Experiment-4

Convolutional Neural Networks (CNN)

1. Aim

To study the theoretical foundations of Convolutional Neural Networks (CNNs) and their role in image classification by introducing convolution, feature maps, and pooling operations for colour images, and to implement a small CNN on a CIFAR-10 subset (limited classes and epochs), visualising learned filters, intermediate feature maps, and class-wise performance results.

2. Theory

I. Motivation for Convolutional Neural Networks

Fully connected networks aren't ideal for images because flattening destroys spatial relationships between nearby pixels, and the dense connections create too many parameters, making them costly and prone to overfitting. CNNs solve this by using local connectivity and shared weights, which preserve spatial patterns and make learning more efficient and effective for visual data.

II. Digital Images as Inputs to CNNs

A digital image can be represented as a 3D array (Tensor) with Height, Width, and Channels. For color images, the three channels correspond to RGB intensity values. For example, CIFAR-10 images are 32 × 32 pixels with three channels, giving an input tensor of shape 32 × 32 × 3. CNNs take this multi-dimensional input directly, preserving spatial and channel-wise information during processing.

III. Data Normalization

Raw pixel values range from 0 to 255. Before training, we scale these values using a specific Mean and Standard Deviation (e.g., Mean: 0.4914, Std: 0.2470 for CIFAR-10).It scales input features to a similar range, which ensures gradient stability and allows the model to converge (learn) faster.

IV. Neurons in the Convolutional Layer

Local Connectivity: Unlike standard neurons that try to look at an entire image at once, a CNN neuron focuses only on a small, specific window of the image called the receptive field. This "local connectivity" ensures the network can focus on small details rather than getting overwhelmed by the whole picture.

Calculate Weighted Sum & Bias: Inside this small window, the neuron performs a simple mathematical operation. It takes the pixel values in that area and multiplies them by its own learned weights (stored in a filter or kernel). It adds these results together along with a bias to produce a single numerical value. This value represents how strongly a specific feature, like a vertical edge, is present in that spot.

Activation & Feature Creation: Finally, this value is passed through an activation function (like ReLU) to introduce non-linearity, allowing the network to learn more complex patterns. The resulting number becomes one single "pixel" or element in a new grid called a feature map.

Below is the equation to calculate the output of a single neuron within a convolutional layer.

$$Y = f\left(\sum_{i=1}^{m}\sum_{j=1}^{n} w_{ij}\, x_{ij} + b\right)$$

Where: i , j = 1,2,3,4,5….

- x_{ij} = input pixel in receptive field
- w_{ij} = filter weight
- b = bias
- f(•) = activation function (ReLU , sigmoid ,etc.)

V. Activation Functions in CNNs

ReLU (Rectified Linear Unit): Applied after convolution, defined as f(x)=max(0,x) as shown in Fig 1.

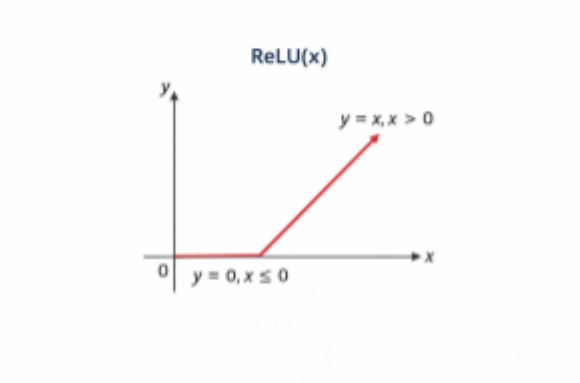


Fig 1: Relu Function

Refer Figure 2 shown below to understand how the Relu function affects the filter output

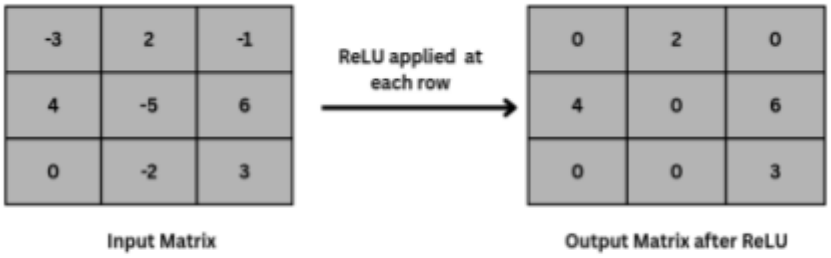


Fig. 2: ReLU activation function applied on input matrix

VI. Convolution Operation

Convolution slides a small filter over the image and computes weighted sums to capture local features, such as edges and textures. Reusing the same filter across the image reduces parameters while keeping strong feature learning.

$$(I * K)(x, y) = \sum_{i} \sum_{j} I(x + i, y + j)\, K(i, j)$$

Mathematically, the convolution operation can be expressed as the above equation where I is the input image and K is the kernel.

Refer fig 3. To understand how convolution operation is actually performed by the kernel on an input image matrix and a feature map is calculated.

Learnable Parameters: The "weights" in the filter are not hand-designed; the network learns the best values to detect features like edges automatically.

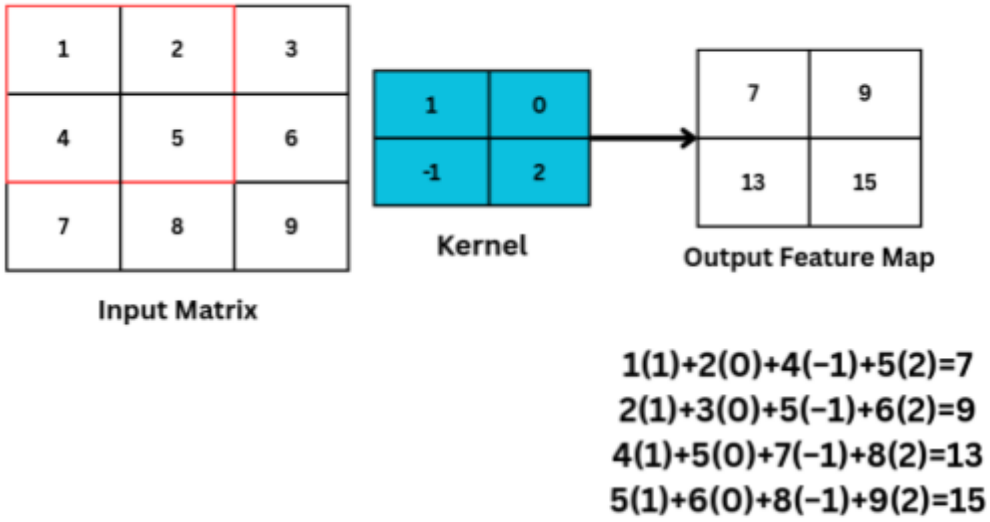


Fig 3: showing how kernel slides and calculate the feature map in convolution operation

VII. Feature Maps and Hierarchical Feature Learning

After convolution, the output is a feature map. Each filter detects a specific feature, and combining multiple filters produces multiple feature maps. As layers go deeper, simple features (like edges) combine into complex ones (shapes and parts), letting CNNs learn useful patterns automatically without manual feature design.

VIII. Stride and Padding

Stride is how many pixels the filter shifts each step; a larger stride reduces the output size. Padding adds (usually zero) pixels around the input to control output dimensions and keep edge information.Below is the equation for output Feature Map if stride and padding is introduced on input image where N = Input size, F = Filter size, P = Padding, and S = Stride.

$$\text{Output} = \frac{N - F + 2P}{S} + 1$$

Refer fig 4 to understand how stride operation is performed on an input matrix with a kernel size of 3 × 3 to get output as 2 × 2.

