(f"  Validation accuracy improved: {self.best_acc:.2f}% →␣
↪{val_acc:.2f}%")
            self.best_acc = val_acc
            self.counter = 0
        else:
            self.counter += 1
            print(f"  No improvement. Counter: {self.counter}/{self.patience}")

        if self.counter >= self.patience:
            self.should_stop = True
            print("Early stopping triggered")

        return self.should_stop
```

```
================================================================================
STEP 18: DEFINING EARLY STOPPING
================================================================================
```

23 STEP 19: Two-Stage Fine-Tuning

[32]:
```
print("\n" + "="*80)
print("STEP 19: TWO-STAGE FINE-TUNING")
print("="*80)

print("\n[STAGE 1: FREEZING BACKBONE]")
for name, param in model.named_parameters():
    if "head" not in name:
        param.requires_grad = False

trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f"• Trainable parameters after freezing: {trainable_params:,}")
print("• Only classification head is trainable")
```

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

```
STEP 19: TWO-STAGE FINE-TUNING
================================================================================

[STAGE 1: FREEZING BACKBONE]
• Trainable parameters after freezing: 3,850
• Only classification head is trainable
```

[33]:
```python
optimizer = optim.AdamW(
    filter(lambda p: p.requires_grad, model.parameters()),
    lr=LEARNING_RATE,
    weight_decay=WEIGHT_DECAY
)

print("Optimizer reconfigured for Stage 1")
```

```
Optimizer reconfigured for Stage 1
```

[34]:
```python
def unfreeze_model(model, n_blocks=12):
    if n_blocks >= 12:
        for param in model.parameters():
            param.requires_grad = True
        print("\n[STAGE 2: UNFREEZING FULL MODEL]")
    else:
        # Freeze all parameters first
        for param in model.parameters():
            param.requires_grad = False

        # Unfreeze the classification head
        for param in model.heads.parameters():
            param.requires_grad = True

        # Unfreeze the last n_blocks of the encoder
        if n_blocks > 0:
            layers = list(model.encoder.layers)
            for layer in layers[-n_blocks:]:
                for param in layer.parameters():
                    param.requires_grad = True

        print(f"\n[STAGE 2: UNFREEZING LAST {n_blocks} BLOCKS + HEAD]")

    trainable = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print(f"• Trainable parameters now: {trainable:,}")
    if n_blocks >= 12:
        print("• Full fine-tuning enabled")
```

24 STEP 20: Create Output Directory

[35]:
```
print("\n" + "="*80)
print("STEP 20: CREATING OUTPUT DIRECTORY")
print("="*80)

OUTPUT_DIR = Path("vit_cifar10_experiment")
OUTPUT_DIR.mkdir(exist_ok=True)

print(f"• Output directory created at: {OUTPUT_DIR.resolve()}")
```

```
================================================================================
STEP 20: CREATING OUTPUT DIRECTORY
================================================================================
• Output directory created at:
/home/onkar/.jackt/sha/vlab/exp2_vit/vit_cifar10_experiment
```

[36]:
```
print("\n" + "="*80)
print("STEP 21: STARTING TRAINING")
print("="*80)

EPOCHS_STAGE1 = 10      # head training
EPOCHS_STAGE2 = 90      # full fine-tuning (upper bound)

early_stopper = EarlyStopping(patience=8)

history = {
    "train_loss": [],
    "train_acc": [],
    "val_loss": [],
    "val_acc": []
}
```

```
================================================================================
STEP 21: STARTING TRAINING
================================================================================

[EARLY STOPPING CONFIGURATION]
• Patience   : 8 epochs
• Min delta  : 0.0
• Monitored  : Validation accuracy
```

[37]:
```
print("\n" + "="*60)
print("STAGE 1 TRAINING: CLASSIFIER HEAD ONLY")
print("="*60)
```

```python
for epoch in range(1, EPOCHS_STAGE1 + 1):
    train_loss, train_acc = train_one_epoch(
        model, train_loader, optimizer, criterion, DEVICE, epoch
    )

    val_loss, val_acc, _, _ = evaluate(
        model, val_loader, criterion, DEVICE, split_name="Validation"
    )

    history["train_loss"].append(train_loss)
    history["train_acc"].append(train_acc)
    history["val_loss"].append(val_loss)
    history["val_acc"].append(val_acc)
```

```
============================================================
STAGE 1 TRAINING: CLASSIFIER HEAD ONLY
============================================================

------------------------------------------------------------
Training Epoch 1
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 1 Training Loss: 2.2682
Epoch 1 Training Accuracy: 25.63%

Evaluating on Validation set…
Validation Loss     : 1.5174
Validation Accuracy : 55.62%

------------------------------------------------------------
Training Epoch 2
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 2 Training Loss: 1.8594
Epoch 2 Training Accuracy: 41.58%

Evaluating on Validation set…
Validation Loss     : 1.2533
Validation Accuracy : 67.54%

------------------------------------------------------------
```

```
Training Epoch 3
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 3 Training Loss: 1.7678
Epoch 3 Training Accuracy: 46.08%

