Lab-5: CIFAR-10 Image Classification: Custom CNN vs AlexNet

R Abhijit Srivathsan - 2448044

Project Overview

This notebook implements and compares two approaches for CIFAR-10 image classification:

1. Custom CNN architecture
2. AlexNet architecture adapted for CIFAR-10

Dataset: CIFAR-10

- 60,000 32x32 color images in 10 classes
- 50,000 training images and 10,000 test images
- Classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

In [1]:
```
# Import required libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report, confusion_matrix
import time

# Set random seeds for reproducibility
tf.random.set_seed(42)
np.random.seed(42)

print(f"TensorFlow version: {tf.__version__}")
print(f"GPU Available: {tf.config.list_physical_devices('GPU')}")
```

```
2025-07-28 20:13:08.597217: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightl
y different numerical results due to floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-07-28 20:13:08.606302: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:467] Unable to register cuFFT
factory: Attempting to register factory for plugin cuFFT when one has already been registered
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1753713788.616927   16550 cuda_dnn.cc:8579] Unable to register cuDNN factory: Attempting to register fac
tory for plugin cuDNN when one has already been registered
E0000 00:00:1753713788.620016   16550 cuda_blas.cc:1407] Unable to register cuBLAS factory: Attempting to register f
actory for plugin cuBLAS when one has already been registered
W0000 00:00:1753713788.627938   16550 computation_placer.cc:177] computation placer already registered. Please check
linkage and avoid linking the same target more than once.
W0000 00:00:1753713788.627954   16550 computation_placer.cc:177] computation placer already registered. Please check
linkage and avoid linking the same target more than once.
W0000 00:00:1753713788.627956   16550 computation_placer.cc:177] computation placer already registered. Please check
linkage and avoid linking the same target more than once.
W0000 00:00:1753713788.627957   16550 computation_placer.cc:177] computation placer already registered. Please check
linkage and avoid linking the same target more than once.
2025-07-28 20:13:08.630928: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized
to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
/home/abhijit/miniconda3/envs/tf-env/lib/python3.12/site-packages/requests/__init__.py:86: RequestsDependencyWarning
: Unable to find acceptable character detection dependency (chardet or charset_normalizer).
  warnings.warn(
TensorFlow version: 2.19.0
GPU Available: [PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

In [2]:

```python
# Load and preprocess CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Class names for CIFAR-10
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

print(f"Training data shape: {x_train.shape}")
print(f"Training labels shape: {y_train.shape}")
print(f"Test data shape: {x_test.shape}")
print(f"Test labels shape: {y_test.shape}")
print(f"Number of classes: {len(class_names)}")
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 529s 3us/step
Training data shape: (50000, 32, 32, 3)
Training labels shape: (50000, 1)
Test data shape: (10000, 32, 32, 3)
Test labels shape: (10000, 1)
Number of classes: 10
```

In [3]:

```
# Visualize sample images from CIFAR-10
plt.figure(figsize=(12, 8))
for i in range(20):
    plt.subplot(4, 5, i + 1)
    plt.imshow(x_train[i])
    plt.title(f'{class_names[y_train[i][0]]}')
    plt.axis('off')
plt.suptitle('Sample CIFAR-10 Images', fontsize=16)
plt.tight_layout()
plt.show()
```


Sample CIFAR-10 Images
frog
truck
truck
deer
automobile
automobile
bird
horse
ship
cat
deer
horse
horse
bird
truck
truck
truck
cat
bird
frog

In [4]:
```python
# Data preprocessing
# Normalize pixel values to [0, 1]
x_train_normalized = x_train.astype('float32') / 255.0
x_test_normalized = x_test.astype('float32') / 255.0

# Convert labels to categorical (one-hot encoding)
y_train_categorical = keras.utils.to_categorical(y_train, 10)
y_test_categorical = keras.utils.to_categorical(y_test, 10)

print(f"Normalized training data range: [{x_train_normalized.min():.2f}, {x_train_normalized.max():.2f}]")
print(f"Label shape after one-hot encoding: {y_train_categorical.shape}")
```

```
Normalized training data range: [0.00, 1.00]
Label shape after one-hot encoding: (50000, 10)
```

Model 1: Custom CNN Architecture

In [5]:
```python
def create_custom_cnn():
    """
    Create a custom CNN architecture optimized for CIFAR-10
    """
    model = models.Sequential([
        # First Convolutional Block
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'),
        layers.BatchNormalization(),
        layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Second Convolutional Block
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Third Convolutional Block
        layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
```

```python
        layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Fully Connected Layers
        layers.Flatten(),
        layers.Dense(512, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(256, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])

    return model

# Create and compile custom CNN
custom_cnn = create_custom_cnn()
custom_cnn.compile(optimizer='adam',
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

print("Custom CNN Architecture:")
custom_cnn.summary()
```

```
/home/abhijit/miniconda3/envs/tf-env/lib/python3.12/site-packages/keras/src/layers/convolutional/base_conv.py:113: U
serWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using
an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
I0000 00:00:1753714324.161874   16550 gpu_device.cc:2019] Created device /job:localhost/replica:0/task:0/device:GP
U:0 with 3775 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 4050 Laptop GPU, pci bus id: 0000:01:00.0, compute
capability: 8.9
Custom CNN Architecture:
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 8, 8, 128)	512
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147,584
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 512)	1,049,088
batch_normalization_3	(None, 512)	2,048

(BatchNormalization)		
dropout_3 (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131,328
dropout_4 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 10)	2,570

Total params: 1,472,938 (5.62 MB)

Trainable params: 1,471,466 (5.61 MB)

Non-trainable params: 1,472 (5.75 KB)

Model 2: AlexNet Architecture (Adapted for CIFAR-10)

In [6]:

```
def create_alexnet_cifar10():
    """
    Create AlexNet architecture adapted for CIFAR-10 (32x32 input)
    Original AlexNet was designed for 224x224 images, so we adapt it for smaller CIFAR-10 images
    """
    model = models.Sequential([
        # First Convolutional Layer
        layers.Conv2D(96, (5, 5), strides=(1, 1), activation='relu', input_shape=(32, 32, 3), padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2), strides=(2, 2)),

        # Second Convolutional Layer
        layers.Conv2D(256, (5, 5), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2), strides=(2, 2)),

        # Third Convolutional Layer
        layers.Conv2D(384, (3, 3), activation='relu', padding='same'),

        # Fourth Convolutional Layer
        layers.Conv2D(384, (3, 3), activation='relu', padding='same'),
```

```python
        # Fifth Convolutional Layer
        layers.Conv2D(256, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2), strides=(2, 2)),

        # Fully Connected Layers
        layers.Flatten(),
        layers.Dense(4096, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(4096, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])

    return model

# Create and compile AlexNet
alexnet_model = create_alexnet_cifar10()
alexnet_model.compile(optimizer='adam',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

print("AlexNet Architecture (Adapted for CIFAR-10):")
alexnet_model.summary()
```

```
AlexNet Architecture (Adapted for CIFAR-10):
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 32, 32, 96)	7,296
batch_normalization_4 (BatchNormalization)	(None, 32, 32, 96)	384
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 96)	0
conv2d_7 (Conv2D)	(None, 16, 16, 256)	614,656
batch_normalization_5 (BatchNormalization)	(None, 16, 16, 256)	1,024
max_pooling2d_4 (MaxPooling2D)	(None, 8, 8, 256)	0
conv2d_8 (Conv2D)	(None, 8, 8, 384)	885,120
conv2d_9 (Conv2D)	(None, 8, 8, 384)	1,327,488
conv2d_10 (Conv2D)	(None, 8, 8, 256)	884,992
max_pooling2d_5 (MaxPooling2D)	(None, 4, 4, 256)	0
flatten_1 (Flatten)	(None, 4096)	0
dense_3 (Dense)	(None, 4096)	16,781,312
dropout_5 (Dropout)	(None, 4096)	0
dense_4 (Dense)	(None, 4096)	16,781,312
dropout_6 (Dropout)	(None, 4096)	0
dense_5 (Dense)	(None, 10)	40,970

Total params: 37,324,554 (142.38 MB)

Trainable params: 37,323,850 (142.38 MB)

Non-trainable params: 704 (2.75 KB)

Training Configuration and Callbacks

In [7]:
```
# Training configuration
EPOCHS = 50
BATCH_SIZE = 128
VALIDATION_SPLIT = 0.1

# Create callbacks for better training
def create_callbacks(model_name):
    return [
        keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            patience=10,
            restore_best_weights=True
        ),
        keras.callbacks.ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.2,
            patience=5,
            min_lr=0.0001
        ),
        keras.callbacks.ModelCheckpoint(
            f'best_{model_name}_model.h5',
            monitor='val_accuracy',
            save_best_only=True,
            verbose=1
        )
    ]
```

Training Custom CNN

In [8]:
```
print("Training Custom CNN...")
start_time = time.time()