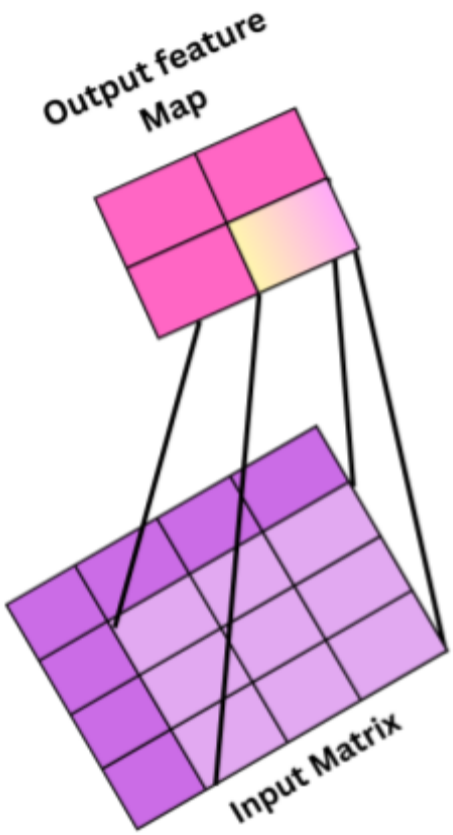


Fig 4: showing the output feature map when 3 × 3 kernel with stride 2 is applied on 4 × 4 image to produce 2 × 2 output.

$$\text{Output Feature Map} = \frac{5 - 3 + 2 \cdot 0}{2} + 1 = 2$$

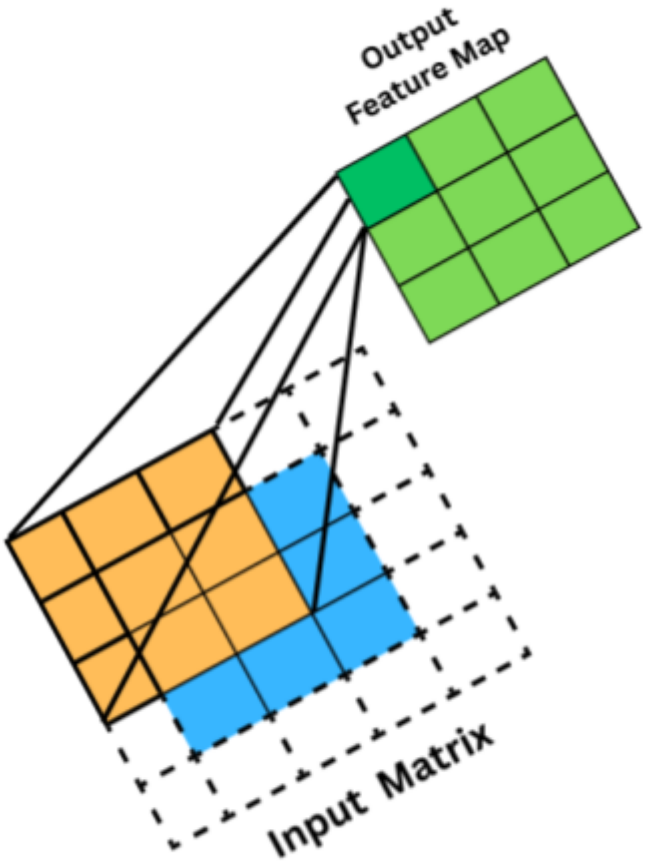


Fig 5: This shows 3 × 3 filter with padding of 1
on 3 × 3 matrix

Refer to Fig.5 to visualise how our output feature map will look like if padding is 1 so,

$$\text{Output Feature Map} = \frac{3 - 3 + 2 \cdot 1}{1} + 1 = 3$$

IX. Downsampling: Pooling Layers

Pooling downsamples feature maps to reduce computation and make features more robust to small shifts. Common types are max pooling, average pooling, and global average pooling, and they can also help reduce overfitting.

i. Max Pooling

Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map as shown in Fig 6.

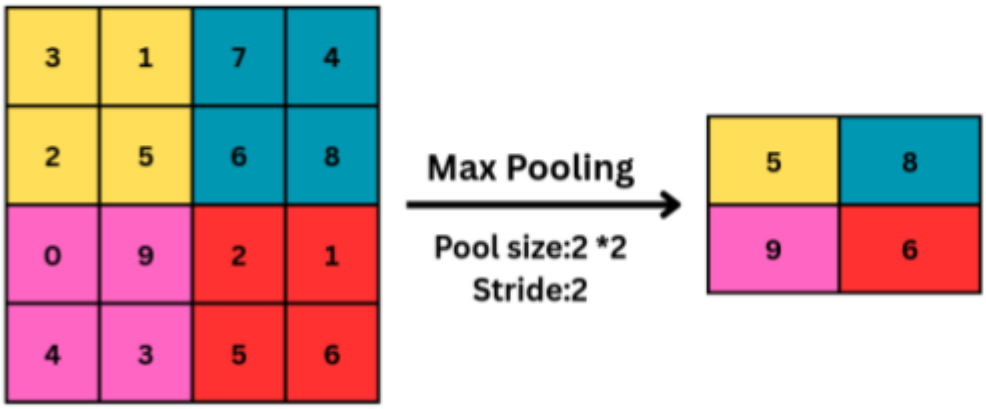


Fig 6: Max pooling of is applied on a feature map

ii. Average Pooling

Average pooling computes the average of the elements present in the region of the feature map covered by the filter.Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch as shown in Fig 7.

Fig 7: showing average pooling operation on matrix

iii. Global Average Pooling (GAP)

GAP is a downsampling operation typically used at the end of a CNN's feature extraction pipeline to prepare data for the final classification.

Unlike standard pooling that operates on small local regions, GAP summarizes each entire feature map into a single numerical value by calculating the average of all pixels in that map.

Its key roles in the architecture include:

- Spatial Aggregation: It aggregates all spatial information across a feature map, making the network's final decision more robust to the specific location of a feature.
- Vector Conversion: It converts 3D feature maps into a single 1D feature vector, which is then used by the classifier to produce the final prediction.
- Parameter Efficiency: By replacing traditional flattening and large fully connected layers, GAP significantly reduces the number of trainable parameters.
- Overfitting Prevention: This reduction in parameters helps reduce computational complexity and prevents overfitting, allowing the model to generalize better to new data.

X. Flattening

Before entering the fully connected layer, the featuremaps from the previous convolutional and pooling layers are typically flattened into a one-dimensional vector as shown in Fig 8.This is done to convert the spatial information into a format suitable for fully connected layers.

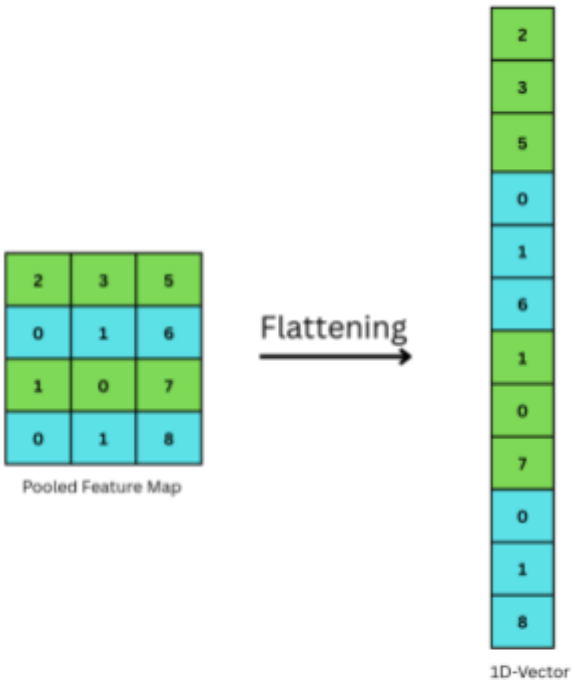


Fig 8: showing how flattening converts matrix in 1D vector

XI. Advanced Training Techniques

Batch Normalization: Used after convolution to normalize activations, which stabilizes and accelerates training.

Data Augmentation: Techniques like Random Horizontal Flips and Random Crops create "virtual" training data, helping the model generalize to unseen images.

MixUp Regularization: A technique where two images and their labels are blended together (e.g., a "cat-dog" hybrid) to force the model to learn more robust decision boundaries.

XII. Overall CNN Architecture

A typical CNN stacks convolution, activation, and pooling layers to extract features, then uses fully connected layers or global pooling as a classifier as shown in Fig. 9. This pipeline enables strong image recognition performance with efficient use of parameters.