Evaluating on Validation set…
Validation Loss     : 1.1710
Validation Accuracy : 72.02%

------------------------------------------------------------
Training Epoch 4
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 4 Training Loss: 1.7490
Epoch 4 Training Accuracy: 47.31%

Evaluating on Validation set…
Validation Loss     : 1.1260
Validation Accuracy : 73.98%

------------------------------------------------------------
Training Epoch 5
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 5 Training Loss: 1.7498
Epoch 5 Training Accuracy: 47.25%

Evaluating on Validation set…
Validation Loss     : 1.1148
Validation Accuracy : 74.80%

------------------------------------------------------------
Training Epoch 6
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
```

```
Epoch 6 Training Loss: 1.7451
Epoch 6 Training Accuracy: 47.29%

Evaluating on Validation set…
Validation Loss     : 1.1045
Validation Accuracy : 75.38%

------------------------------------------------------------
Training Epoch 7
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 7 Training Loss: 1.7098
Epoch 7 Training Accuracy: 49.13%

Evaluating on Validation set…
Validation Loss     : 1.0815
Validation Accuracy : 76.48%

------------------------------------------------------------
Training Epoch 8
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 8 Training Loss: 1.7097
Epoch 8 Training Accuracy: 49.10%

Evaluating on Validation set…
Validation Loss     : 1.0818
Validation Accuracy : 76.66%

------------------------------------------------------------
Training Epoch 9
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 9 Training Loss: 1.6966
Epoch 9 Training Accuracy: 49.64%

Evaluating on Validation set…
Validation Loss     : 1.0686
Validation Accuracy : 77.56%
```

```
------------------------------------------------------------
Training Epoch 10
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 10 Training Loss: 1.7055
Epoch 10 Training Accuracy: 49.35%

Evaluating on Validation set…
Validation Loss     : 1.0750
Validation Accuracy : 76.44%
```

25 Unfreeze Model for Stage 2

[38]:
```
NUM_BLOCKS_TO_UNFREEZE = 12
```

[39]:
```
unfreeze_model(model, NUM_BLOCKS_TO_UNFREEZE)

optimizer = optim.AdamW(
    model.parameters(),
    lr=LEARNING_RATE / 10,   # smaller LR for fine-tuning
    weight_decay=WEIGHT_DECAY
)

scheduler = optim.lr_scheduler.CosineAnnealingLR(
    optimizer, T_max=EPOCHS_STAGE2
)

print(" Optimizer and scheduler reconfigured for Stage 2")
```

```
[STAGE 2: UNFREEZING FULL MODEL]
• Trainable parameters now: 21,379,210
• Full fine-tuning enabled
 Optimizer and scheduler reconfigured for Stage 2
```

26 Stage 2 Training Loop (Full Fine-Tuning)

[40]:
```
print("\n" + "="*60)
print("STAGE 2 TRAINING: FULL MODEL FINE-TUNING")
print("="*60)

for epoch in range(1, EPOCHS_STAGE2 + 1):
```

```
    train_loss, train_acc = train_one_epoch(
        model, train_loader, optimizer, criterion, DEVICE, epoch
    )

    val_loss, val_acc, _, _ = evaluate(
        model, val_loader, criterion, DEVICE, split_name="Validation"
    )

    scheduler.step()

    history["train_loss"].append(train_loss)
    history["train_acc"].append(train_acc)
    history["val_loss"].append(val_loss)
    history["val_acc"].append(val_acc)

    if early_stopper.step(val_acc):
        print(f"\nTraining stopped early at epoch {epoch}")
        break
```

```
============================================================
STAGE 2 TRAINING: FULL MODEL FINE-TUNING
============================================================

------------------------------------------------------------
Training Epoch 1
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 1 Training Loss: 1.5969
Epoch 1 Training Accuracy: 54.91%

Evaluating on Validation set…
Validation Loss     : 0.8358
Validation Accuracy : 88.82%
  Validation accuracy improved: 0.00% → 88.82%

------------------------------------------------------------
Training Epoch 2
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 2 Training Loss: 1.5109
Epoch 2 Training Accuracy: 59.44%
```

```
Evaluating on Validation set…
Validation Loss     : 0.8063
Validation Accuracy : 90.10%
  Validation accuracy improved: 88.82% → 90.10%

------------------------------------------------------------
Training Epoch 3
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 3 Training Loss: 1.4261
Epoch 3 Training Accuracy: 63.55%

Evaluating on Validation set…
Validation Loss     : 0.7615
Validation Accuracy : 91.44%
  Validation accuracy improved: 90.10% → 91.44%

------------------------------------------------------------
Training Epoch 4
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 4 Training Loss: 1.4265
Epoch 4 Training Accuracy: 63.31%

Evaluating on Validation set…
Validation Loss     : 0.7535
Validation Accuracy : 92.60%
  Validation accuracy improved: 91.44% → 92.60%

------------------------------------------------------------
Training Epoch 5
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 5 Training Loss: 1.4128
Epoch 5 Training Accuracy: 63.83%