# Train custom CNN
custom_cnn_history = custom_cnn.fit(
    x_train_normalized, y_train_categorical,
```

```python
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_split=VALIDATION_SPLIT,
    callbacks=create_callbacks('custom_cnn'),
    verbose=1
)

custom_cnn_training_time = time.time() - start_time
print(f"Custom CNN training completed in {custom_cnn_training_time:.2f} seconds")
```

```
Training Custom CNN...
Epoch 1/50
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1753714327.019846   20630 service.cc:152] XLA service 0x76744c007380 initialized for platform CUDA (this
does not guarantee that XLA will be used). Devices:
I0000 00:00:1753714327.019869   20630 service.cc:160]   StreamExecutor device (0): NVIDIA GeForce RTX 4050 Laptop GP
U, Compute Capability 8.9
2025-07-28 20:22:07.077048: I tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:269] disabling MLIR crash
reproducer, set env var `MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
I0000 00:00:1753714327.430286   20630 cuda_dnn.cc:529] Loaded cuDNN version 90300
 11/352 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.1285 - loss: 3.7682
I0000 00:00:1753714334.588397   20630 device_compiler.h:188] Compiled cluster using XLA!  This line is logged at mos
t once for the lifetime of the process.
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.2753 - loss: 2.3329
2025-07-28 20:22:21.261254: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_3803', 60 bytes spill stores, 60 bytes spill
loads

352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - accuracy: 0.2759 - loss: 2.3293
2025-07-28 20:22:26.415992: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_207', 4 bytes spill stores, 4 bytes spill lo
ads

2025-07-28 20:22:26.426765: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_207', 4 bytes spill stores, 4 bytes spill lo
ads

Epoch 1: val_accuracy improved from -inf to 0.12880, saving model to best_custom_cnn_model.h5
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 24s 42ms/step - accuracy: 0.2761 - loss: 2.3281 - val_accuracy: 0.1288 - val_loss: 3.68
80 - learning_rate: 0.0010
Epoch 2/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.4781 - loss: 1.4330
Epoch 2: val_accuracy improved from 0.12880 to 0.58760, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.4783 - loss: 1.4325 - val_accuracy: 0.5876 - val_loss: 1.145
5 - learning_rate: 0.0010
Epoch 3/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.5825 - loss: 1.1629
Epoch 3: val_accuracy improved from 0.58760 to 0.65720, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.5826 - loss: 1.1628 - val_accuracy: 0.6572 - val_loss: 0.980
7 - learning_rate: 0.0010
Epoch 4/50
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.6506 - loss: 0.9878
Epoch 4: val_accuracy improved from 0.65720 to 0.71900, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.6507 - loss: 0.9875 - val_accuracy: 0.7190 - val_loss: 0.797
5 - learning_rate: 0.0010
Epoch 5/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.6896 - loss: 0.8752
Epoch 5: val_accuracy did not improve from 0.71900
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.6896 - loss: 0.8751 - val_accuracy: 0.7190 - val_loss: 0.815
3 - learning_rate: 0.0010
Epoch 6/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7208 - loss: 0.8054
Epoch 6: val_accuracy improved from 0.71900 to 0.74840, saving model to best_custom_cnn_model.h5
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.7208 - loss: 0.8053 - val_accuracy: 0.7484 - val_loss: 0.7164 - learning_rate: 0.0010
Epoch 7/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7437 - loss: 0.7378
Epoch 7: val_accuracy improved from 0.74840 to 0.76560, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.7438 - loss: 0.7377 - val_accuracy: 0.7656 - val_loss: 0.7203 - learning_rate: 0.0010
Epoch 8/50
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7590 - loss: 0.6984
Epoch 8: val_accuracy improved from 0.76560 to 0.77280, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.7590 - loss: 0.6982 - val_accuracy: 0.7728 - val_loss: 0.6670 - learning_rate: 0.0010
Epoch 9/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7767 - loss: 0.6404
Epoch 9: val_accuracy improved from 0.77280 to 0.77940, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.7768 - loss: 0.6403 - val_accuracy: 0.7794 - val_loss: 0.6639 - learning_rate: 0.0010
Epoch 10/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7897 - loss: 0.6095
Epoch 10: val_accuracy improved from 0.77940 to 0.79800, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.7897 - loss: 0.6094 - val_accuracy: 0.7980 - val_loss: 0.6023 - learning_rate: 0.0010
Epoch 11/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8004 - loss: 0.5823
Epoch 11: val_accuracy improved from 0.79800 to 0.80620, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.8005 - loss: 0.5822 - val_accuracy: 0.8062 - val_loss: 0.5710 - learning_rate: 0.0010
Epoch 12/50
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8111 - loss: 0.5496
Epoch 12: val_accuracy did not improve from 0.80620
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.8111 - loss: 0.5495 - val_accuracy: 0.7980 - val_loss: 0.6106 - learning_rate: 0.0010
Epoch 13/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8224 - loss: 0.5189
Epoch 13: val_accuracy improved from 0.80620 to 0.81600, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.8224 - loss: 0.5188 - val_accuracy: 0.8160 - val_loss: 0.5592 - learning_rate: 0.0010
Epoch 14/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8320 - loss: 0.4929
Epoch 14: val_accuracy did not improve from 0.81600
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.8320 - loss: 0.4929 - val_accuracy: 0.8114 - val_loss: 0.5848 - learning_rate: 0.0010
Epoch 15/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8350 - loss: 0.4794
Epoch 15: val_accuracy did not improve from 0.81600
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.8351 - loss: 0.4793 - val_accuracy: 0.8112 - val_loss: 0.5624 - learning_rate: 0.0010
Epoch 16/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8424 - loss: 0.4591
Epoch 16: val_accuracy improved from 0.81600 to 0.82940, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.8424 - loss: 0.4591 - val_accuracy: 0.8294 - val_loss: 0.516
7 - learning_rate: 0.0010
Epoch 17/50
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8510 - loss: 0.4357
Epoch 17: val_accuracy did not improve from 0.82940
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.8510 - loss: 0.4356 - val_accuracy: 0.8274 - val_loss: 0.512
8 - learning_rate: 0.0010
Epoch 18/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8567 - loss: 0.4174
Epoch 18: val_accuracy did not improve from 0.82940
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.8567 - loss: 0.4174 - val_accuracy: 0.8138 - val_loss: 0.567
3 - learning_rate: 0.0010
Epoch 19/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8595 - loss: 0.4067
Epoch 19: val_accuracy improved from 0.82940 to 0.83540, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.8596 - loss: 0.4067 - val_accuracy: 0.8354 - val_loss: 0.518
4 - learning_rate: 0.0010
Epoch 20/50
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8679 - loss: 0.3866
Epoch 20: val_accuracy did not improve from 0.83540
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.8679 - loss: 0.3865 - val_accuracy: 0.8304 - val_loss: 0.606
0 - learning_rate: 0.0010
Epoch 21/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8719 - loss: 0.3756
Epoch 21: val_accuracy did not improve from 0.83540
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.8719 - loss: 0.3756 - val_accuracy: 0.8144 - val_loss: 0.603
3 - learning_rate: 0.0010
Epoch 22/50
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8719 - loss: 0.3677
Epoch 22: val_accuracy improved from 0.83540 to 0.83980, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.8720 - loss: 0.3676 - val_accuracy: 0.8398 - val_loss: 0.518
3 - learning_rate: 0.0010
Epoch 23/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.8888 - loss: 0.3171
Epoch 23: val_accuracy improved from 0.83980 to 0.85800, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.8888 - loss: 0.3170 - val_accuracy: 0.8580 - val_loss: 0.472
4 - learning_rate: 2.0000e-04
Epoch 24/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.9029 - loss: 0.2776
Epoch 24: val_accuracy did not improve from 0.85800
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.9030 - loss: 0.2775 - val_accuracy: 0.8580 - val_loss: 0.470
4 - learning_rate: 2.0000e-04
Epoch 25/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.9084 - loss: 0.2674
Epoch 25: val_accuracy improved from 0.85800 to 0.86100, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.9084 - loss: 0.2673 - val_accuracy: 0.8610 - val_loss: 0.468
0 - learning_rate: 2.0000e-04
Epoch 26/50
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.9097 - loss: 0.2549
Epoch 26: val_accuracy improved from 0.86100 to 0.86160, saving model to best_custom_cnn_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.9097 - loss: 0.2547 - val_accuracy: 0.8616 - val_loss: 0.472
8 - learning_rate: 2.0000e-04
Epoch 27/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.9161 - loss: 0.2408
Epoch 27: val_accuracy did not improve from 0.86160
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.9162 - loss: 0.2408 - val_accuracy: 0.8602 - val_loss: 0.476
0 - learning_rate: 2.0000e-04
Epoch 28/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.9177 - loss: 0.2338
Epoch 28: val_accuracy did not improve from 0.86160
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.9177 - loss: 0.2337 - val_accuracy: 0.8578 - val_loss: 0.491
0 - learning_rate: 2.0000e-04
Epoch 29/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.9214 - loss: 0.2205
Epoch 29: val_accuracy improved from 0.86160 to 0.86440, saving model to best_custom_cnn_model.h5
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.9214 - loss: 0.2204 - val_accuracy: 0.8644 - val_loss: 0.485
2 - learning_rate: 2.0000e-04
Epoch 30/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.9237 - loss: 0.2209
Epoch 30: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9237 - loss: 0.2209 - val_accuracy: 0.8616 - val_loss: 0.482
8 - learning_rate: 2.0000e-04
Epoch 31/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9253 - loss: 0.2126
Epoch 31: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9253 - loss: 0.2126 - val_accuracy: 0.8642 - val_loss: 0.491
6 - learning_rate: 1.0000e-04
Epoch 32/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9270 - loss: 0.2080
Epoch 32: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9270 - loss: 0.2079 - val_accuracy: 0.8634 - val_loss: 0.485
1 - learning_rate: 1.0000e-04
Epoch 33/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9291 - loss: 0.1962
Epoch 33: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9291 - loss: 0.1961 - val_accuracy: 0.8638 - val_loss: 0.491
2 - learning_rate: 1.0000e-04
Epoch 34/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9315 - loss: 0.1948
Epoch 34: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9315 - loss: 0.1947 - val_accuracy: 0.8602 - val_loss: 0.495
3 - learning_rate: 1.0000e-04
Epoch 35/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9333 - loss: 0.1911
Epoch 35: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9333 - loss: 0.1910 - val_accuracy: 0.8606 - val_loss: 0.489
7 - learning_rate: 1.0000e-04
Epoch 36/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9339 - loss: 0.1846
Epoch 36: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9339 - loss: 0.1845 - val_accuracy: 0.8604 - val_loss: 0.491
1 - learning_rate: 1.0000e-04
Epoch 37/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9362 - loss: 0.1829
Epoch 37: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9362 - loss: 0.1829 - val_accuracy: 0.8616 - val_loss: 0.491
```