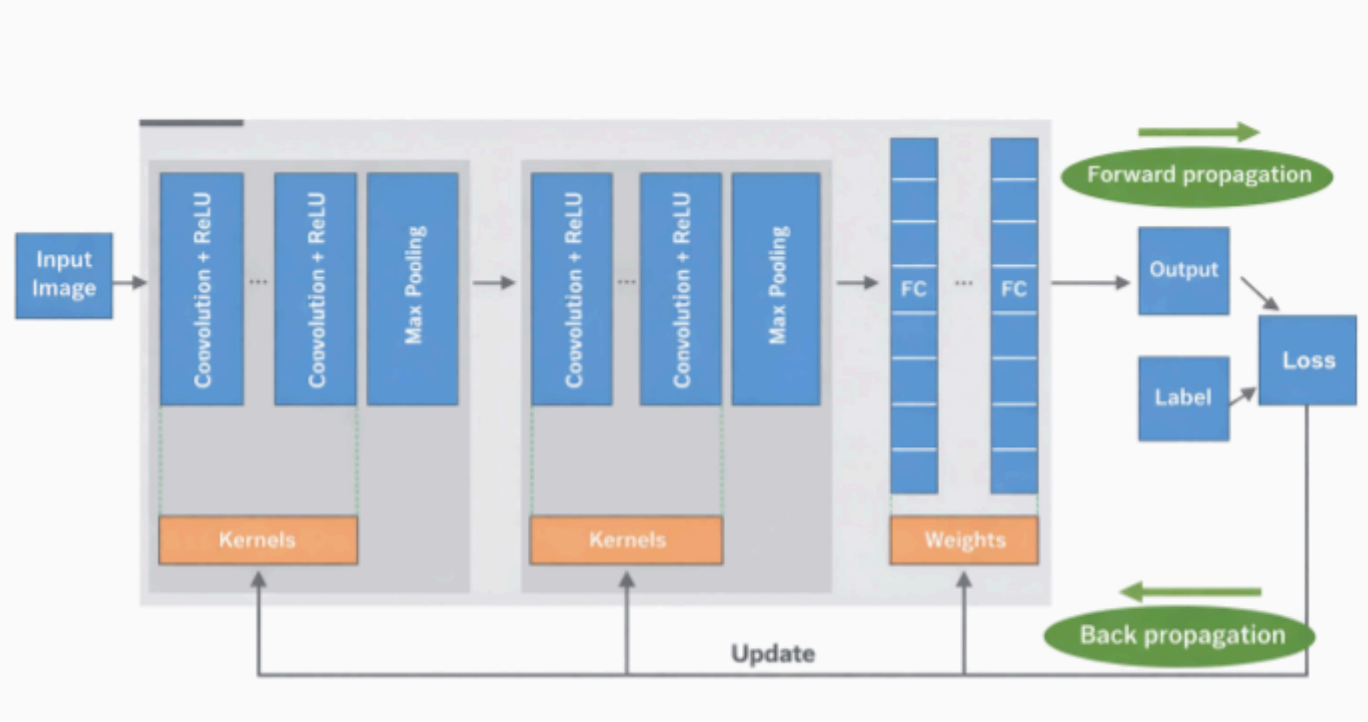


Fig 9: Complete CNN architecture
Source:(Yamashita *et al.*, *Convolutional neural networks: an overview and application*, 2018.

Merits of Convolutional Neural Networks:

- Efficient parameter sharing (fewer weights than fully connected networks)
- Preserves spatial structure in images
- Automatically learns and extracts features
- Strong generalization on image data
- Scalable to large and complex visual tasks
- Backbone of modern computer vision systems

Demerits of Convolutional Neural Networks:

- Require high computational resources (GPU/TPU)
- Need large labeled datasets for best performance
- Training deep CNNs can be time-consuming
- Features learned are hard to interpret (low explainability)

3. Code and Result

1 CIFAR-10 Image Classification with Convolutional Neural Networks

1.1 About the Dataset

CIFAR-10 (Canadian Institute For Advanced Research) is a benchmark dataset for image classification, consisting of: - **60,000 color images** of size 32x32 pixels - **10 mutually exclusive classes**: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck - **50,000 training images** and **10,000 test images** - **Balanced distribution**: 6,000 images per class

The small image size (32x32) and diverse object categories make this a challenging yet computationally tractable benchmark for evaluating convolutional neural network architectures.

1.2 The Classification Task

Given a 32x32 RGB image, the goal is to predict which of the 10 categories the image belongs to. This is a **multi-class classification** problem where the model outputs a probability distribution over all classes, and the class with the highest probability is selected as the prediction.

1.3 Notebook Structure

1. Environment setup and data exploration
2. Data preprocessing and augmentation
3. Model architecture design
4. Training configuration
5. Model training and evaluation
6. Results visualization and analysis

1.4 1. Import Libraries

WHY: We need tools for: - `torch` - Build and train neural networks (the brain) - `torchvision` - Load image datasets and apply transforms - `matplotlib` - Visualize data and results - `numpy` - Numerical operations on arrays

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
```

```
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
from pathlib import Path

# Success check
print(" All libraries imported successfully!")
print(f" PyTorch version: {torch.__version__}")
print(f" CUDA available: {torch.cuda.is_available()}")
```

```
All libraries imported successfully!
 PyTorch version: 2.8.0+cu129
 CUDA available: True
```

1.5 2. Set Device (CPU or GPU)

WHY: GPUs can train neural networks 10-100x faster than CPUs because they can perform many matrix operations in parallel.

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

print(f" Using device: {DEVICE}")
if DEVICE.type == 'cuda':
    print(f" GPU Name: {torch.cuda.get_device_name(0)}")
    print(f" GPU Memory: {torch.cuda.get_device_properties(0).total_memory /␣
 ↪1e9:.1f} GB")
```

```
Using device: cuda
 GPU Name: NVIDIA GeForce RTX 5090
 GPU Memory: 33.7 GB
```

1.6 3. Define Class Names

WHY: CIFAR-10 has 10 classes. Labels are stored as numbers (0-9), so we need a mapping to understand what each number means.

[3]:
```
CLASSES = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck')

print(" CIFAR-10 Classes:")
for i, name in enumerate(CLASSES):
    print(f" {i} → {name}")
```

```
CIFAR-10 Classes:
 0 → plane
```

```
 1 → car
 2 → bird
 3 → cat
 4 → deer
 5 → dog
 6 → frog
 7 → horse
 8 → ship
 9 → truck
```

1.7 4. Load Raw Dataset (For Exploration)

WHY: Before building a model, we must understand our data. What do the images look like? How are classes distributed? What are the pixel value ranges?

We load the data **without** transforms first to see the original images.

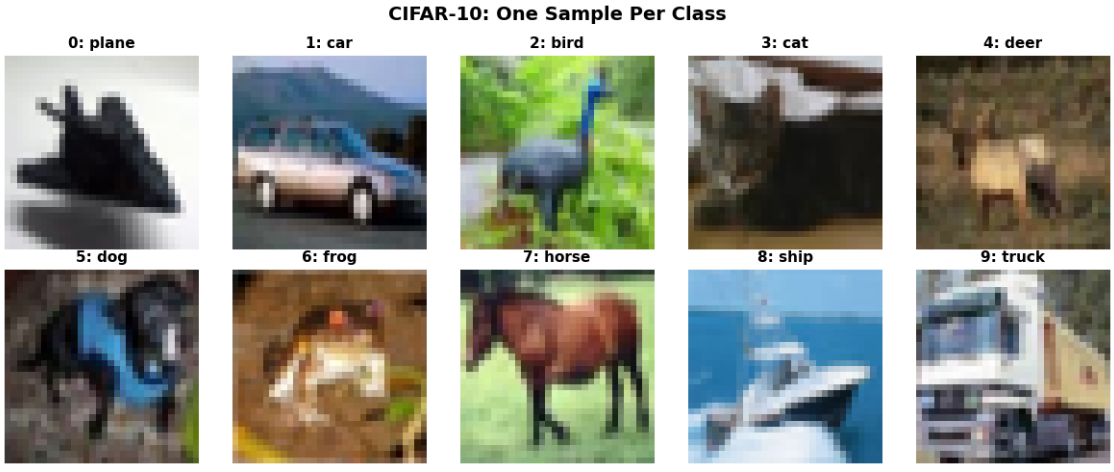
[4]:
```
# Load dataset without any transforms (raw images)
raw_trainset = torchvision.datasets.CIFAR10(root='./data', train=True,␣
 ↪download=True)
raw_testset = torchvision.datasets.CIFAR10(root='./data', train=False,␣
 ↪download=True)

print(" Dataset downloaded successfully!")
print(f"\n Dataset Statistics:")
print(f"  Training samples: {len(raw_trainset):,}")
print(f"  Test samples:     {len(raw_testset):,}")
print(f"  Total samples:    {len(raw_trainset) + len(raw_testset):,}")
print(f"\n Image Properties:")
sample_img, sample_label = raw_trainset[0]
print(f"  Image size: {sample_img.size} (Width x Height)")
print(f"  Channels: 3 (RGB color)")
print(f"  Pixel value range: 0-255")
```

```
 Dataset downloaded successfully!

Dataset Statistics:
 Training samples: 50,000
 Test samples:     10,000
 Total samples:    60,000

Image Properties:
 Image size: (32, 32) (Width x Height)
 Channels: 3 (RGB color)
 Pixel value range: 0-255
```

1.8 5. Visualize Sample Images (One Per Class)

WHY: Seeing examples helps us understand the task difficulty. Notice how small 32x32 images are - this is a challenging classification task!