Evaluating on Validation set…
Validation Loss     : 0.7626
```

```
Validation Accuracy : 92.00%
  No improvement. Counter: 1/8

------------------------------------------------------------
Training Epoch 6
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 6 Training Loss: 1.4342
Epoch 6 Training Accuracy: 61.73%

Evaluating on Validation set…
Validation Loss     : 0.7232
Validation Accuracy : 92.82%
  Validation accuracy improved: 92.60% → 92.82%

------------------------------------------------------------
Training Epoch 7
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 7 Training Loss: 1.3316
Epoch 7 Training Accuracy: 66.92%

Evaluating on Validation set…
Validation Loss     : 0.7111
Validation Accuracy : 93.28%
  Validation accuracy improved: 92.82% → 93.28%

------------------------------------------------------------
Training Epoch 8
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 8 Training Loss: 1.3065
Epoch 8 Training Accuracy: 68.54%

Evaluating on Validation set…
Validation Loss     : 0.6956
Validation Accuracy : 93.90%
  Validation accuracy improved: 93.28% → 93.90%
```

```
------------------------------------------------------------
Training Epoch 9
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 9 Training Loss: 1.3324
Epoch 9 Training Accuracy: 66.31%

Evaluating on Validation set…
Validation Loss     : 0.6903
Validation Accuracy : 93.86%
  No improvement. Counter: 1/8

------------------------------------------------------------
Training Epoch 10
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 10 Training Loss: 1.3292
Epoch 10 Training Accuracy: 67.59%

Evaluating on Validation set…
Validation Loss     : 0.7070
Validation Accuracy : 93.52%
  No improvement. Counter: 2/8

------------------------------------------------------------
Training Epoch 11
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 11 Training Loss: 1.3032
Epoch 11 Training Accuracy: 67.70%

Evaluating on Validation set…
Validation Loss     : 0.6703
Validation Accuracy : 93.72%
  No improvement. Counter: 3/8

------------------------------------------------------------
Training Epoch 12
------------------------------------------------------------
```

```
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 12 Training Loss: 1.3046
Epoch 12 Training Accuracy: 68.45%

Evaluating on Validation set…
Validation Loss     : 0.6735
Validation Accuracy : 93.94%
  Validation accuracy improved: 93.90% → 93.94%

------------------------------------------------------------
Training Epoch 13
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 13 Training Loss: 1.2779
Epoch 13 Training Accuracy: 69.01%

Evaluating on Validation set…
Validation Loss     : 0.6818
Validation Accuracy : 94.24%
  Validation accuracy improved: 93.94% → 94.24%

------------------------------------------------------------
Training Epoch 14
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 14 Training Loss: 1.2904
Epoch 14 Training Accuracy: 68.98%

Evaluating on Validation set…
Validation Loss     : 0.6720
Validation Accuracy : 94.28%
  Validation accuracy improved: 94.24% → 94.28%

------------------------------------------------------------
Training Epoch 15
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
```

```
Expected image shape: (batch_size, 3, 32, 32)
Epoch 15 Training Loss: 1.2490
Epoch 15 Training Accuracy: 70.84%

Evaluating on Validation set…
Validation Loss     : 0.6782
Validation Accuracy : 93.82%
  No improvement. Counter: 1/8

------------------------------------------------------------
Training Epoch 16
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 16 Training Loss: 1.2618
Epoch 16 Training Accuracy: 69.67%

Evaluating on Validation set…
Validation Loss     : 0.6760
Validation Accuracy : 94.50%
  Validation accuracy improved: 94.28% → 94.50%

------------------------------------------------------------
Training Epoch 17
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 17 Training Loss: 1.2660
Epoch 17 Training Accuracy: 69.18%

Evaluating on Validation set…
Validation Loss     : 0.6594
Validation Accuracy : 94.60%
  Validation accuracy improved: 94.50% → 94.60%

------------------------------------------------------------
Training Epoch 18
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 18 Training Loss: 1.2778
Epoch 18 Training Accuracy: 68.98%
```

```
Evaluating on Validation set…
Validation Loss     : 0.6579
Validation Accuracy : 94.34%
  No improvement. Counter: 1/8

------------------------------------------------------------
Training Epoch 19
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 19 Training Loss: 1.2799
Epoch 19 Training Accuracy: 68.78%

Evaluating on Validation set…
Validation Loss     : 0.6563
Validation Accuracy : 94.66%
  Validation accuracy improved: 94.60% → 94.66%

------------------------------------------------------------
Training Epoch 20
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 20 Training Loss: 1.2494
Epoch 20 Training Accuracy: 70.68%

Evaluating on Validation set…
Validation Loss     : 0.6672
Validation Accuracy : 94.60%
  No improvement. Counter: 1/8

------------------------------------------------------------
Training Epoch 21
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 21 Training Loss: 1.2701
Epoch 21 Training Accuracy: 68.84%

Evaluating on Validation set…
Validation Loss     : 0.6715
```

```
Validation Accuracy : 94.46%
  No improvement. Counter: 2/8

------------------------------------------------------------
Training Epoch 22
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 22 Training Loss: 1.2021
Epoch 22 Training Accuracy: 72.30%