```
9 - learning_rate: 1.0000e-04
Epoch 38/50
349/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9374 - loss: 0.1767
Epoch 38: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9374 - loss: 0.1767 - val_accuracy: 0.8630 - val_loss: 0.501
7 - learning_rate: 1.0000e-04
Epoch 39/50
350/352 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9375 - loss: 0.1803
Epoch 39: val_accuracy did not improve from 0.86440
352/352 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - accuracy: 0.9375 - loss: 0.1802 - val_accuracy: 0.8604 - val_loss: 0.505
1 - learning_rate: 1.0000e-04
Custom CNN training completed in 257.18 seconds
```

Training AlexNet

In [9]:

```
print("Training AlexNet...")
start_time = time.time()

# Train AlexNet
alexnet_history = alexnet_model.fit(
    x_train_normalized, y_train_categorical,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_split=VALIDATION_SPLIT,
    callbacks=create_callbacks('alexnet'),
    verbose=1
)

alexnet_training_time = time.time() - start_time
print(f"AlexNet training completed in {alexnet_training_time:.2f} seconds")
```

```
Training AlexNet...
Epoch 1/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 74ms/step - accuracy: 0.2323 - loss: 2.4036
2025-07-28 20:27:04.625738: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_2024', 60 bytes spill stores, 60 bytes spill
loads

352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 107ms/step - accuracy: 0.2325 - loss: 2.4020
```

```
2025-07-28 20:27:15.896641: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 532 bytes spill stores, 532 bytes spil
l loads

2025-07-28 20:27:16.015756: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 3148 bytes spill stores, 3124 bytes sp
ill loads

2025-07-28 20:27:16.055718: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 488 bytes spill stores, 488 bytes spil
l loads

2025-07-28 20:27:16.146785: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 4 bytes spill stores, 4 bytes spill lo
ads

2025-07-28 20:27:16.224192: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 820 bytes spill stores, 820 bytes spil
l loads

2025-07-28 20:27:16.289605: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160_0', 136 bytes spill stores, 136 bytes sp
ill loads

2025-07-28 20:27:16.674302: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160_0', 1604 bytes spill stores, 1552 bytes
spill loads

2025-07-28 20:27:16.722740: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 8 bytes spill stores, 8 bytes spill lo
ads

2025-07-28 20:27:16.815072: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 4 bytes spill stores, 4 bytes spill lo
ads

2025-07-28 20:27:16.863591: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 128 bytes spill stores, 128 bytes spil
l loads
```

```
2025-07-28 20:27:16.896665: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 64 bytes spill stores, 64 bytes spill
loads

2025-07-28 20:27:17.022873: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 76 bytes spill stores, 76 bytes spill
loads

2025-07-28 20:27:17.181230: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 4 bytes spill stores, 4 bytes spill lo
ads

2025-07-28 20:27:17.298295: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 896 bytes spill stores, 896 bytes spil
l loads

2025-07-28 20:27:17.470751: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 4284 bytes spill stores, 4260 bytes sp
ill loads

2025-07-28 20:27:19.767977: E external/local_xla/xla/service/slow_operation_alarm.cc:73] Trying algorithm eng0{} for
conv %cudnn-conv-bias-activation.16 = (f32[128,256,16,16]{3,2,1,0}, u8[0]{0}) custom-call(f32[128,96,16,16]{3,2,1,0}
%bitcast.729, f32[256,96,5,5]{3,2,1,0} %bitcast.736, f32[256]{0} %bitcast.738), window={size=5x5 pad=2_2x2_2}, dim_l
abels=bf01_oi01->bf01, custom_call_target="__cudnn$convBiasActivationForward", metadata={op_type="Conv2D" op_name="s
equential_1_1/conv2d_7_1/convolution" source_file="/home/abhijit/miniconda3/envs/tf-env/lib/python3.12/site-package
s/tensorflow/python/framework/ops.py" source_line=1200}, backend_config={"operation_queue_id":"0","wait_on_operation
_queues":[],"cudnn_conv_backend_config":{"conv_result_scale":1,"activation_mode":"kRelu","side_input_scale":0,"leaky
relu_alpha":0},"force_earliest_schedule":false} is taking a while...
2025-07-28 20:27:20.364799: E external/local_xla/xla/service/slow_operation_alarm.cc:140] The operation took 1.59693
629s
Trying algorithm eng0{} for conv %cudnn-conv-bias-activation.16 = (f32[128,256,16,16]{3,2,1,0}, u8[0]{0}) custom-cal
l(f32[128,96,16,16]{3,2,1,0} %bitcast.729, f32[256,96,5,5]{3,2,1,0} %bitcast.736, f32[256]{0} %bitcast.738), window=
{size=5x5 pad=2_2x2_2}, dim_labels=bf01_oi01->bf01, custom_call_target="__cudnn$convBiasActivationForward", metadata
={op_type="Conv2D" op_name="sequential_1_1/conv2d_7_1/convolution" source_file="/home/abhijit/miniconda3/envs/tf-en
v/lib/python3.12/site-packages/tensorflow/python/framework/ops.py" source_line=1200}, backend_config={"operation_que
ue_id":"0","wait_on_operation_queues":[],"cudnn_conv_backend_config":{"conv_result_scale":1,"activation_mode":"kRel
u","side_input_scale":0,"leakyrelu_alpha":0},"force_earliest_schedule":false} is taking a while...
2025-07-28 20:27:24.339164: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 56 bytes spill stores, 56 bytes spill
loads
```

```
2025-07-28 20:27:24.376780: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 408 bytes spill stores, 408 bytes spill loads

2025-07-28 20:27:24.590444: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 4 bytes spill stores, 4 bytes spill loads

2025-07-28 20:27:24.836074: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 844 bytes spill stores, 844 bytes spill loads

2025-07-28 20:27:24.996506: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 100 bytes spill stores, 100 bytes spill loads

2025-07-28 20:27:25.056185: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_160_0', 3392 bytes spill stores, 3120 bytes spill loads

