🔬 CNN Layer-by-Layer Visualization

This notebook helps you understand what happens in each layer of the CNN by:

1. Visualizing layer outputs (feature maps/activations)
2. Examining learned filters/kernels
3. Analyzing how images transform through the network
4. Understanding the hierarchical feature learning

Setup

In [1]:
```
import sys
import os

# Add project root to Python path
project_root = os.path.abspath(os.path.join(os.getcwd(), '../../..'))
if project_root not in sys.path:
    sys.path.insert(0, project_root)

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models, datasets, utils, Model
import tensorflow as tf

print("TensorFlow version:", tf.__version__)
```

```
TensorFlow version: 2.20.0
```

1. Load Data and Model

First, we'll rebuild the CNN model from the main notebook.

In [2]:
```
# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()
```

```
# Normalize
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

NUM_CLASSES = 10
y_train = utils.to_categorical(y_train, NUM_CLASSES)
y_test = utils.to_categorical(y_test, NUM_CLASSES)

CLASSES = np.array([
    "airplane", "automobile", "bird", "cat", "deer",
    "dog", "frog", "horse", "ship", "truck"
])

print(f"Training data shape: {x_train.shape}")
print(f"Test data shape: {x_test.shape}")
```

```
Training data shape: (50000, 32, 32, 3)
Test data shape: (10000, 32, 32, 3)
```

In [3]:
```
# Rebuild the CNN model (same architecture as cnn.ipynb)
input_layer = layers.Input((32, 32, 3), name="input")

x = layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same", name="conv1")(input_layer)
x = layers.BatchNormalization(name="bn1")(x)
x = layers.LeakyReLU(name="leaky_relu1")(x)

x = layers.Conv2D(filters=32, kernel_size=3, strides=2, padding="same", name="conv2")(x)
x = layers.BatchNormalization(name="bn2")(x)
x = layers.LeakyReLU(name="leaky_relu2")(x)

x = layers.Conv2D(filters=64, kernel_size=3, strides=1, padding="same", name="conv3")(x)
x = layers.BatchNormalization(name="bn3")(x)
x = layers.LeakyReLU(name="leaky_relu3")(x)

x = layers.Conv2D(filters=64, kernel_size=3, strides=2, padding="same", name="conv4")(x)
x = layers.BatchNormalization(name="bn4")(x)
x = layers.LeakyReLU(name="leaky_relu4")(x)

x = layers.Flatten(name="flatten")(x)

x = layers.Dense(128, name="dense1")(x)
```

```
x = layers.BatchNormalization(name="bn5")(x)
x = layers.LeakyReLU(name="leaky_relu5")(x)
x = layers.Dropout(rate=0.5, name="dropout")(x)

x = layers.Dense(NUM_CLASSES, name="dense2")(x)
output_layer = layers.Activation("softmax", name="softmax")(x)

model = models.Model(input_layer, output_layer, name="CNN_Classifier")

model.summary()
```

Model: "CNN_Classifier"

Layer (type)	Output Shape	Param #
input (InputLayer)	(None, 32, 32, 3)	0
conv1 (Conv2D)	(None, 32, 32, 32)	896
bn1 (BatchNormalization)	(None, 32, 32, 32)	128
leaky_relu1 (LeakyReLU)	(None, 32, 32, 32)	0
conv2 (Conv2D)	(None, 16, 16, 32)	9,248
bn2 (BatchNormalization)	(None, 16, 16, 32)	128
leaky_relu2 (LeakyReLU)	(None, 16, 16, 32)	0
conv3 (Conv2D)	(None, 16, 16, 64)	18,496
bn3 (BatchNormalization)	(None, 16, 16, 64)	256
leaky_relu3 (LeakyReLU)	(None, 16, 16, 64)	0
conv4 (Conv2D)	(None, 8, 8, 64)	36,928
bn4 (BatchNormalization)	(None, 8, 8, 64)	256
leaky_relu4 (LeakyReLU)	(None, 8, 8, 64)	0
flatten (Flatten)	(None, 4096)	0
dense1 (Dense)	(None, 128)	524,416
bn5 (BatchNormalization)	(None, 128)	512
leaky_relu5 (LeakyReLU)	(None, 128)	0
dropout (Dropout)	(None, 128)	0
dense2 (Dense)	(None, 10)	1,290
softmax (Activation)	(None, 10)	0

Total params: 592,554 (2.26 MB)

```
Trainable params: 591,914 (2.26 MB)
Non-trainable params: 640 (2.50 KB)
```

2. Train the Model (or Load Pre-trained)

You can either train the model or load a pre-trained one if you already trained it.