Evaluating on Validation set…
Validation Loss     : 0.6534
Validation Accuracy : 94.96%
  Validation accuracy improved: 94.66% → 94.96%

------------------------------------------------------------
Training Epoch 23
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 23 Training Loss: 1.2439
Epoch 23 Training Accuracy: 69.64%

Evaluating on Validation set…
Validation Loss     : 0.6526
Validation Accuracy : 94.90%
  No improvement. Counter: 1/8

------------------------------------------------------------
Training Epoch 24
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 24 Training Loss: 1.2676
Epoch 24 Training Accuracy: 69.13%

Evaluating on Validation set…
Validation Loss     : 0.6689
Validation Accuracy : 94.58%
  No improvement. Counter: 2/8
```

```
------------------------------------------------------------
Training Epoch 25
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 25 Training Loss: 1.2223
Epoch 25 Training Accuracy: 70.59%

Evaluating on Validation set…
Validation Loss     : 0.6548
Validation Accuracy : 95.00%
  Validation accuracy improved: 94.96% → 95.00%

------------------------------------------------------------
Training Epoch 26
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 26 Training Loss: 1.1970
Epoch 26 Training Accuracy: 72.84%

Evaluating on Validation set…
Validation Loss     : 0.6481
Validation Accuracy : 95.06%
  Validation accuracy improved: 95.00% → 95.06%

------------------------------------------------------------
Training Epoch 27
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 27 Training Loss: 1.2253
Epoch 27 Training Accuracy: 70.27%

Evaluating on Validation set…
Validation Loss     : 0.6555
Validation Accuracy : 95.20%
  Validation accuracy improved: 95.06% → 95.20%

------------------------------------------------------------
Training Epoch 28
------------------------------------------------------------
```

```
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 28 Training Loss: 1.2380
Epoch 28 Training Accuracy: 70.17%

Evaluating on Validation set…
Validation Loss     : 0.6689
Validation Accuracy : 94.94%
  No improvement. Counter: 1/8

------------------------------------------------------------
Training Epoch 29
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 29 Training Loss: 1.1820
Epoch 29 Training Accuracy: 73.34%

Evaluating on Validation set…
Validation Loss     : 0.6633
Validation Accuracy : 94.90%
  No improvement. Counter: 2/8

------------------------------------------------------------
Training Epoch 30
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 30 Training Loss: 1.2242
Epoch 30 Training Accuracy: 71.18%

Evaluating on Validation set…
Validation Loss     : 0.6501
Validation Accuracy : 94.76%
  No improvement. Counter: 3/8

------------------------------------------------------------
Training Epoch 31
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
```

```
Expected image shape: (batch_size, 3, 32, 32)
Epoch 31 Training Loss: 1.2186
Epoch 31 Training Accuracy: 70.80%

Evaluating on Validation set…
Validation Loss     : 0.6347
Validation Accuracy : 95.18%
  No improvement. Counter: 4/8

------------------------------------------------------------
Training Epoch 32
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 32 Training Loss: 1.1568
Epoch 32 Training Accuracy: 74.19%

Evaluating on Validation set…
Validation Loss     : 0.6375
Validation Accuracy : 95.18%
  No improvement. Counter: 5/8

------------------------------------------------------------
Training Epoch 33
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 33 Training Loss: 1.2653
Epoch 33 Training Accuracy: 68.56%

Evaluating on Validation set…
Validation Loss     : 0.6671
Validation Accuracy : 95.14%
  No improvement. Counter: 6/8

------------------------------------------------------------
Training Epoch 34
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 34 Training Loss: 1.2057
Epoch 34 Training Accuracy: 71.67%
```

```
Evaluating on Validation set…
Validation Loss     : 0.6415
Validation Accuracy : 95.40%
  Validation accuracy improved: 95.20% → 95.40%

------------------------------------------------------------
Training Epoch 35
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 35 Training Loss: 1.2265
Epoch 35 Training Accuracy: 69.96%

Evaluating on Validation set…
Validation Loss     : 0.6470
Validation Accuracy : 95.32%
  No improvement. Counter: 1/8

------------------------------------------------------------
Training Epoch 36
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 36 Training Loss: 1.2039
Epoch 36 Training Accuracy: 71.49%

Evaluating on Validation set…
Validation Loss     : 0.6412
Validation Accuracy : 95.38%
  No improvement. Counter: 2/8

------------------------------------------------------------
Training Epoch 37
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 37 Training Loss: 1.2132
Epoch 37 Training Accuracy: 70.92%

Evaluating on Validation set…
Validation Loss     : 0.6600
```

```
Validation Accuracy : 95.26%
  No improvement. Counter: 3/8

------------------------------------------------------------
Training Epoch 38
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 38 Training Loss: 1.2183
Epoch 38 Training Accuracy: 69.74%

Evaluating on Validation set…
Validation Loss     : 0.6436
Validation Accuracy : 95.20%
  No improvement. Counter: 4/8