Epoch 1: val_accuracy improved from -inf to 0.22380, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 64s 140ms/step - accuracy: 0.2328 - loss: 2.4005 - val_accuracy: 0.2238 - val_loss: 2.4434 - learning_rate: 0.0010
Epoch 2/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - accuracy: 0.4797 - loss: 1.4224
Epoch 2: val_accuracy improved from 0.22380 to 0.41820, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 82ms/step - accuracy: 0.4799 - loss: 1.4220 - val_accuracy: 0.4182 - val_loss: 1.6395 - learning_rate: 0.0010
Epoch 3/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.5732 - loss: 1.1931
Epoch 3: val_accuracy improved from 0.41820 to 0.48860, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 30s 84ms/step - accuracy: 0.5732 - loss: 1.1930 - val_accuracy: 0.4886 - val_loss: 1.45
92 - learning_rate: 0.0010
Epoch 4/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.6336 - loss: 1.0468
Epoch 4: val_accuracy improved from 0.48860 to 0.53160, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 30s 84ms/step - accuracy: 0.6336 - loss: 1.0467 - val_accuracy: 0.5316 - val_loss: 1.42
76 - learning_rate: 0.0010
Epoch 5/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.6816 - loss: 0.9027
Epoch 5: val_accuracy improved from 0.53160 to 0.54560, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 82ms/step - accuracy: 0.6816 - loss: 0.9026 - val_accuracy: 0.5456 - val_loss: 1.29
56 - learning_rate: 0.0010
Epoch 6/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - accuracy: 0.7200 - loss: 0.8033
Epoch 6: val_accuracy improved from 0.54560 to 0.61180, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 82ms/step - accuracy: 0.7201 - loss: 0.8032 - val_accuracy: 0.6118 - val_loss: 1.10
90 - learning_rate: 0.0010
Epoch 7/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 77ms/step - accuracy: 0.7431 - loss: 0.7411
Epoch 7: val_accuracy improved from 0.61180 to 0.67320, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 81ms/step - accuracy: 0.7432 - loss: 0.7410 - val_accuracy: 0.6732 - val_loss: 0.95
36 - learning_rate: 0.0010
Epoch 8/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.7690 - loss: 0.6664
Epoch 8: val_accuracy did not improve from 0.67320
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 80ms/step - accuracy: 0.7690 - loss: 0.6664 - val_accuracy: 0.4462 - val_loss: 1.84
38 - learning_rate: 0.0010
Epoch 9/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 77ms/step - accuracy: 0.7878 - loss: 0.6154
Epoch 9: val_accuracy did not improve from 0.67320
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.7878 - loss: 0.6154 - val_accuracy: 0.5324 - val_loss: 1.44
36 - learning_rate: 0.0010
Epoch 10/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.8059 - loss: 0.5608
Epoch 10: val_accuracy did not improve from 0.67320
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 83ms/step - accuracy: 0.8059 - loss: 0.5607 - val_accuracy: 0.6706 - val_loss: 1.06
98 - learning_rate: 0.0010
Epoch 11/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - accuracy: 0.8226 - loss: 0.5160
Epoch 11: val_accuracy did not improve from 0.67320
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 81ms/step - accuracy: 0.8226 - loss: 0.5159 - val_accuracy: 0.6592 - val_loss: 1.11
02 - learning_rate: 0.0010
Epoch 12/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - accuracy: 0.8341 - loss: 0.4824
Epoch 12: val_accuracy did not improve from 0.67320
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 81ms/step - accuracy: 0.8341 - loss: 0.4823 - val_accuracy: 0.6520 - val_loss: 1.08
95 - learning_rate: 0.0010
Epoch 13/50
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - accuracy: 0.8838 - loss: 0.3427
Epoch 13: val_accuracy improved from 0.67320 to 0.75220, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 83ms/step - accuracy: 0.8838 - loss: 0.3424 - val_accuracy: 0.7522 - val_loss: 0.91
02 - learning_rate: 2.0000e-04
Epoch 14/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.9426 - loss: 0.1688
Epoch 14: val_accuracy improved from 0.75220 to 0.75820, saving model to best_alexnet_model.h5
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 81ms/step - accuracy: 0.9426 - loss: 0.1686 - val_accuracy: 0.7582 - val_loss: 1.1199 - learning_rate: 2.0000e-04
Epoch 15/50
351/352 ━━━━━━━━━━━━━━━━━━━ 0s 77ms/step - accuracy: 0.9724 - loss: 0.0824
Epoch 15: val_accuracy did not improve from 0.75820
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.9724 - loss: 0.0823 - val_accuracy: 0.7496 - val_loss: 1.4581 - learning_rate: 2.0000e-04
Epoch 16/50
351/352 ━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - accuracy: 0.9835 - loss: 0.0511
Epoch 16: val_accuracy did not improve from 0.75820
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 81ms/step - accuracy: 0.9835 - loss: 0.0511 - val_accuracy: 0.7556 - val_loss: 1.6869 - learning_rate: 2.0000e-04
Epoch 17/50
351/352 ━━━━━━━━━━━━━━━━━━━ 0s 77ms/step - accuracy: 0.9877 - loss: 0.0378
Epoch 17: val_accuracy did not improve from 0.75820
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 78ms/step - accuracy: 0.9877 - loss: 0.0377 - val_accuracy: 0.7536 - val_loss: 1.7954 - learning_rate: 2.0000e-04
Epoch 18/50
351/352 ━━━━━━━━━━━━━━━━━━━ 0s 77ms/step - accuracy: 0.9885 - loss: 0.0344
Epoch 18: val_accuracy did not improve from 0.75820
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.9885 - loss: 0.0344 - val_accuracy: 0.7582 - val_loss: 1.7593 - learning_rate: 2.0000e-04
Epoch 19/50
351/352 ━━━━━━━━━━━━━━━━━━━ 0s 77ms/step - accuracy: 0.9928 - loss: 0.0225
Epoch 19: val_accuracy improved from 0.75820 to 0.76780, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 80ms/step - accuracy: 0.9928 - loss: 0.0225 - val_accuracy: 0.7678 - val_loss: 1.8207 - learning_rate: 1.0000e-04
Epoch 20/50
351/352 ━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.9958 - loss: 0.0133
Epoch 20: val_accuracy improved from 0.76780 to 0.76920, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 81ms/step - accuracy: 0.9959 - loss: 0.0133 - val_accuracy: 0.7692 - val_loss: 1.92
72 - learning_rate: 1.0000e-04
Epoch 21/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.9985 - loss: 0.0060
Epoch 21: val_accuracy improved from 0.76920 to 0.77240, saving model to best_alexnet_model.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.ker
as')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 29s 81ms/step - accuracy: 0.9985 - loss: 0.0060 - val_accuracy: 0.7724 - val_loss: 1.97
47 - learning_rate: 1.0000e-04
Epoch 22/50
351/352 0s 79ms/step - accuracy: 0.9983 - loss: 0.0060
Epoch 22: val_accuracy did not improve from 0.77240
352/352 29s 81ms/step - accuracy: 0.9983 - loss: 0.0060 - val_accuracy: 0.7706 - val_loss: 2.08
02 - learning_rate: 1.0000e-04
Epoch 23/50
352/352 0s 81ms/step - accuracy: 0.9985 - loss: 0.0046
Epoch 23: val_accuracy did not improve from 0.77240
352/352 29s 83ms/step - accuracy: 0.9985 - loss: 0.0046 - val_accuracy: 0.7698 - val_loss: 2.10
56 - learning_rate: 1.0000e-04
Epoch 24/50
351/352 0s 85ms/step - accuracy: 0.9989 - loss: 0.0041
Epoch 24: val_accuracy did not improve from 0.77240
352/352 31s 87ms/step - accuracy: 0.9989 - loss: 0.0041 - val_accuracy: 0.7658 - val_loss: 2.35
12 - learning_rate: 1.0000e-04
Epoch 25/50
352/352 0s 80ms/step - accuracy: 0.9988 - loss: 0.0037
Epoch 25: val_accuracy did not improve from 0.77240
352/352 29s 81ms/step - accuracy: 0.9988 - loss: 0.0037 - val_accuracy: 0.7712 - val_loss: 2.40
86 - learning_rate: 1.0000e-04
Epoch 26/50
352/352 0s 79ms/step - accuracy: 0.9975 - loss: 0.0086
Epoch 26: val_accuracy did not improve from 0.77240
352/352 29s 81ms/step - accuracy: 0.9975 - loss: 0.0086 - val_accuracy: 0.7578 - val_loss: 2.42
84 - learning_rate: 1.0000e-04
Epoch 27/50
351/352 0s 74ms/step - accuracy: 0.9971 - loss: 0.0105
Epoch 27: val_accuracy did not improve from 0.77240
352/352 27s 76ms/step - accuracy: 0.9971 - loss: 0.0105 - val_accuracy: 0.7696 - val_loss: 2.31
89 - learning_rate: 1.0000e-04
Epoch 28/50
351/352 0s 64ms/step - accuracy: 0.9989 - loss: 0.0040
Epoch 28: val_accuracy did not improve from 0.77240
352/352 23s 65ms/step - accuracy: 0.9989 - loss: 0.0040 - val_accuracy: 0.7712 - val_loss: 2.39
14 - learning_rate: 1.0000e-04
Epoch 29/50
351/352 0s 57ms/step - accuracy: 0.9979 - loss: 0.0067
Epoch 29: val_accuracy did not improve from 0.77240
352/352 20s 58ms/step - accuracy: 0.9979 - loss: 0.0066 - val_accuracy: 0.7720 - val_loss: 2.38
```

```
63 - learning_rate: 1.0000e-04
Epoch 30/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.9985 - loss: 0.0059
Epoch 30: val_accuracy did not improve from 0.77240
352/352 ━━━━━━━━━━━━━━━━━━━━ 20s 58ms/step - accuracy: 0.9985 - loss: 0.0059 - val_accuracy: 0.7698 - val_loss: 2.61
28 - learning_rate: 1.0000e-04
Epoch 31/50
351/352 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - accuracy: 0.9989 - loss: 0.0040
Epoch 31: val_accuracy did not improve from 0.77240
352/352 ━━━━━━━━━━━━━━━━━━━━ 21s 58ms/step - accuracy: 0.9989 - loss: 0.0040 - val_accuracy: 0.7684 - val_loss: 2.69
89 - learning_rate: 1.0000e-04
AlexNet training completed in 892.38 seconds
```

Model Evaluation and Testing

In [10]:

```
# Evaluate both models on test set
print("Evaluating Custom CNN on test set...")
custom_cnn_test_loss, custom_cnn_test_accuracy = custom_cnn.evaluate(
    x_test_normalized, y_test_categorical, verbose=0
)

print("Evaluating AlexNet on test set...")
alexnet_test_loss, alexnet_test_accuracy = alexnet_model.evaluate(
    x_test_normalized, y_test_categorical, verbose=0
)

# Generate predictions for detailed analysis
custom_cnn_predictions = custom_cnn.predict(x_test_normalized)
alexnet_predictions = alexnet_model.predict(x_test_normalized)

custom_cnn_pred_classes = np.argmax(custom_cnn_predictions, axis=1)
alexnet_pred_classes = np.argmax(alexnet_predictions, axis=1)
true_classes = np.argmax(y_test_categorical, axis=1)

print(f"\nTest Results:")
print(f"Custom CNN - Test Accuracy: {custom_cnn_test_accuracy:.4f} ({custom_cnn_test_accuracy*100:.2f}%)")
print(f"AlexNet - Test Accuracy: {alexnet_test_accuracy:.4f} ({alexnet_test_accuracy*100:.2f}%)")
```

```
Evaluating Custom CNN on test set...
```

```
2025-07-28 20:41:14.684317: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_207', 4 bytes spill stores, 4 bytes spill lo
ads
```

```
Evaluating AlexNet on test set...
```

```
2025-07-28 20:41:17.694746: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 64 bytes spill stores, 64 bytes spill loads

2025-07-28 20:41:17.853248: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 488 bytes spill stores, 488 bytes spill loads

2025-07-28 20:41:17.977578: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 532 bytes spill stores, 532 bytes spill loads

2025-07-28 20:41:17.996921: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_160_0', 48 bytes spill stores, 48 bytes spill loads

2025-07-28 20:41:17.999872: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 820 bytes spill stores, 820 bytes spill loads

2025-07-28 20:41:18.276553: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 4 bytes spill stores, 4 bytes spill loads

2025-07-28 20:41:18.323096: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 3148 bytes spill stores, 3124 bytes spill loads

2025-07-28 20:41:18.546823: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 76 bytes spill stores, 76 bytes spill loads

2025-07-28 20:41:18.614435: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 4284 bytes spill stores, 4260 bytes spill loads

2025-07-28 20:41:18.722345: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warning : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 896 bytes spill stores, 896 bytes spill loads
```

```
2025-07-28 20:41:21.707077: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 64 bytes spill stores, 64 bytes spill
loads

2025-07-28 20:41:22.144993: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 64 bytes spill stores, 64 bytes spill
loads

2025-07-28 20:41:22.170171: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 532 bytes spill stores, 532 bytes spil
l loads

2025-07-28 20:41:22.380889: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_160', 3148 bytes spill stores, 3124 bytes sp
ill loads

2025-07-28 20:41:22.429729: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_167', 896 bytes spill stores, 896 bytes spil
l loads
```

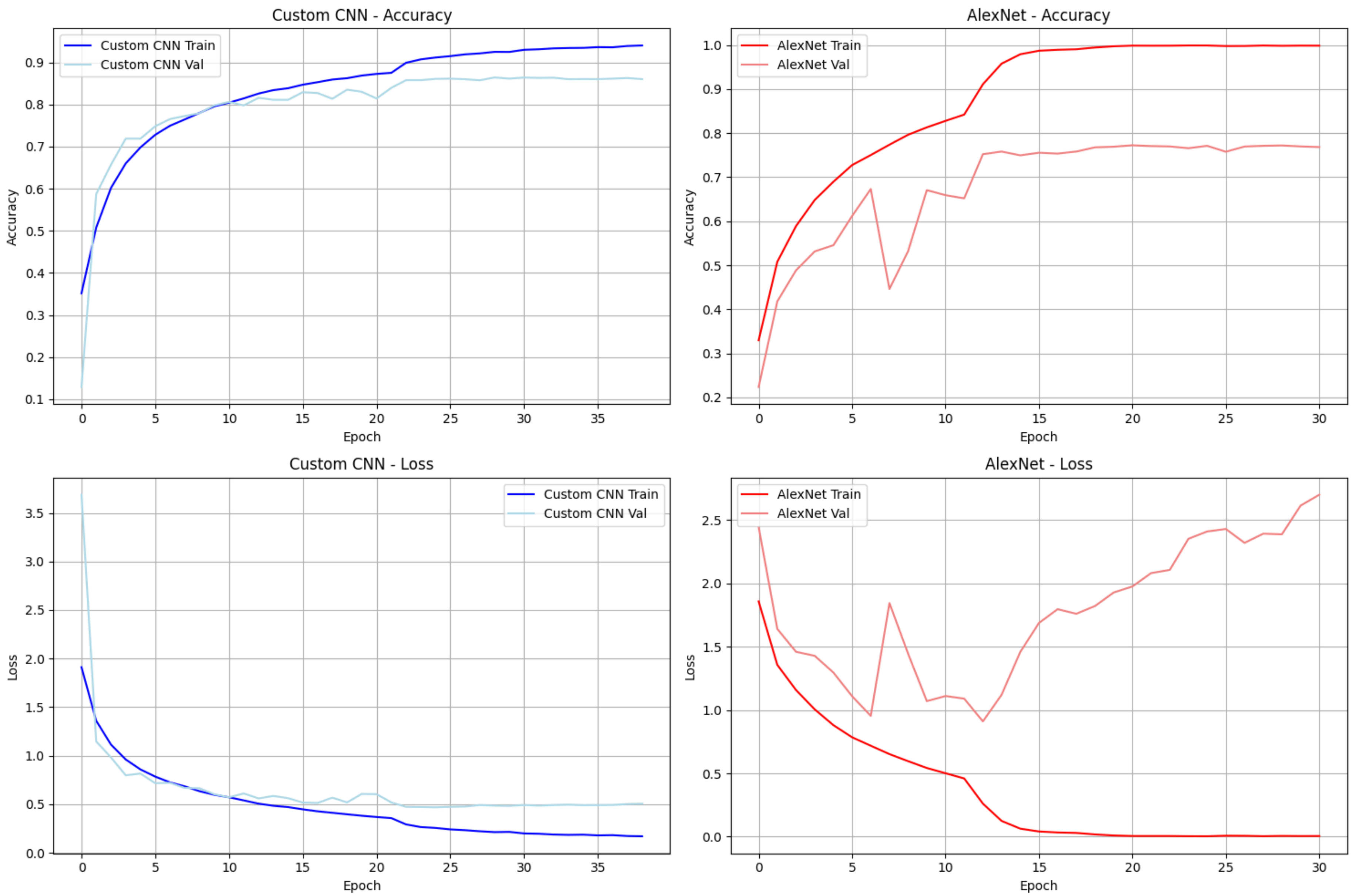
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step

Test Results:
Custom CNN - Test Accuracy: 0.8574 (85.74%)
AlexNet - Test Accuracy: 0.7533 (75.33%)
```

Training History Visualization

In [11]:

```python
# Plot training history
def plot_training_history(custom_history, alexnet_history):
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # Accuracy plots
    axes[0, 0].plot(custom_history.history['accuracy'], label='Custom CNN Train', color='blue')
    axes[0, 0].plot(custom_history.history['val_accuracy'], label='Custom CNN Val', color='lightblue')
    axes[0, 0].set_title('Custom CNN - Accuracy')
    axes[0, 0].set_xlabel('Epoch')
    axes[0, 0].set_ylabel('Accuracy')
```