[5]:
```
# Find one image for each class
class_samples = {}
for img, label in raw_trainset:
    if label not in class_samples:
        class_samples[label] = img
    if len(class_samples) == 10:
        break

# Plot one image per class
fig, axes = plt.subplots(2, 5, figsize=(12, 5))
for i, ax in enumerate(axes.flat):
    ax.imshow(class_samples[i])
    ax.set_title(f"{i}: {CLASSES[i]}", fontsize=11, fontweight='bold')
    ax.axis('off')

plt.suptitle('CIFAR-10: One Sample Per Class', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()

print(" Displayed one sample image for each of the 10 classes")
```

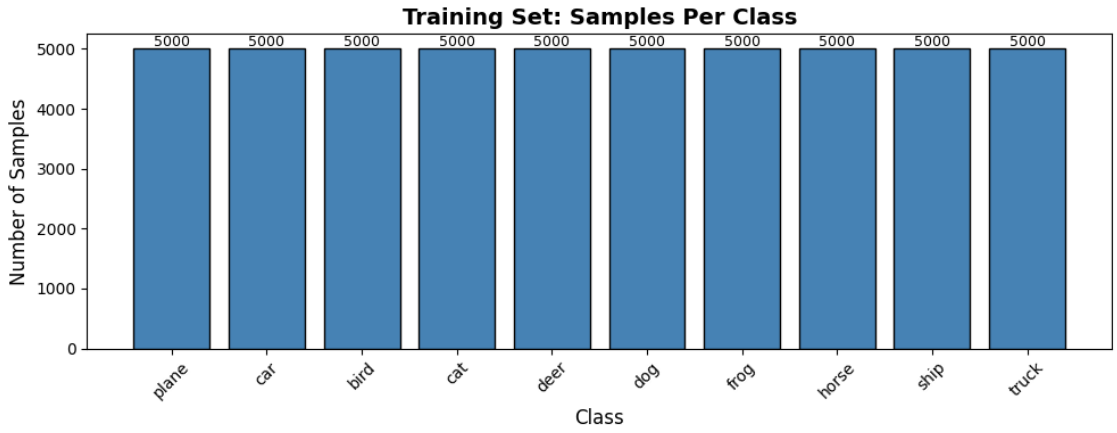



```
Displayed one sample image for each of the 10 classes
```

1.9 6. Check Class Distribution

WHY: Imbalanced datasets (where some classes have more samples) can bias the model. We verify CIFAR-10 is balanced.

[6]:
```
# Count samples per class
train_labels = [label for _, label in raw_trainset]
class_counts = [train_labels.count(i) for i in range(10)]

# Plot distribution
plt.figure(figsize=(10, 4))
bars = plt.bar(CLASSES, class_counts, color='steelblue', edgecolor='black')
plt.xlabel('Class', fontsize=12)
plt.ylabel('Number of Samples', fontsize=12)
plt.title('Training Set: Samples Per Class', fontsize=14, fontweight='bold')
plt.xticks(rotation=45)

# Add count labels on bars
for bar, count in zip(bars, class_counts):
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 50,
             str(count), ha='center', fontsize=9)

plt.tight_layout()
plt.show()

print(f" Dataset is balanced: {class_counts[0]} samples per class")
print(f"  Total: {sum(class_counts):,} training images")
```




```
Dataset is balanced: 5000 samples per class
 Total: 50,000 training images
```

1.10 7. Define Normalization Values

WHY: Neural networks train better when input values are small and centered around 0. We use pre-computed mean and std of CIFAR-10 to normalize pixel values from [0,255] to roughly [-1,1].

[7]:
```
# Pre-computed mean and std for CIFAR-10 (per channel: R, G, B)
CIFAR_MEAN = (0.4914, 0.4822, 0.4465)
CIFAR_STD = (0.2023, 0.1994, 0.2010)

print(" Normalization values set:")
print(f"  Mean (R,G,B): {CIFAR_MEAN}")
print(f"  Std  (R,G,B): {CIFAR_STD}")
print(f"\n After normalization:")
print(f"   Pixel 0   → {(0 - CIFAR_MEAN[0]) / CIFAR_STD[0]:.2f}")
print(f"   Pixel 255 → {(1 - CIFAR_MEAN[0]) / CIFAR_STD[0]:.2f}")
```

```
 Normalization values set:
  Mean (R,G,B): (0.4914, 0.4822, 0.4465)
  Std  (R,G,B): (0.2023, 0.1994, 0.201)

 After normalization:
   Pixel 0   → -2.43
   Pixel 255 → 2.51
```

1.11 8. Define Data Transforms (Augmentation)

WHY Data Augmentation? - We only have 50,000 training images - Augmentation creates "virtual" new images by applying random transformations - This helps the model generalize better and reduces overfitting

Transforms used: - `RandomHorizontalFlip` - Flip left right (a flipped cat is still a cat) - `RandomCrop` - Slight position changes (objects can appear anywhere) - `AutoAugment` - Learned augmentation policies (color, rotation, etc.)

[8]:
```
# Training transforms (WITH augmentation)
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),           # 50% chance to flip
    transforms.RandomCrop(32, padding=4),             # Random crop with padding
    transforms.AutoAugment(transforms.AutoAugmentPolicy.CIFAR10),  # Auto␣
 ↪augmentation
    transforms.ToTensor(),                            # Convert to tensor [0,1]
    transforms.Normalize(CIFAR_MEAN, CIFAR_STD)       # Normalize to ~[-1,1]
])

# Test transforms (NO augmentation - we want consistent evaluation)
test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(CIFAR_MEAN, CIFAR_STD)
```

```
])

print(" Data transforms defined!")
print("\n Training augmentations:")
print("  1. Random horizontal flip (50%)")
print("  2. Random crop (32x32 from 40x40 padded)")
print("  3. AutoAugment (learned color/geometric transforms)")
print("  4. Normalize to zero-centered values")
```

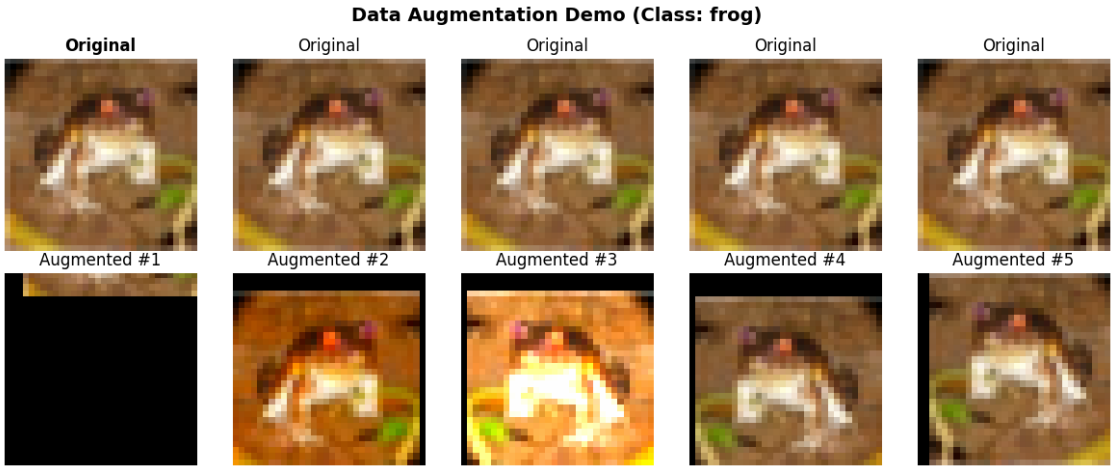
```
 Data transforms defined!

Training augmentations:
 1. Random horizontal flip (50%)
 2. Random crop (32x32 from 40x40 padded)
 3. AutoAugment (learned color/geometric transforms)
 4. Normalize to zero-centered values
```

1.12 9. Visualize Augmentation Effects

WHY: Let's see what augmented images look like. Each time we load the same image, it looks different!

[9]:
```
# Get one original image
original_img, label = raw_trainset[0]

# Apply augmentation multiple times to the same image
fig, axes = plt.subplots(2, 5, figsize=(12, 5))

# First row: original image repeated
axes[0, 0].imshow(original_img)
axes[0, 0].set_title('Original', fontweight='bold')
axes[0, 0].axis('off')
for i in range(1, 5):
    axes[0, i].imshow(original_img)
    axes[0, i].set_title('Original')
    axes[0, i].axis('off')

# Second row: augmented versions
for i in range(5):
    aug_tensor = train_transform(original_img)
    # Denormalize for display
    aug_img = aug_tensor.numpy().transpose(1, 2, 0)
    aug_img = aug_img * np.array(CIFAR_STD) + np.array(CIFAR_MEAN)
    aug_img = np.clip(aug_img, 0, 1)
    axes[1, i].imshow(aug_img)
    axes[1, i].set_title(f'Augmented #{i+1}')
```

```
    axes[1, i].axis('off')

plt.suptitle(f'Data Augmentation Demo (Class: {CLASSES[label]})', fontsize=14,␣
 ↪fontweight='bold')
plt.tight_layout()
plt.show()

print(" Each augmented version is different - this creates 'virtual' training␣
 ↪data!")
```




```
Each augmented version is different - this creates 'virtual' training data!
```

1.13 10. Create Data Loaders

WHY: DataLoaders batch our data, shuffle it, and load it efficiently in parallel. - **Batch size 128**: Process 128 images at once (balances speed and memory) - **Shuffle**: Randomize order each epoch so model doesn't memorize sequence