------------------------------------------------------------
Training Epoch 39
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 39 Training Loss: 1.1776
Epoch 39 Training Accuracy: 72.35%

Evaluating on Validation set…
Validation Loss     : 0.6363
Validation Accuracy : 95.44%
  Validation accuracy improved: 95.40% → 95.44%

------------------------------------------------------------
Training Epoch 40
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 40 Training Loss: 1.1505
Epoch 40 Training Accuracy: 74.62%

Evaluating on Validation set…
Validation Loss     : 0.6434
Validation Accuracy : 95.38%
  No improvement. Counter: 1/8
```

```
------------------------------------------------------------
Training Epoch 41
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 41 Training Loss: 1.2084
Epoch 41 Training Accuracy: 70.80%

Evaluating on Validation set…
Validation Loss     : 0.6491
Validation Accuracy : 95.04%
  No improvement. Counter: 2/8

------------------------------------------------------------
Training Epoch 42
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 42 Training Loss: 1.1796
Epoch 42 Training Accuracy: 72.40%

Evaluating on Validation set…
Validation Loss     : 0.6425
Validation Accuracy : 95.38%
  No improvement. Counter: 3/8

------------------------------------------------------------
Training Epoch 43
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 43 Training Loss: 1.1993
Epoch 43 Training Accuracy: 72.05%

Evaluating on Validation set…
Validation Loss     : 0.6356
Validation Accuracy : 95.18%
  No improvement. Counter: 4/8

------------------------------------------------------------
Training Epoch 44
------------------------------------------------------------
```

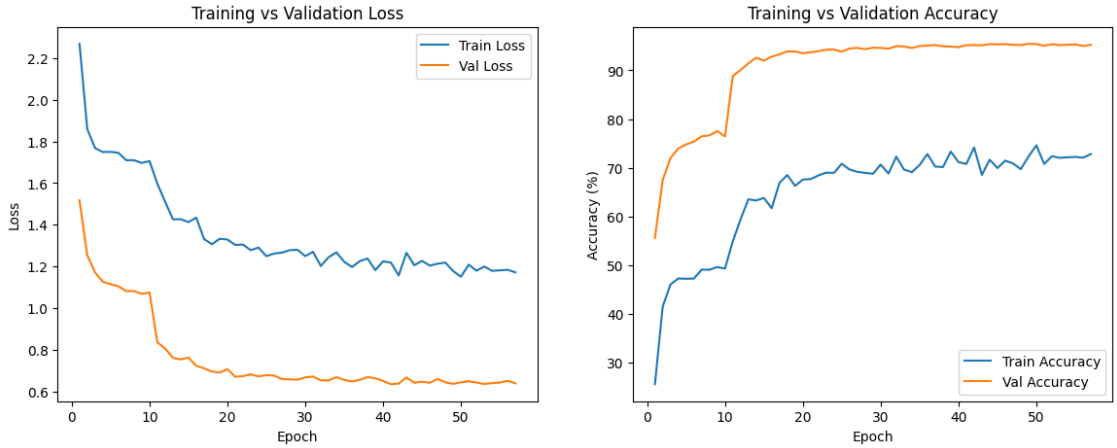
```
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 44 Training Loss: 1.1787
Epoch 44 Training Accuracy: 72.17%

Evaluating on Validation set…
Validation Loss     : 0.6400
Validation Accuracy : 95.26%
  No improvement. Counter: 5/8

------------------------------------------------------------
Training Epoch 45
------------------------------------------------------------
First batch augmentation used: CutMix
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 45 Training Loss: 1.1811
Epoch 45 Training Accuracy: 72.25%

Evaluating on Validation set…
Validation Loss     : 0.6425
Validation Accuracy : 95.32%
  No improvement. Counter: 6/8

------------------------------------------------------------
Training Epoch 46
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
Expected image shape: (batch_size, 3, 32, 32)
Epoch 46 Training Loss: 1.1839
Epoch 46 Training Accuracy: 72.10%

Evaluating on Validation set…
Validation Loss     : 0.6512
Validation Accuracy : 95.00%
  No improvement. Counter: 7/8

------------------------------------------------------------
Training Epoch 47
------------------------------------------------------------
First batch augmentation used: MixUp
• Batch image shape: torch.Size([128, 3, 32, 32])
• Batch label shape: torch.Size([128])
```

```
Expected image shape: (batch_size, 3, 32, 32)
Epoch 47 Training Loss: 1.1714
Epoch 47 Training Accuracy: 72.83%

Evaluating on Validation set…
Validation Loss     : 0.6387
Validation Accuracy : 95.24%
  No improvement. Counter: 8/8
Early stopping triggered

Training stopped early at epoch 47
```

27 STEP 22: Plot Training Curves (Loss & Accuracy)

[41]:
```python
print("\n" + "="*80)
print("STEP 22: PLOTTING TRAINING CURVES")
print("="*80)

epochs = range(1, len(history["train_loss"]) + 1)

plt.figure(figsize=(14, 5))