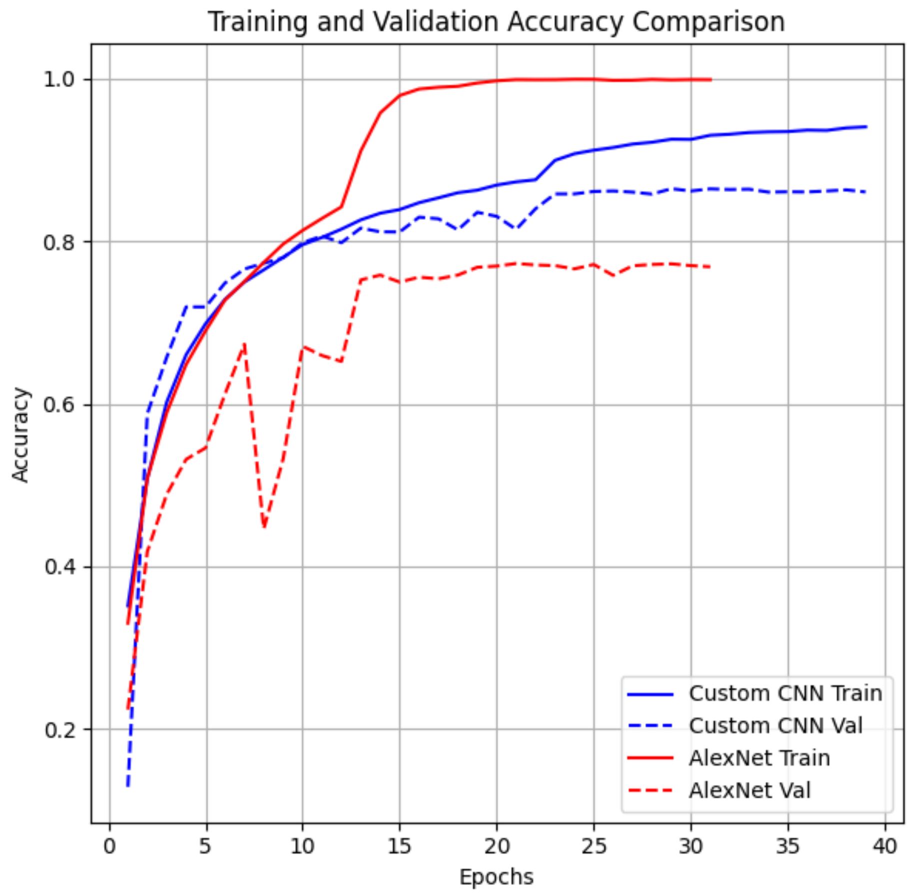
```
    axes[0, 0].legend()
    axes[0, 0].grid(True)

    axes[0, 1].plot(alexnet_history.history['accuracy'], label='AlexNet Train', color='red')
    axes[0, 1].plot(alexnet_history.history['val_accuracy'], label='AlexNet Val', color='lightcoral')
    axes[0, 1].set_title('AlexNet - Accuracy')
    axes[0, 1].set_xlabel('Epoch')
    axes[0, 1].set_ylabel('Accuracy')
    axes[0, 1].legend()
    axes[0, 1].grid(True)

    # Loss plots
    axes[1, 0].plot(custom_history.history['loss'], label='Custom CNN Train', color='blue')
    axes[1, 0].plot(custom_history.history['val_loss'], label='Custom CNN Val', color='lightblue')
    axes[1, 0].set_title('Custom CNN - Loss')
    axes[1, 0].set_xlabel('Epoch')
    axes[1, 0].set_ylabel('Loss')
    axes[1, 0].legend()
    axes[1, 0].grid(True)

    axes[1, 1].plot(alexnet_history.history['loss'], label='AlexNet Train', color='red')
    axes[1, 1].plot(alexnet_history.history['val_loss'], label='AlexNet Val', color='lightcoral')
    axes[1, 1].set_title('AlexNet - Loss')
    axes[1, 1].set_xlabel('Epoch')
    axes[1, 1].set_ylabel('Loss')
    axes[1, 1].legend()
    axes[1, 1].grid(True)

    plt.tight_layout()
    plt.show()

plot_training_history(custom_cnn_history, alexnet_history)
```

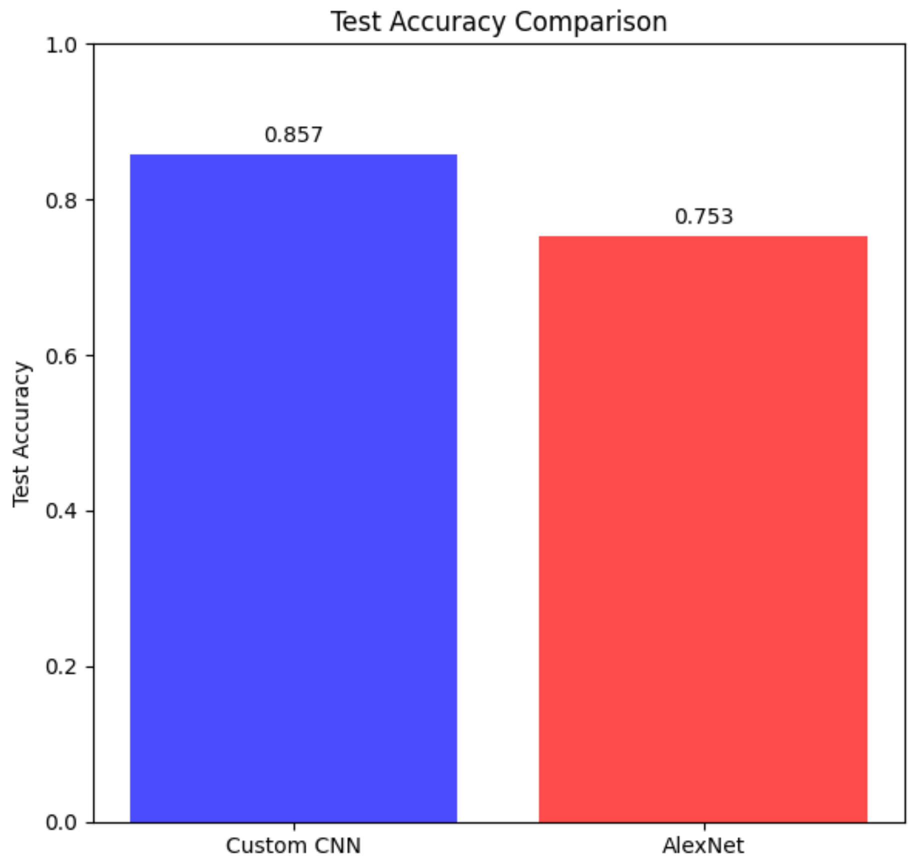


Comparative Analysis

In [12]:

```
# Comparative accuracy plot
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
epochs_range = range(1, len(custom_cnn_history.history['accuracy']) + 1)
plt.plot(epochs_range, custom_cnn_history.history['accuracy'], 'b-', label='Custom CNN Train')
plt.plot(epochs_range, custom_cnn_history.history['val_accuracy'], 'b--', label='Custom CNN Val')

epochs_range_alex = range(1, len(alexnet_history.history['accuracy']) + 1)
plt.plot(epochs_range_alex, alexnet_history.history['accuracy'], 'r-', label='AlexNet Train')
plt.plot(epochs_range_alex, alexnet_history.history['val_accuracy'], 'r--', label='AlexNet Val')

plt.title('Training and Validation Accuracy Comparison')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
models = ['Custom CNN', 'AlexNet']
test_accuracies = [custom_cnn_test_accuracy, alexnet_test_accuracy]
colors = ['blue', 'red']

bars = plt.bar(models, test_accuracies, color=colors, alpha=0.7)
plt.title('Test Accuracy Comparison')
plt.ylabel('Test Accuracy')
plt.ylim(0, 1)

# Add value labels on bars
for bar, acc in zip(bars, test_accuracies):
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
             f'{acc:.3f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()
```