[10]:
```
BATCH_SIZE = 128

# Load datasets with transforms
trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=False, transform=train_transform
)
testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=False, transform=test_transform
)

# Create data loaders
```

```
trainloader = DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True,␣
 ↪num_workers=4, pin_memory=True)
testloader = DataLoader(testset, batch_size=BATCH_SIZE, shuffle=False,␣
 ↪num_workers=4, pin_memory=True)

print(" DataLoaders created!")
print(f"\n Batch configuration:")
print(f"  Batch size: {BATCH_SIZE}")
print(f"  Training batches per epoch: {len(trainloader)}")
print(f"  Test batches: {len(testloader)}")
print(f"\n Each epoch processes {len(trainloader)} batches × {BATCH_SIZE}␣
 ↪images = {len(trainset):,} images")
```

```
 DataLoaders created!

Batch configuration:
  Batch size: 128
  Training batches per epoch: 391
  Test batches: 79

Each epoch processes 391 batches × 128 images = 50,000 images
```

1.14 11. Build the CNN Model

WHY CNNs for images? - Convolutional layers detect local patterns (edges, textures, shapes) - Deeper layers combine simple patterns into complex features - Much fewer parameters than fully-connected networks

Architecture Overview:

`Input (3x32x32) → [Conv→BN→ReLU] × 6 blocks → Global Pool → FC → 10 classes`

Layer explanations: - `Conv2d` - Detect patterns using learnable filters - `BatchNorm2d` - Stabilize training by normalizing activations - `ReLU` - Add non-linearity (let the network learn complex functions) - `AvgPool2d` - Reduce spatial size, keeping important features - `AdaptiveAvgPool2d` - Reduce all spatial dimensions to 1x1 - `Dropout` - Randomly disable neurons to prevent overfitting

[11]:
```
# Build CNN layer by layer
model = nn.Sequential(
    # === Block 1: 3 → 64 channels ===
    nn.Conv2d(3, 64, kernel_size=5, padding=2, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(inplace=True),

    # === Block 2: 64 → 119 channels ===
    nn.Conv2d(64, 119, kernel_size=5, padding=2, bias=False),
    nn.BatchNorm2d(119),
    nn.ReLU(inplace=True),
```

```
    nn.Dropout2d(0.04),
    nn.AvgPool2d(2),  # 32x32 → 16x16

    # === Block 3: 119 → 221 channels ===
    nn.Conv2d(119, 221, kernel_size=3, padding=1, bias=False),
    nn.BatchNorm2d(221),
    nn.ReLU(inplace=True),
    nn.Dropout2d(0.05),

    # === Block 4: 221 → 256 channels ===
    nn.Conv2d(221, 256, kernel_size=5, padding=2, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(inplace=True),
    nn.Dropout2d(0.02),
    nn.AvgPool2d(2),  # 16x16 → 8x8

    # === Block 5: 256 → 256 channels ===
    nn.Conv2d(256, 256, kernel_size=3, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(inplace=True),
    nn.Dropout2d(0.09),

    # === Block 6: 256 → 256 channels ===
    nn.Conv2d(256, 256, kernel_size=5, padding=2, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(inplace=True),
    nn.Dropout2d(0.08),

    # === Global Average Pooling ===
    nn.AdaptiveAvgPool2d(1),  # 8x8 → 1x1
    nn.Flatten(),

    # === Classifier ===
    nn.Dropout(0.38),
    nn.Linear(256, 10)
)

# Move to GPU
model = model.to(DEVICE)

print(" CNN Model created!")
print(f"\n Architecture: 6 conv blocks + Global Average Pooling + FC")
```

```
 CNN Model created!

Architecture: 6 conv blocks + Global Average Pooling + FC
```

1.15 12. Count Model Parameters

WHY: More parameters = more capacity to learn, but also more risk of overfitting and slower training.

[12]:
```
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(" Model Parameters:")
print(f"  Total:     {total_params:,}")
print(f"  Trainable: {trainable_params:,}")
print(f"  Size:      ~{total_params * 4 / 1024 / 1024:.1f} MB (float32)")

print("\n Parameters per layer:")
for i, layer in enumerate(model):
    params = sum(p.numel() for p in layer.parameters())
    if params > 0:
        print(f"  [{i:2d}] {layer.__class__.__name__:20s}: {params:,}")
```

```
Model Parameters:
 Total:     4,079,429
 Trainable: 4,079,429
 Size:      ~15.6 MB (float32)

Parameters per layer:
 [ 0] Conv2d              : 4,800
 [ 1] BatchNorm2d         : 128
 [ 3] Conv2d              : 190,400
 [ 4] BatchNorm2d         : 238
 [ 8] Conv2d              : 236,691
 [ 9] BatchNorm2d         : 442
 [12] Conv2d              : 1,414,400
 [13] BatchNorm2d         : 512
 [17] Conv2d              : 589,824
 [18] BatchNorm2d         : 512
 [21] Conv2d              : 1,638,400
 [22] BatchNorm2d         : 512
 [28] Linear              : 2,570
```

1.16 13. Print Model Architecture

WHY: Verify the model structure before training. Check input/output dimensions are correct.

[13]:
```
print("  Full Model Architecture:\n")
print(model)

# Test forward pass with dummy input
```

```
dummy_input = torch.randn(1, 3, 32, 32).to(DEVICE)
dummy_output = model(dummy_input)
print(f"\n  Forward pass test: Input {list(dummy_input.shape)} → Output␣
  ↪{list(dummy_output.shape)}")
```

```
  Full Model Architecture:

Sequential(
  (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
bias=False)
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): Conv2d(64, 119, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
bias=False)
  (4): BatchNorm2d(119, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (5): ReLU(inplace=True)
  (6): Dropout2d(p=0.04, inplace=False)
  (7): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (8): Conv2d(119, 221, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
  (9): BatchNorm2d(221, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (10): ReLU(inplace=True)
  (11): Dropout2d(p=0.05, inplace=False)
  (12): Conv2d(221, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
bias=False)
  (13): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (14): ReLU(inplace=True)
  (15): Dropout2d(p=0.02, inplace=False)
  (16): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
  (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (19): ReLU(inplace=True)
  (20): Dropout2d(p=0.09, inplace=False)
  (21): Conv2d(256, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
bias=False)
  (22): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (23): ReLU(inplace=True)
  (24): Dropout2d(p=0.08, inplace=False)
  (25): AdaptiveAvgPool2d(output_size=1)
  (26): Flatten(start_dim=1, end_dim=-1)
  (27): Dropout(p=0.38, inplace=False)
```

```
  (28): Linear(in_features=256, out_features=10, bias=True)
)

 Forward pass test: Input [1, 3, 32, 32] → Output [1, 10]
```

1.17 14. Define Loss Function

WHY CrossEntropyLoss? - Standard loss for multi-class classification - Combines softmax (convert to probabilities) + negative log likelihood

WHY Label Smoothing (0.04)? - Instead of hard labels [0,0,1,0,...], use soft labels [0.004,0.004,0.96,0.004,...] - Prevents overconfidence and improves generalization

[14]:
```
LABEL_SMOOTHING = 0.04

criterion = nn.CrossEntropyLoss(label_smoothing=LABEL_SMOOTHING)

print(" Loss function: CrossEntropyLoss")
print(f"  Label smoothing: {LABEL_SMOOTHING}")
print(f"\n  Example: True label 'cat' (class 3)")
print(f"   Hard target:  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]")
print(f"   Soft target:  [0.004, 0.004, 0.004, 0.96, 0.004, ...]")
```

```
 Loss function: CrossEntropyLoss
  Label smoothing: 0.04

 Example: True label 'cat' (class 3)
   Hard target:  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
   Soft target:  [0.004, 0.004, 0.004, 0.96, 0.004, …]
```

1.18 15. Define Optimizer

WHY AdamW? - Adam = Adaptive learning rates per parameter (faster convergence) - W = Decoupled weight decay (better regularization)

WHY Weight Decay? - Penalizes large weights → simpler models → less overfitting

[15]:
```
LEARNING_RATE = 0.00182
WEIGHT_DECAY = 0.000035

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

print(" Optimizer: AdamW")
print(f"  Learning rate: {LEARNING_RATE}")
print(f"  Weight decay:  {WEIGHT_DECAY}")
```

```
Optimizer: AdamW
 Learning rate: 0.00182
 Weight decay:  3.5e-05
```

1.19 16. Define Learning Rate Scheduler

WHY OneCycleLR? - Starts with low LR → ramps up to max → slowly decreases - "Super-convergence": trains faster and often achieves better accuracy

Phases: 1. Warmup (26%): LR increases from base to max 2. Annealing (74%): LR decreases following cosine curve

[16]:
```
NUM_EPOCHS = 150
MAX_LR = LEARNING_RATE * 6.74  # Peak learning rate
PCT_START = 0.30  # Warmup takes 26% of training

scheduler = optim.lr_scheduler.OneCycleLR(
    optimizer,
    max_lr=MAX_LR,
    epochs=NUM_EPOCHS,
    steps_per_epoch=len(trainloader),
    pct_start=PCT_START,
    anneal_strategy='cos'
)

print(" LR Scheduler: OneCycleLR")
print(f"  Base LR:     {LEARNING_RATE:.6f}")
print(f"  Max LR:      {MAX_LR:.6f}")
print(f"  Warmup:      {PCT_START*100:.0f}% of training ({int(NUM_EPOCHS *␣
 ↪PCT_START)} epochs)")
print(f"  Total steps: {NUM_EPOCHS * len(trainloader):,}")
```

```
LR Scheduler: OneCycleLR
 Base LR:     0.001820
 Max LR:      0.012267
 Warmup:      30% of training (45 epochs)
 Total steps: 58,650
```

1.20 17. Define Training Function

WHY a function? We'll call this once per epoch. It does one thing: train on all batches.