# Loss
plt.subplot(1, 2, 1)
plt.plot(epochs, history["train_loss"], label="Train Loss")
plt.plot(epochs, history["val_loss"], label="Val Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training vs Validation Loss")
plt.legend()

# Accuracy
plt.subplot(1, 2, 2)
plt.plot(epochs, history["train_acc"], label="Train Accuracy")
plt.plot(epochs, history["val_acc"], label="Val Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.title("Training vs Validation Accuracy")
plt.legend()

plt.show()

print("\nExpected behavior:")
print("• Training loss ↓ steadily")
print("• Validation loss ↓ then stabilizes")
print("• Early stopping prevents divergence")
```

```
================================================================================
STEP 22: PLOTTING TRAINING CURVES
================================================================================
```



Expected behavior:

- Training loss ↓ steadily
- Validation loss ↓ then stabilizes
- Early stopping prevents divergence

28 STEP 23: Visualize Sample Predictions (Subset Test)

[42]:
```
print("\n" + "="*80)
print("STEP 23: VISUALIZING SAMPLE PREDICTIONS")
print("="*80)

model.eval()
images, labels = next(iter(subset_test_loader))
images, labels = images.to(DEVICE), labels.to(DEVICE)

with torch.no_grad():
    outputs = model(images)
    _, preds = outputs.max(1)

plt.figure(figsize=(15, 4))
for i in range(8):
    img = images[i].cpu()
    img = img * torch.tensor(CIFAR_STD).view(3,1,1) + torch.tensor(CIFAR_MEAN).
 ↪view(3,1,1)
    img = img.clamp(0,1)
```

```
    plt.subplot(1, 8, i + 1)
    plt.imshow(img.permute(1,2,0))
    plt.title(f"P: {CLASSES[preds[i]]}\nT: {CLASSES[labels[i]]}")
    plt.axis("off")

plt.show()

print("\nInterpretation:")
print("• Correct predictions → confidence in learning")
print("• Wrong predictions → visually ambiguous cases")
```

```
================================================================================
STEP 23: VISUALIZING SAMPLE PREDICTIONS
================================================================================
```


```
Interpretation:
• Correct predictions → confidence in learning
• Wrong predictions → visually ambiguous cases
```

29 STEP 24: Calculate Per-Class Accuracy

[43]:
```
print("\n" + "="*80)
print("STEP 24: PER-CLASS ACCURACY CALCULATION")
print("="*80)

def per_class_accuracy(labels, preds, num_classes):
    correct = np.zeros(num_classes)
    total = np.zeros(num_classes)

    for l, p in zip(labels, preds):
        total[l] += 1
        if l == p:
            correct[l] += 1

    return 100 * correct / total
```

```
================================================================================
STEP 24: PER-CLASS ACCURACY CALCULATION
================================================================================
```

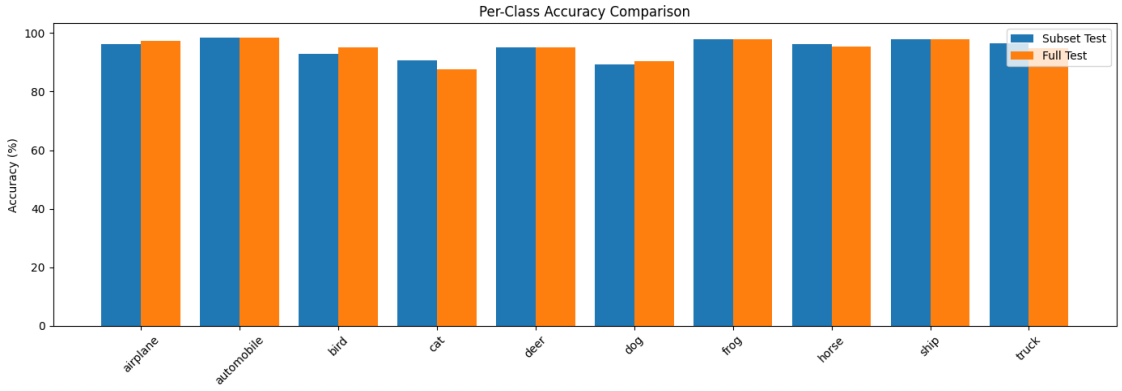
[44]:
```
_, _, subset_labels, subset_preds = evaluate(
    model, subset_test_loader, criterion, DEVICE, split_name="Subset Test"
)

_, _, full_labels, full_preds = evaluate(
    model, full_test_loader, criterion, DEVICE, split_name="Full Test"
)

subset_acc = per_class_accuracy(subset_labels, subset_preds, NUM_CLASSES)
full_acc = per_class_accuracy(full_labels, full_preds, NUM_CLASSES)
```

```
Evaluating on Subset Test set…
Subset Test Loss     : 0.6529
Subset Test Accuracy : 95.10%