Confusion Matrix and Classification Reports

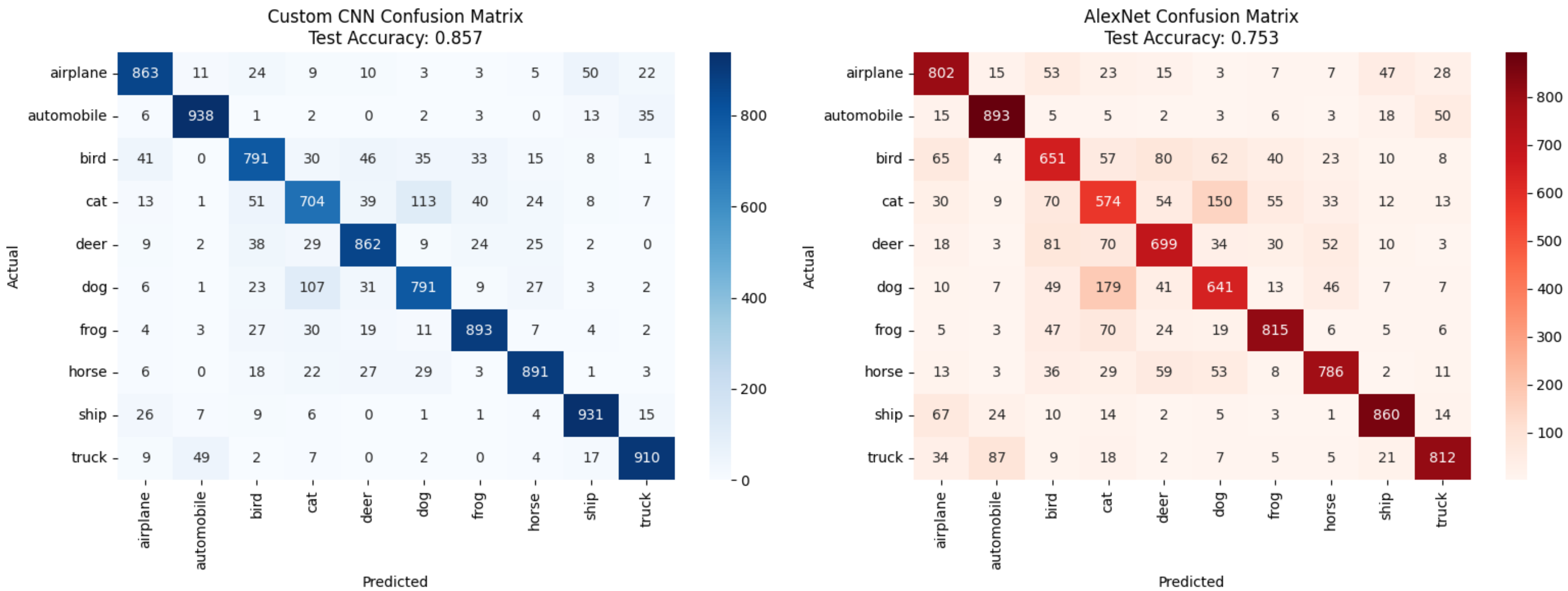
In [13]:
```
# Generate classification reports
print("Classification Report - Custom CNN:")
print(classification_report(true_classes, custom_cnn_pred_classes, target_names=class_names))

print("\nClassification Report - AlexNet:")
print(classification_report(true_classes, alexnet_pred_classes, target_names=class_names))
```

```
Classification Report - Custom CNN:
              precision    recall  f1-score   support

    airplane       0.88      0.86      0.87      1000
  automobile       0.93      0.94      0.93      1000
        bird       0.80      0.79      0.80      1000
         cat       0.74      0.70      0.72      1000
        deer       0.83      0.86      0.85      1000
         dog       0.79      0.79      0.79      1000
        frog       0.89      0.89      0.89      1000
       horse       0.89      0.89      0.89      1000
        ship       0.90      0.93      0.91      1000
       truck       0.91      0.91      0.91      1000

    accuracy                           0.86     10000
   macro avg       0.86      0.86      0.86     10000
weighted avg       0.86      0.86      0.86     10000


Classification Report - AlexNet:
              precision    recall  f1-score   support

    airplane       0.76      0.80      0.78      1000
  automobile       0.85      0.89      0.87      1000
        bird       0.64      0.65      0.65      1000
         cat       0.55      0.57      0.56      1000
        deer       0.71      0.70      0.71      1000
         dog       0.66      0.64      0.65      1000
        frog       0.83      0.81      0.82      1000
       horse       0.82      0.79      0.80      1000
        ship       0.87      0.86      0.86      1000
       truck       0.85      0.81      0.83      1000

    accuracy                           0.75     10000
   macro avg       0.75      0.75      0.75     10000
weighted avg       0.75      0.75      0.75     10000
```

In [14]:

```
# Plot confusion matrices
fig, axes = plt.subplots(1, 2, figsize=(16, 6))
```

```
# Custom CNN confusion matrix
cm_custom = confusion_matrix(true_classes, custom_cnn_pred_classes)
sns.heatmap(cm_custom, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names, ax=axes[0])
axes[0].set_title(f'Custom CNN Confusion Matrix\nTest Accuracy: {custom_cnn_test_accuracy:.3f}')
axes[0].set_xlabel('Predicted')
axes[0].set_ylabel('Actual')

# AlexNet confusion matrix
cm_alexnet = confusion_matrix(true_classes, alexnet_pred_classes)
sns.heatmap(cm_alexnet, annot=True, fmt='d', cmap='Reds',
            xticklabels=class_names, yticklabels=class_names, ax=axes[1])
axes[1].set_title(f'AlexNet Confusion Matrix\nTest Accuracy: {alexnet_test_accuracy:.3f}')
axes[1].set_xlabel('Predicted')
axes[1].set_ylabel('Actual')

plt.tight_layout()
plt.show()
```



Custom CNN Confusion Matrix (Test Accuracy: 0.857)

Actual \ Predicted	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	863	11	24	9	10	3	3	5	50	22
automobile	6	938	1	2	0	2	3	0	13	35
bird	41	0	791	30	46	35	33	15	8	1
cat	13	1	51	704	39	113	40	24	8	7
deer	9	2	38	29	862	9	24	25	2	0
dog	6	1	23	107	31	791	9	27	3	2
frog	4	3	27	30	19	11	893	7	4	2
horse	6	0	18	22	27	29	3	891	1	3
ship	26	7	9	6	0	1	1	4	931	15
truck	9	49	2	7	0	2	0	4	17	910

AlexNet Confusion Matrix (Test Accuracy: 0.753)

Actual \ Predicted	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	802	15	53	23	15	3	7	7	47	28
automobile	15	893	5	5	2	3	6	3	18	50
bird	65	4	651	57	80	62	40	23	10	8
cat	30	9	70	574	54	150	55	33	12	13
deer	18	3	81	70	699	34	30	52	10	3
dog	10	7	49	179	41	641	13	46	7	7
frog	5	3	47	70	24	19	815	6	5	6
horse	13	3	36	29	59	53	8	786	2	11
ship	67	24	10	14	2	5	3	1	860	14
truck	34	87	9	18	2	7	5	5	21	812

Model Complexity Analysis

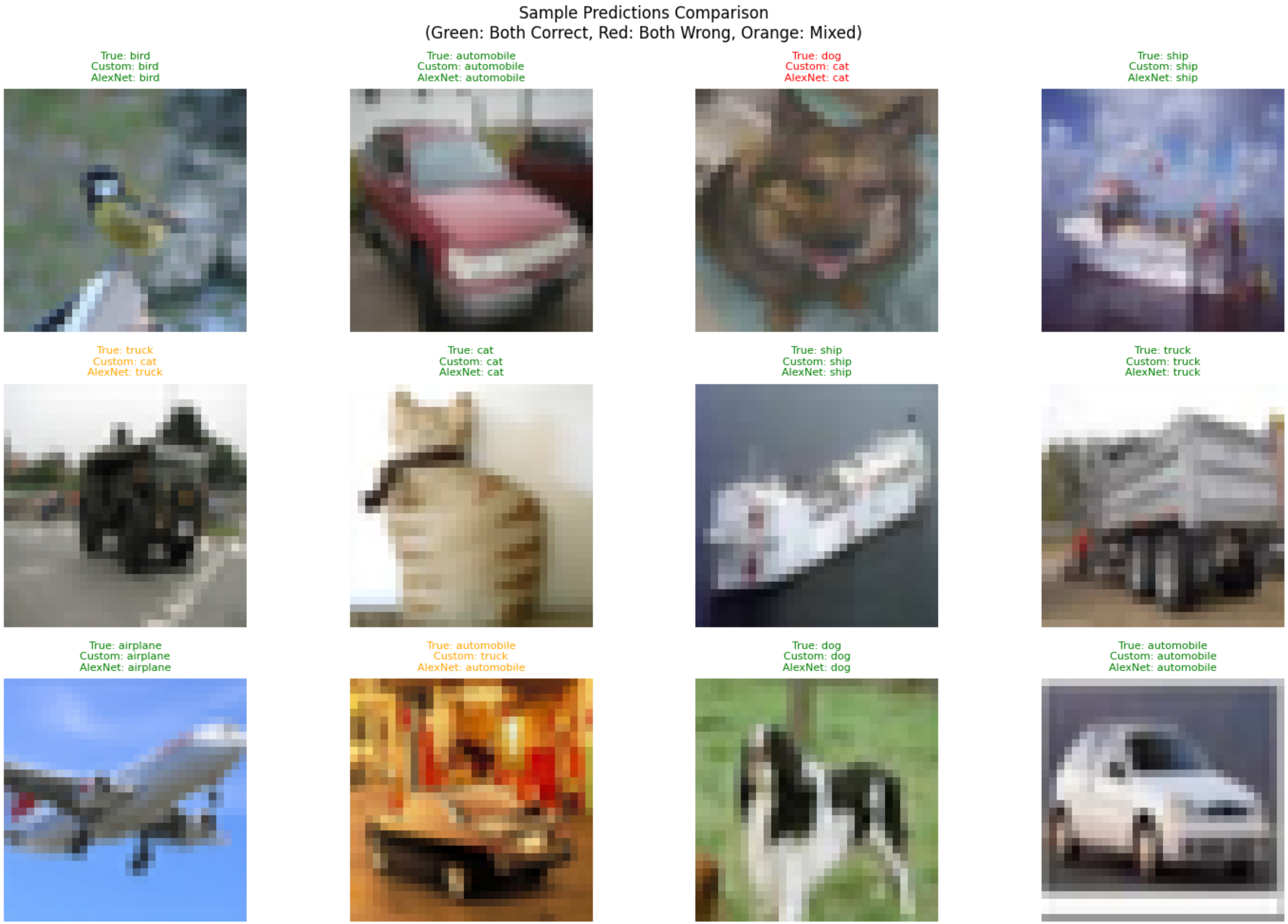
In [15]:
```
# Calculate model parameters and complexity
def count_parameters(model):
    return model.count_params()

custom_cnn_params = count_parameters(custom_cnn)
alexnet_params = count_parameters(alexnet_model)

print("Model Complexity Analysis:")
print(f"Custom CNN:")
print(f"  - Total Parameters: {custom_cnn_params:,}")
print(f"  - Training Time: {custom_cnn_training_time:.2f} seconds")
print(f"  - Test Accuracy: {custom_cnn_test_accuracy:.4f}")

print(f"\nAlexNet:")
print(f"  - Total Parameters: {alexnet_params:,}")
print(f"  - Training Time: {alexnet_training_time:.2f} seconds")
print(f"  - Test Accuracy: {alexnet_test_accuracy:.4f}")

print(f"\nParameter Ratio: AlexNet has {alexnet_params/custom_cnn_params:.1f}x more parameters than Custom CNN")
```

```
Model Complexity Analysis:
Custom CNN:
  - Total Parameters: 1,472,938
  - Training Time: 257.18 seconds
  - Test Accuracy: 0.8574

AlexNet:
  - Total Parameters: 37,324,554
  - Training Time: 892.38 seconds
  - Test Accuracy: 0.7533

Parameter Ratio: AlexNet has 25.3x more parameters than Custom CNN
```