[17]:
```
def mixup_data(x, y, alpha=0.4, use_cuda=True):
    """Returns mixed inputs, pairs of targets, and lambda."""
    if alpha > 0:
        lam = np.random.beta(alpha, alpha)
```

```
    else:
        lam = 1

    if use_cuda:
        index = torch.randperm(x.size(0)).cuda()
    else:
        index = torch.randperm(x.size(0))

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

def mixup_criterion(criterion, pred, y_a, y_b, lam):
    return lam * criterion(pred, y_a) + (1 - lam) * criterion(pred, y_b)

def train_one_epoch(model, loader, criterion, optimizer, scheduler,␣
 ↪mixup_alpha=0.4):
    """Train model for one epoch and return (loss, accuracy)."""
    model.train()
    total_loss, correct, total = 0, 0, 0

    for images, labels in loader:
        images, labels = images.to(DEVICE), labels.to(DEVICE)

        # MixUp
        images, targets_a, targets_b, lam = mixup_data(images, labels,␣
 ↪mixup_alpha, use_cuda=images.is_cuda)

        # Forward pass
        optimizer.zero_grad()
        outputs = model(images)
        loss = mixup_criterion(criterion, outputs, targets_a, targets_b, lam)

        # Backward pass
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        scheduler.step()

        # Track metrics
        total_loss += loss.item()
        _, predicted = outputs.max(1)
        total += labels.size(0)

        # Weighted accuracy for MixUp
        if mixup_alpha > 0:
```

```
            correct += (lam * predicted.eq(targets_a).sum().float() + (1 - lam)␣
 ↪* predicted.eq(targets_b).sum().float()).item()
        else:
            correct += predicted.eq(labels).sum().item()

    return total_loss / len(loader), 100.0 * correct / total

print("  Training function defined (with MixUp)")
```

```
  Training function defined (with MixUp)
```

1.21 18. Define Evaluation Function

WHY separate from training? - `torch.no_grad()` disables gradient computation (faster, less memory) - No dropout/augmentation during evaluation for consistent results

[18]:
```
def evaluate(model, loader, criterion):
    """Evaluate model and return (loss, accuracy)."""
    model.eval()
    total_loss, correct, total = 0, 0, 0

    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(DEVICE), labels.to(DEVICE)
            outputs = model(images)
            loss = criterion(outputs, labels)

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

    return total_loss / len(loader), 100.0 * correct / total

print("  Evaluation function defined")
```

```
  Evaluation function defined
```

1.22 19. Create Output Directory

WHY: Save models and results for later use and reproducibility.

[19]:
```
OUTPUT_DIR = Path("cifar10_results")
OUTPUT_DIR.mkdir(exist_ok=True)

print(f"  Output directory: {OUTPUT_DIR}/")
```

```
Output directory: cifar10_results/
```

1.23 20. Train the Model!

What happens each epoch: 1. Train on all 50,000 images (in batches of 128) 2. Evaluate on 10,000 test images 3. Save model if test accuracy improved 4. Print progress every 5 epochs

This will take some time (GPU recommended)

[20]:
```python
print("="*70)
print(f"STARTING TRAINING FOR {NUM_EPOCHS} EPOCHS")
print("="*70 + "\n")

# Track history
train_accs, test_accs = [], []
train_losses, test_losses = [], []
best_acc = 0
best_epoch = 0

# Training loop
for epoch in range(NUM_EPOCHS):
    train_loss, train_acc = train_one_epoch(model, trainloader, criterion,␣
 ↪optimizer, scheduler)
    test_loss, test_acc = evaluate(model, testloader, criterion)

    # Store metrics
    train_accs.append(train_acc)
    test_accs.append(test_acc)
    train_losses.append(train_loss)
    test_losses.append(test_loss)

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        best_epoch = epoch + 1
        torch.save(model.state_dict(), OUTPUT_DIR / 'best_model.pth')
        marker = "  NEW BEST!"
    else:
        marker = ""

    # Print progress
    if (epoch + 1) % 5 == 0 or epoch == 0:
        print(f"Epoch [{epoch+1:3d}/{NUM_EPOCHS}] | "
              f"Train: {train_acc:5.2f}% | "
              f"Test: {test_acc:5.2f}% | "
              f"Loss: {test_loss:.4f}{marker}")
```

```python
print(f"\n{'='*70}")
print(f"TRAINING COMPLETE!")
print(f"{'='*70}")
print(f"\n  Best Test Accuracy: {best_acc:.2f}% (achieved at epoch␣
  ↪{best_epoch})")
print(f"  Final Test Accuracy: {test_accs[-1]:.2f}%")
```

```
======================================================================
STARTING TRAINING FOR 150 EPOCHS
======================================================================

Epoch [  1/150] | Train: 27.49% | Test: 48.56% | Loss: 1.5192   NEW BEST!
Epoch [  5/150] | Train: 48.00% | Test: 66.93% | Loss: 1.1223   NEW BEST!
Epoch [ 10/150] | Train: 57.56% | Test: 77.31% | Loss: 0.8456
Epoch [ 15/150] | Train: 62.27% | Test: 84.31% | Loss: 0.7112   NEW BEST!
Epoch [ 20/150] | Train: 65.22% | Test: 85.53% | Loss: 0.6925
Epoch [ 25/150] | Train: 67.57% | Test: 86.66% | Loss: 0.6671
Epoch [ 30/150] | Train: 69.23% | Test: 87.06% | Loss: 0.6486
Epoch [ 35/150] | Train: 69.43% | Test: 89.43% | Loss: 0.5953   NEW BEST!
Epoch [ 40/150] | Train: 70.38% | Test: 89.38% | Loss: 0.5619
Epoch [ 45/150] | Train: 71.11% | Test: 90.54% | Loss: 0.6206   NEW BEST!
Epoch [ 50/150] | Train: 71.10% | Test: 88.76% | Loss: 0.6044
Epoch [ 55/150] | Train: 72.31% | Test: 91.37% | Loss: 0.5581
Epoch [ 60/150] | Train: 71.96% | Test: 92.04% | Loss: 0.5848
Epoch [ 65/150] | Train: 73.11% | Test: 91.85% | Loss: 0.5363
Epoch [ 70/150] | Train: 75.01% | Test: 92.33% | Loss: 0.5045
Epoch [ 75/150] | Train: 75.90% | Test: 92.77% | Loss: 0.4935
Epoch [ 80/150] | Train: 75.61% | Test: 92.37% | Loss: 0.4923
Epoch [ 85/150] | Train: 75.01% | Test: 92.75% | Loss: 0.5046
Epoch [ 90/150] | Train: 73.35% | Test: 93.07% | Loss: 0.5097
Epoch [ 95/150] | Train: 76.29% | Test: 93.45% | Loss: 0.4956   NEW BEST!
Epoch [100/150] | Train: 76.31% | Test: 93.57% | Loss: 0.5111
Epoch [105/150] | Train: 77.69% | Test: 93.76% | Loss: 0.4592   NEW BEST!
Epoch [110/150] | Train: 76.36% | Test: 93.55% | Loss: 0.5129
Epoch [115/150] | Train: 77.50% | Test: 93.90% | Loss: 0.4902
Epoch [120/150] | Train: 77.44% | Test: 94.16% | Loss: 0.4413
Epoch [125/150] | Train: 77.64% | Test: 94.16% | Loss: 0.5472
Epoch [130/150] | Train: 78.90% | Test: 94.23% | Loss: 0.4761
Epoch [135/150] | Train: 76.93% | Test: 94.39% | Loss: 0.4460
Epoch [140/150] | Train: 79.82% | Test: 94.30% | Loss: 0.4980
Epoch [145/150] | Train: 77.86% | Test: 94.35% | Loss: 0.4702
Epoch [150/150] | Train: 78.69% | Test: 94.39% | Loss: 0.4466

======================================================================
TRAINING COMPLETE!
======================================================================

  Best Test Accuracy: 94.48% (achieved at epoch 149)
```

```
Final Test Accuracy: 94.39%
```

1.24 21. Plot Training Curves

WHY: Visualize how the model learned over time. The gap between train and test accuracy indicates overfitting.