Evaluating on Full Test set…
Full Test Loss     : 0.6469
Full Test Accuracy : 94.98%
```

[45]:
```
print("\n" + "="*80)
print("STEP 25: PER-CLASS ACCURACY VISUALIZATION")
print("="*80)

x = np.arange(NUM_CLASSES)

plt.figure(figsize=(14, 5))
plt.bar(x - 0.2, subset_acc, width=0.4, label="Subset Test")
plt.bar(x + 0.2, full_acc, width=0.4, label="Full Test")

plt.xticks(x, CLASSES, rotation=45)
plt.ylabel("Accuracy (%)")
plt.title("Per-Class Accuracy Comparison")
plt.legend()
plt.tight_layout()
plt.show()

print("\nInterpretation:")
print("• Subset test is usually slightly higher")
print("• Full test reflects real generalization")
```

```
================================================================================
STEP 25: PER-CLASS ACCURACY VISUALIZATION
```

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

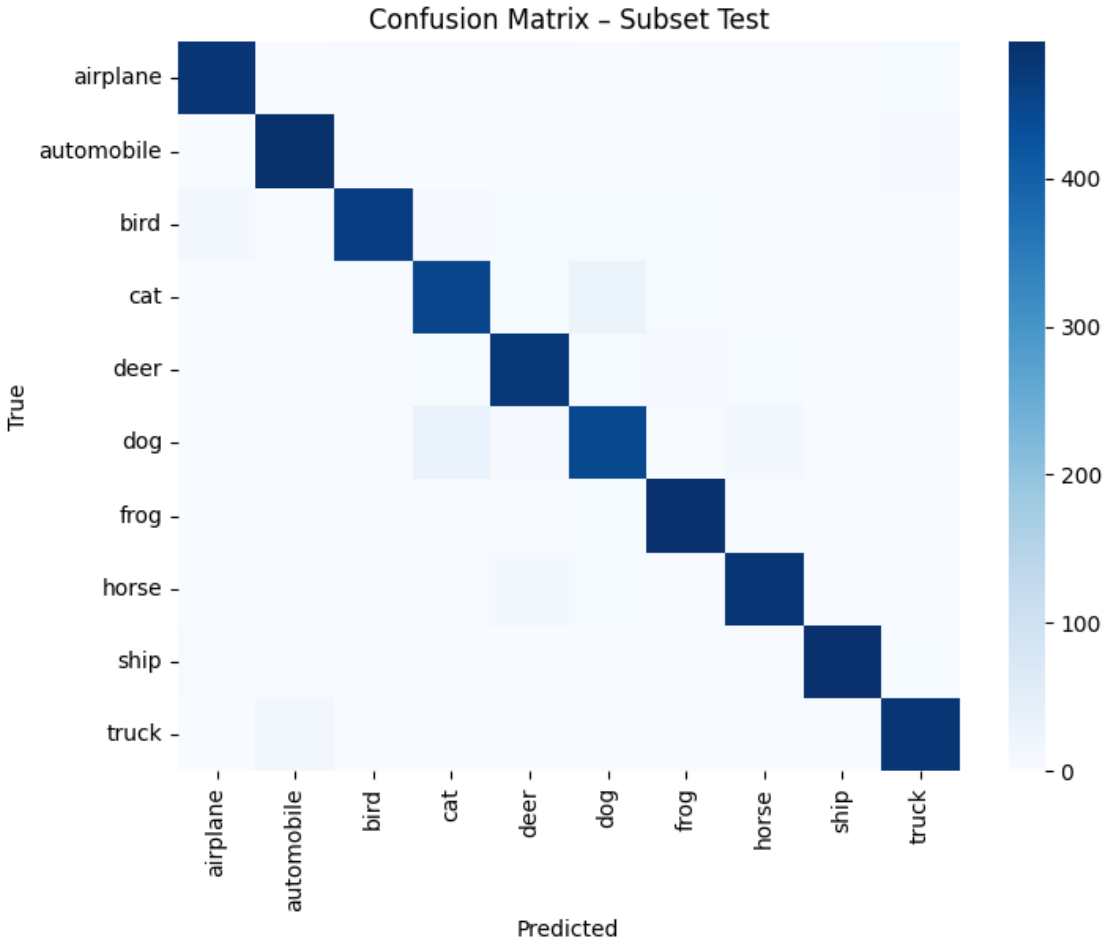


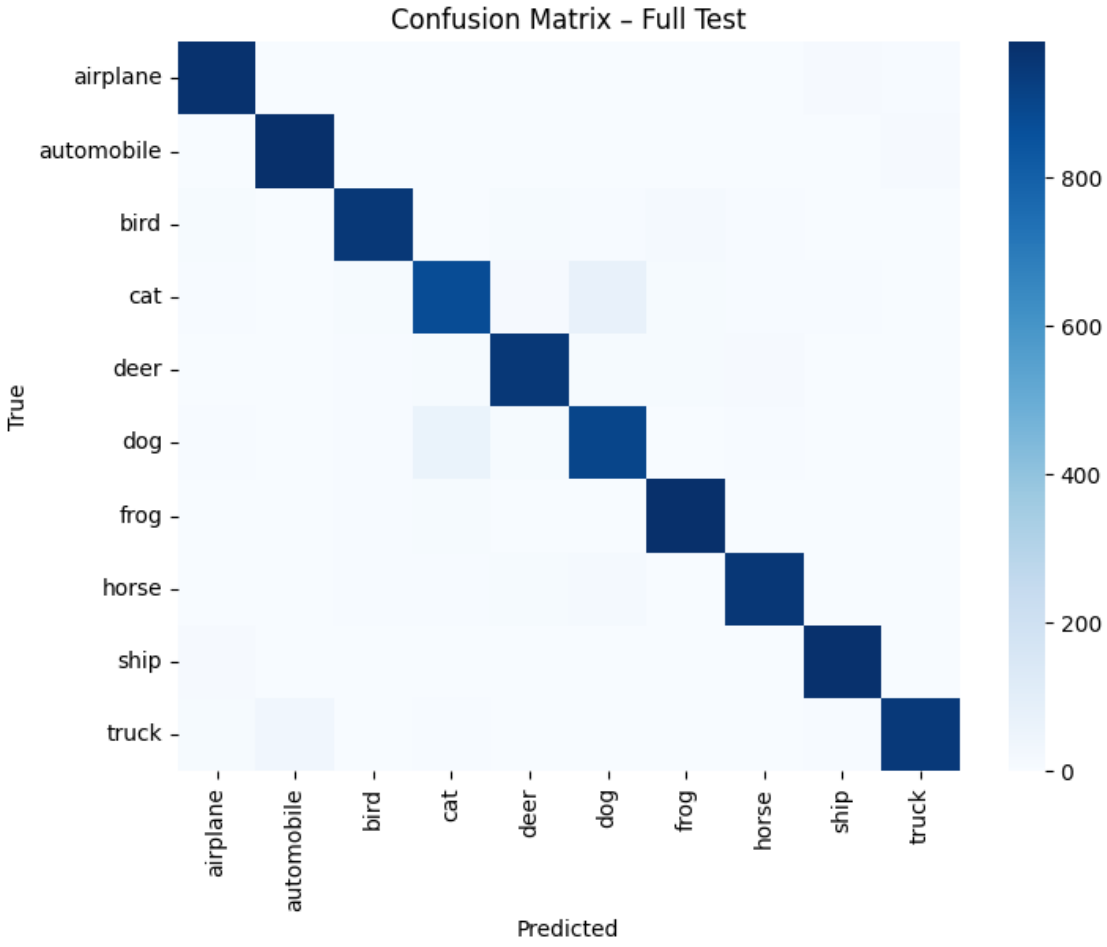
Interpretation:

- Subset test is usually slightly higher
- Full test reflects real generalization

[46]:
```
print("\n" + "="*80)
print("STEP 26: CONFUSION MATRICES")
print("="*80)