Sample Predictions Visualization

In [16]:
```
# Visualize some predictions
```

```python
def plot_predictions(images, true_labels, custom_predictions, alexnet_predictions, class_names, num_samples=12):
    plt.figure(figsize=(15, 10))

    for i in range(num_samples):
        plt.subplot(3, 4, i + 1)
        plt.imshow(images[i])

        true_class = class_names[true_labels[i]]
        custom_pred_class = class_names[custom_predictions[i]]
        alexnet_pred_class = class_names[alexnet_predictions[i]]

        # Color code: green if both correct, red if both wrong, yellow if mixed
        custom_correct = custom_predictions[i] == true_labels[i]
        alexnet_correct = alexnet_predictions[i] == true_labels[i]

        if custom_correct and alexnet_correct:
            color = 'green'
        elif not custom_correct and not alexnet_correct:
            color = 'red'
        else:
            color = 'orange'

        plt.title(f'True: {true_class}\nCustom: {custom_pred_class}\nAlexNet: {alexnet_pred_class}',
                  fontsize=8, color=color)
        plt.axis('off')

    plt.suptitle('Sample Predictions Comparison\n(Green: Both Correct, Red: Both Wrong, Orange: Mixed)',
                 fontsize=12)
    plt.tight_layout()
    plt.show()

# Show some random predictions
random_indices = np.random.choice(len(x_test), 12, replace=False)
plot_predictions(x_test[random_indices], true_classes[random_indices],
                 custom_cnn_pred_classes[random_indices], alexnet_pred_classes[random_indices],
                 class_names)
```


Sample Predictions Comparison
(Green: Both Correct, Red: Both Wrong, Orange: Mixed)
True: bird
Custom: bird
AlexNet: bird
True: automobile
Custom: automobile
AlexNet: automobile
True: dog
Custom: cat
AlexNet: cat
True: ship
Custom: ship
AlexNet: ship
True: truck
Custom: cat
AlexNet: truck
True: cat
Custom: cat
AlexNet: cat
True: ship
Custom: ship
AlexNet: ship
True: truck
Custom: truck
AlexNet: truck
True: airplane
Custom: airplane
AlexNet: airplane
True: automobile
Custom: truck
AlexNet: automobile
True: dog
Custom: dog
AlexNet: dog
True: automobile
Custom: automobile
AlexNet: automobile

Final Performance Summary and Analysis

In [17]:

```
print("="*80)
print("CIFAR-10 CLASSIFICATION - FINAL PERFORMANCE SUMMARY")
print("="*80)

print(f"\n📊 TEST ACCURACY RESULTS:")
print(f"Custom CNN:  {custom_cnn_test_accuracy:.4f} ({custom_cnn_test_accuracy*100:.2f}%)")
print(f"AlexNet:     {alexnet_test_accuracy:.4f} ({alexnet_test_accuracy*100:.2f}%)")

if custom_cnn_test_accuracy > alexnet_test_accuracy:
    winner = "Custom CNN"
    difference = custom_cnn_test_accuracy - alexnet_test_accuracy
else:
    winner = "AlexNet"
    difference = alexnet_test_accuracy - custom_cnn_test_accuracy

print(f"\n🏆 WINNER: {winner} (by {difference*100:.2f} percentage points)")

print(f"\n⚙️ MODEL COMPLEXITY:")
print(f"Custom CNN Parameters:  {custom_cnn_params:,}")
print(f"AlexNet Parameters:     {alexnet_params:,}")
print(f"Parameter Ratio:        {alexnet_params/custom_cnn_params:.1f}x")

print(f"\n⏱️ TRAINING TIME:")
print(f"Custom CNN:  {custom_cnn_training_time:.2f} seconds")
print(f"AlexNet:     {alexnet_training_time:.2f} seconds")

print(f"\n📈 OBSERVATIONS AND ANALYSIS:")
print(f"\n1. ACCURACY COMPARISON:")
if custom_cnn_test_accuracy > alexnet_test_accuracy:
    print(f"   • Custom CNN outperformed AlexNet by {difference*100:.2f}%")
    print(f"   • This suggests that the custom architecture is better suited for CIFAR-10")
else:
    print(f"   • AlexNet outperformed Custom CNN by {difference*100:.2f}%")
    print(f"   • This demonstrates the power of the proven AlexNet architecture")

print(f"\n2. MODEL EFFICIENCY:")
custom_efficiency = custom_cnn_test_accuracy / (custom_cnn_params / 1000000)
```

```python
alexnet_efficiency = alexnet_test_accuracy / (alexnet_params / 1000000)
print(f"   • Custom CNN Efficiency: {custom_efficiency:.3f} (accuracy per million parameters)")
print(f"   • AlexNet Efficiency: {alexnet_efficiency:.3f} (accuracy per million parameters)")

if custom_efficiency > alexnet_efficiency:
    print(f"   • Custom CNN is more parameter-efficient")
else:
    print(f"   • AlexNet achieves better accuracy despite having more parameters")

print(f"\n3. ARCHITECTURE INSIGHTS:")
print(f"   • Custom CNN uses modern techniques: Batch Normalization, Dropout")
print(f"   • AlexNet adapted from 224x224 to 32x32 input size")
print(f"   • Both models benefit from data normalization and proper regularization")

print(f"\n4. PRACTICAL CONSIDERATIONS:")
if custom_cnn_training_time < alexnet_training_time:
    print(f"   • Custom CNN trains {alexnet_training_time/custom_cnn_training_time:.1f}x faster")
else:
    print(f"   • AlexNet trains {custom_cnn_training_time/alexnet_training_time:.1f}x faster")
print(f"   • Model size affects deployment and inference speed")
print(f"   • Custom architectures can be optimized for specific datasets")

print(f"\n" + "="*80)
print(f"CONCLUSION: The {winner} model provides the best performance for CIFAR-10 classification")
print(f"in terms of test accuracy, achieving {max(custom_cnn_test_accuracy, alexnet_test_accuracy)*100:.2f}% accura
print("="*80)
```

```
================================================================================
CIFAR-10 CLASSIFICATION - FINAL PERFORMANCE SUMMARY
================================================================================

📊 TEST ACCURACY RESULTS:
Custom CNN:  0.8574 (85.74%)
AlexNet:     0.7533 (75.33%)

🏆 WINNER: Custom CNN (by 10.41 percentage points)

⚙️ MODEL COMPLEXITY:
Custom CNN Parameters:  1,472,938
AlexNet Parameters:     37,324,554
Parameter Ratio:        25.3x

⏱️ TRAINING TIME:
Custom CNN:  257.18 seconds
AlexNet:     892.38 seconds

📈 OBSERVATIONS AND ANALYSIS:

1. ACCURACY COMPARISON:
   • Custom CNN outperformed AlexNet by 10.41%
   • This suggests that the custom architecture is better suited for CIFAR-10

2. MODEL EFFICIENCY:
   • Custom CNN Efficiency: 0.582 (accuracy per million parameters)
   • AlexNet Efficiency: 0.020 (accuracy per million parameters)
   • Custom CNN is more parameter-efficient

3. ARCHITECTURE INSIGHTS:
   • Custom CNN uses modern techniques: Batch Normalization, Dropout
   • AlexNet adapted from 224x224 to 32x32 input size
   • Both models benefit from data normalization and proper regularization

4. PRACTICAL CONSIDERATIONS:
   • Custom CNN trains 3.5x faster
   • Model size affects deployment and inference speed
   • Custom architectures can be optimized for specific datasets

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

CONCLUSION: The Custom CNN model provides the best performance for CIFAR-10 classification in terms of test accuracy, achieving 85.74% accuracy.
==