[21]:
```python
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Accuracy plot
axes[0].plot(train_accs, label='Train Accuracy', linewidth=2)
axes[0].plot(test_accs, label='Test Accuracy', linewidth=2)
axes[0].axhline(y=best_acc, color='r', linestyle='--', alpha=0.5, label=f'Best:␣
 ↪{best_acc:.2f}%')
axes[0].axvline(x=best_epoch-1, color='g', linestyle='--', alpha=0.3,␣
 ↪label=f'Best Epoch: {best_epoch}')
axes[0].set_xlabel('Epoch', fontsize=12)
axes[0].set_ylabel('Accuracy (%)', fontsize=12)
axes[0].set_title('Training Progress - Accuracy', fontsize=14,␣
 ↪fontweight='bold')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Loss plot
axes[1].plot(train_losses, label='Train Loss', linewidth=2)
axes[1].plot(test_losses, label='Test Loss', linewidth=2)
axes[1].set_xlabel('Epoch', fontsize=12)
axes[1].set_ylabel('Loss', fontsize=12)
axes[1].set_title('Training Progress - Loss', fontsize=14, fontweight='bold')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.suptitle(f'CIFAR-10 Training Results (Best: {best_acc:.2f}% at epoch␣
 ↪{best_epoch})',
             fontsize=16, fontweight='bold', y=1.02)
plt.tight_layout()
plt.savefig(OUTPUT_DIR / 'training_curves.png', dpi=150, bbox_inches='tight')
plt.show()

print(f"  Training curves saved to: {OUTPUT_DIR / 'training_curves.png'}")
```

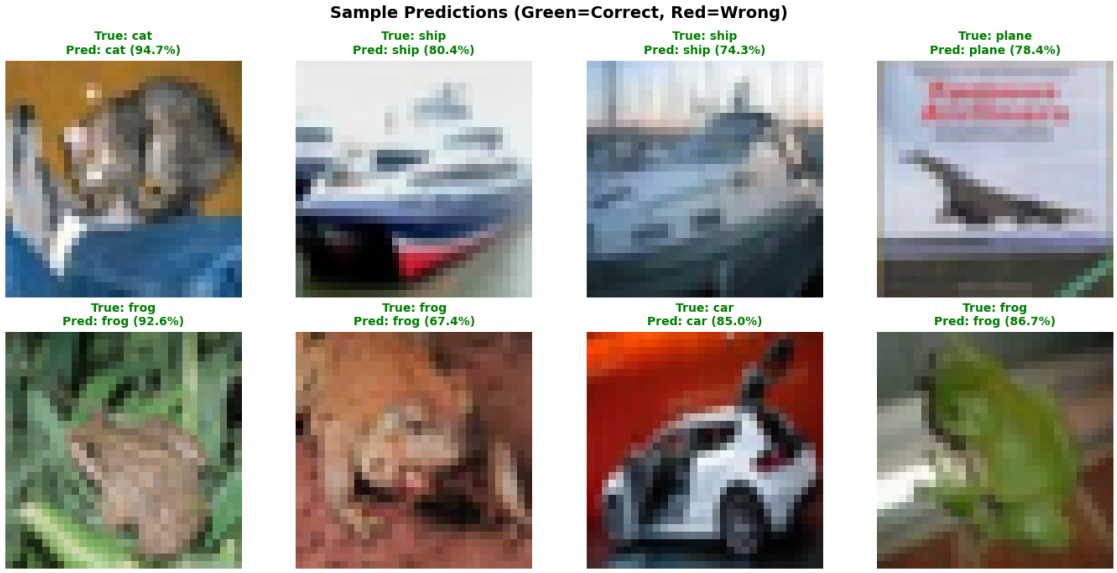

```
Training curves saved to: cifar10_results/training_curves.png
```

1.25 22. Visualize Sample Predictions

WHY: See what the model actually predicts. Green = correct, Red = wrong.

[22]:
```python
# Get a batch of test images
images, labels = next(iter(testloader))
images, labels = images.to(DEVICE), labels.to(DEVICE)

# Make predictions
model.eval()
with torch.no_grad():
    outputs = model(images)
    probs = torch.nn.functional.softmax(outputs, dim=1)
    _, predicted = outputs.max(1)

# Visualize
fig, axes = plt.subplots(2, 4, figsize=(14, 7))
for i, ax in enumerate(axes.flat):
    # Denormalize image
    img = images[i].cpu().numpy().transpose(1, 2, 0)
    img = img * np.array(CIFAR_STD) + np.array(CIFAR_MEAN)
    img = np.clip(img, 0, 1)

    ax.imshow(img)
    confidence = probs[i][predicted[i]].item() * 100
    color = 'green' if predicted[i] == labels[i] else 'red'
    ax.set_title(f"True: {CLASSES[labels[i]]}\nPred: {CLASSES[predicted[i]]}␣
 ↪({confidence:.1f}%)",
                 color=color, fontsize=10, fontweight='bold')
```

```
    ax.axis('off')

plt.suptitle('Sample Predictions (Green=Correct, Red=Wrong)', fontsize=14,␣
 ↪fontweight='bold')
plt.tight_layout()
plt.savefig(OUTPUT_DIR / 'sample_predictions.png', dpi=150, bbox_inches='tight')
plt.show()

print(f"  Sample predictions saved to: {OUTPUT_DIR / 'sample_predictions.png'}")
```




```
  Sample predictions saved to: cifar10_results/sample_predictions.png
```

1.26 23. Calculate Per-Class Accuracy

WHY: Some classes may be harder than others. Identify weaknesses.

[23]:
```
class_correct = [0] * 10
class_total = [0] * 10

model.eval()
with torch.no_grad():
    for images, labels in testloader:
        images, labels = images.to(DEVICE), labels.to(DEVICE)
        outputs = model(images)
        _, predicted = outputs.max(1)

        for i in range(len(labels)):
```

```
            label = labels[i].item()
            class_correct[label] += (predicted[i] == labels[i]).item()
            class_total[label] += 1

# Calculate accuracies
class_accs = [100 * class_correct[i] / class_total[i] for i in range(10)]

print("\nPer-Class Accuracy:")
print("-" * 40)
for i in range(10):
    bar = ' ' * int(class_accs[i] / 5)
    print(f"{CLASSES[i]:10s}: {class_accs[i]:5.2f}% {bar}")
print("-" * 40)
print(f"{'Overall':10s}: {best_acc:5.2f}%")
```

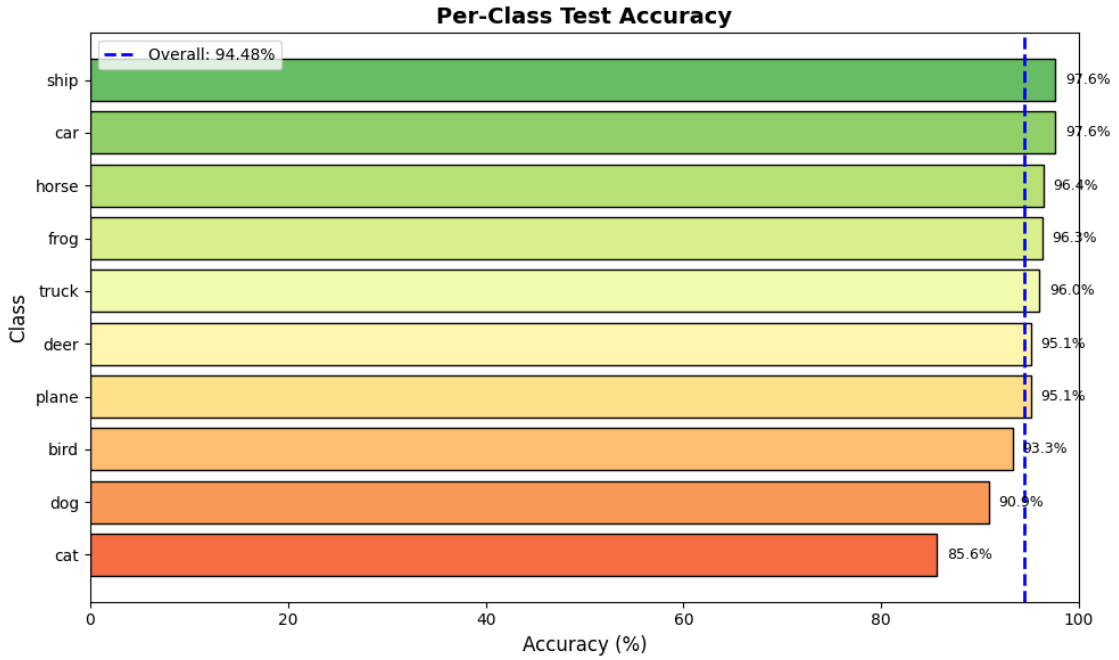
```
Per-Class Accuracy:
----------------------------------------
plane     : 95.10%
car       : 97.60%
bird      : 93.30%
cat       : 85.60%
deer      : 95.10%
dog       : 90.90%
frog      : 96.30%
horse     : 96.40%
ship      : 97.60%
truck     : 96.00%
----------------------------------------
Overall   : 94.48%
```

1.27 24. Visualize Per-Class Accuracy

WHY: A bar chart makes it easy to compare class performance at a glance.