def plot_confusion_matrix(labels, preds, title):
    cm = confusion_matrix(labels, preds)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=False, cmap="Blues",
                xticklabels=CLASSES, yticklabels=CLASSES)
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title(title)
    plt.show()

plot_confusion_matrix(subset_labels, subset_preds, "Confusion Matrix - Subset␣
 ↪Test")
plot_confusion_matrix(full_labels, full_preds, "Confusion Matrix - Full Test")
```

```
================================================================================
STEP 26: CONFUSION MATRICES
================================================================================
```


Confusion Matrix – Subset Test
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
True
Predicted
400
300
200
100
0



[47]:
```
print("\n" + "="*80)
print("STEP 27: FINAL SUMMARY")
print("="*80)

print("MODEL               : ViT-Small (Patch 8), ImageNet pretrained")
print("DATASET             : CIFAR-10 (class-balanced subset)")
print("TRAIN SAMPLES       : 20,000")
print("VALIDATION SAMPLES  : 5,000")
print("TEST (SUBSET)       : 5,000")
print("TEST (FULL)         : 10,000")
print("\nKEY TECHNIQUES USED:")
print("• Transfer learning (ImageNet → CIFAR-10)")
print("• Strong augmentation (MixUp + CutMix)")
print("• Two-stage fine-tuning")
print("• Early stopping")
print("• Cosine learning rate schedule")
```

```
print("\nWHAT STUDENTS SHOULD REMEMBER:")
print("• ViTs need pretraining")
print("• Patch size matters")
print("• Validation controls training")
print("• Attention replaces convolution")

print("\nEXPERIMENT COMPLETE")
```

```
================================================================================
STEP 27: FINAL SUMMARY
================================================================================
MODEL              : ViT-Small (Patch 8), ImageNet pretrained
DATASET            : CIFAR-10 (class-balanced subset)
TRAIN SAMPLES      : 20,000
VALIDATION SAMPLES : 5,000
TEST (SUBSET)      : 5,000
TEST (FULL)        : 10,000

KEY TECHNIQUES USED:
• Transfer learning (ImageNet → CIFAR-10)
• Strong augmentation (MixUp + CutMix)
• Two-stage fine-tuning
• Early stopping
• Cosine learning rate schedule

WHAT STUDENTS SHOULD REMEMBER:
• ViTs need pretraining
• Patch size matters
• Validation controls training
• Attention replaces convolution

EXPERIMENT COMPLETE
```