[24]:
```
# Sort by accuracy for better visualization
sorted_idx = np.argsort(class_accs)
sorted_classes = [CLASSES[i] for i in sorted_idx]
sorted_accs = [class_accs[i] for i in sorted_idx]

# Color code: red for low, green for high
colors = plt.cm.RdYlGn(np.linspace(0.2, 0.8, 10))

plt.figure(figsize=(10, 6))
bars = plt.barh(sorted_classes, sorted_accs, color=colors, edgecolor='black')
```

```
plt.axvline(x=best_acc, color='blue', linestyle='--', linewidth=2,␣
 ↪label=f'Overall: {best_acc:.2f}%')
plt.xlabel('Accuracy (%)', fontsize=12)
plt.ylabel('Class', fontsize=12)
plt.title('Per-Class Test Accuracy', fontsize=14, fontweight='bold')
plt.xlim(0, 100)
plt.legend()

# Add accuracy labels
for bar, acc in zip(bars, sorted_accs):
    plt.text(acc + 1, bar.get_y() + bar.get_height()/2, f'{acc:.1f}%',␣
 ↪va='center', fontsize=9)

plt.tight_layout()
plt.savefig(OUTPUT_DIR / 'per_class_accuracy.png', dpi=150, bbox_inches='tight')
plt.show()

print(f" Per-class accuracy chart saved to: {OUTPUT_DIR / 'per_class_accuracy.
 ↪png'}")
```



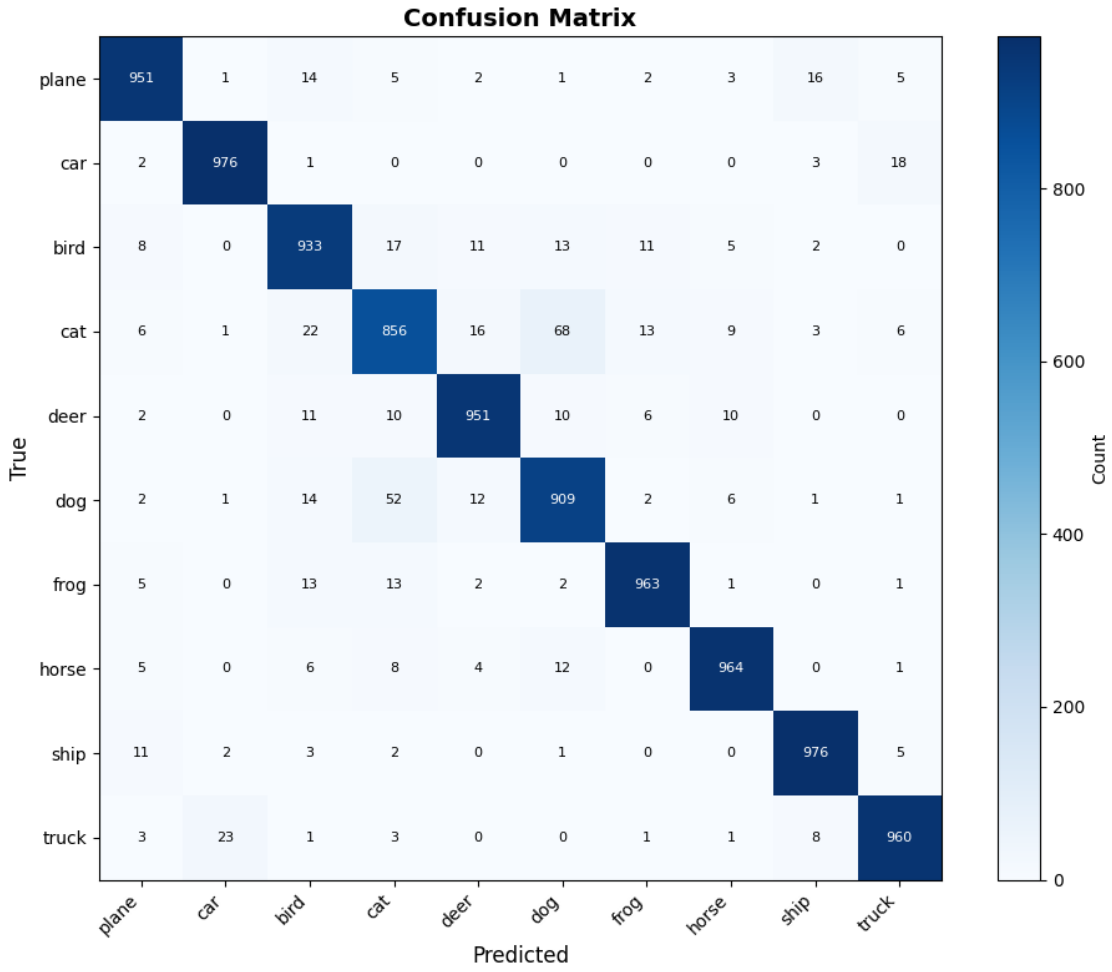

```
Per-class accuracy chart saved to: cifar10_results/per_class_accuracy.png
```

1.28 25. Create Confusion Matrix

WHY: Shows which classes get confused with each other (e.g., cat dog).

[25]:
```
# Collect all predictions
all_preds = []
all_labels = []

model.eval()
with torch.no_grad():
    for images, labels in testloader:
        images = images.to(DEVICE)
        outputs = model(images)
        _, predicted = outputs.max(1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.numpy())

# Create confusion matrix
confusion = np.zeros((10, 10), dtype=int)
for true, pred in zip(all_labels, all_preds):
    confusion[true][pred] += 1

# Plot
plt.figure(figsize=(10, 8))
plt.imshow(confusion, cmap='Blues')
plt.colorbar(label='Count')

# Add labels
plt.xticks(range(10), CLASSES, rotation=45, ha='right')
plt.yticks(range(10), CLASSES)
plt.xlabel('Predicted', fontsize=12)
plt.ylabel('True', fontsize=12)
plt.title('Confusion Matrix', fontsize=14, fontweight='bold')

# Add numbers in cells
for i in range(10):
    for j in range(10):
        color = 'white' if confusion[i, j] > confusion.max() / 2 else 'black'
        plt.text(j, i, str(confusion[i, j]), ha='center', va='center',␣
 ↪color=color, fontsize=8)

plt.tight_layout()
plt.savefig(OUTPUT_DIR / 'confusion_matrix.png', dpi=150, bbox_inches='tight')
plt.show()

print(f" Confusion matrix saved to: {OUTPUT_DIR / 'confusion_matrix.png'}")

# Find most confused pairs
```

```python
print("\n Most Common Confusions:")
for i in range(10):
    for j in range(10):
        if i != j and confusion[i, j] > 50:
            print(f"  {CLASSES[i]} → {CLASSES[j]}: {confusion[i, j]} times")
```




```
Confusion matrix saved to: cifar10_results/confusion_matrix.png

Most Common Confusions:
 cat → dog: 68 times
 dog → cat: 52 times
```

1.29 26. Final Summary

What we accomplished:

[26]:
```
print("="*70)
print("TRAINING SUMMARY")
print("="*70)

print(f"\nResults:")
print(f"   Best Accuracy:  {best_acc:.2f}% (Epoch {best_epoch})")
print(f"   Final Accuracy: {test_accs[-1]:.2f}%")
print(f"   Total Epochs:   {NUM_EPOCHS}")

print(f"\nModel:")
print(f"   Parameters: {total_params:,}")
print(f"   Size:       ~{total_params * 4 / 1024 / 1024:.1f} MB")

print(f"\nSaved Files in '{OUTPUT_DIR}/':")
print(f"    best_model.pth         - Best model weights")
print(f"    training_curves.png    - Accuracy and loss plots")
print(f"    sample_predictions.png - Visual predictions")
print(f"    per_class_accuracy.png - Class-wise performance")
print(f"    confusion_matrix.png   - Confusion analysis")

print(f"\n{'='*70}")
print("Training completed successfully!")
print("="*70)
```

```
======================================================================
TRAINING SUMMARY
======================================================================

Results:
   Best Accuracy:  94.48% (Epoch 149)
   Final Accuracy: 94.39%
   Total Epochs:   150

Model:
   Parameters: 4,079,429
   Size:       ~15.6 MB

Saved Files in 'cifar10_results/':
    best_model.pth         - Best model weights
    training_curves.png    - Accuracy and loss plots
    sample_predictions.png - Visual predictions
    per_class_accuracy.png - Class-wise performance
    confusion_matrix.png   - Confusion analysis

======================================================================
Training completed successfully!
======================================================================
```