In [4]:
```
# Option 1: Train the model (quick training for visualization purposes)
TRAIN_MODEL = True  # Set to False if you want to load pre-trained weights

if TRAIN_MODEL:
    from tensorflow.keras import optimizers

    opt = optimizers.Adam(learning_rate=0.0005)
    model.compile(
        loss="categorical_crossentropy",
        optimizer=opt,
        metrics=["accuracy"]
    )

    # Train for just 3 epochs for quick visualization
    # (Use more epochs for better accuracy)
    history = model.fit(
        x_train,
        y_train,
        batch_size=32,
        epochs=3,
        validation_split=0.1,
        verbose=1
    )

    # Save the model
    model.save('cnn_model.h5')
    print("\n✓ Model trained and saved!")
else:
    # Option 2: Load pre-trained model
    try:
        model = models.load_model('cnn_model.h5')
        print("✓ Model loaded!")
```

```
    except:
        print("❌ No saved model found. Set TRAIN_MODEL=True to train.")
```

```
Epoch 1/3
1407/1407 ━━━━━━━━━━━━━━━━━━━━ 21s 14ms/step - accuracy: 0.4510 - loss: 1.5616 - val_accuracy: 0.4890 - val
_loss: 1.4948
Epoch 2/3
1407/1407 ━━━━━━━━━━━━━━━━━━━━ 18s 13ms/step - accuracy: 0.5861 - loss: 1.1662 - val_accuracy: 0.5622 - val
_loss: 1.3000
Epoch 3/3
1407/1407 ━━━━━━━━━━━━━━━━━━━━ 18s 13ms/step - accuracy: 0.6427 - loss: 1.0166 - val_accuracy: 0.6616 - val
_loss: 0.9721
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(mode
l)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `mode
l.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
✓ Model trained and saved!
```

3. Select a Sample Image

Let's pick an image to analyze through the network.

In [5]:
```
# Select a test image
img_index = 15  # Change this to visualize different images

sample_image = x_test[img_index]
true_label = CLASSES[np.argmax(y_test[img_index])]

# Get prediction
prediction = model.predict(np.expand_dims(sample_image, axis=0), verbose=0)
predicted_label = CLASSES[np.argmax(prediction)]
confidence = np.max(prediction) * 100

# Display the image
plt.figure(figsize=(5, 5))
plt.imshow(sample_image)
plt.title(f"True: {true_label}\nPredicted: {predicted_label} ({confidence:.1f}%)")
plt.axis('off')
plt.show()
```

```
print(f"Image shape: {sample_image.shape}")
print(f"Pixel value range: [{sample_image.min():.3f}, {sample_image.max():.3f}]")
```


```
Image shape: (32, 32, 3)
Pixel value range: [0.000, 0.992]
```

4. Visualize Layer Outputs (Feature Maps)

This shows what each layer "sees" when processing the image.

In [6]:
```
# Get all layer names
layer_names = [layer.name for layer in model.layers]
print("Layers in the model:")
```

```
for i, name in enumerate(layer_names):
    print(f"{i:2d}. {name:15s} - {model.layers[i].__class__.__name__}")
```

```
Layers in the model:
 0. input           - InputLayer
 1. conv1           - Conv2D
 2. bn1             - BatchNormalization
 3. leaky_relu1     - LeakyReLU
 4. conv2           - Conv2D
 5. bn2             - BatchNormalization
 6. leaky_relu2     - LeakyReLU
 7. conv3           - Conv2D
 8. bn3             - BatchNormalization
 9. leaky_relu3     - LeakyReLU
10. conv4           - Conv2D
11. bn4             - BatchNormalization
12. leaky_relu4     - LeakyReLU
13. flatten         - Flatten
14. dense1          - Dense
15. bn5             - BatchNormalization
16. leaky_relu5     - LeakyReLU
17. dropout         - Dropout
18. dense2          - Dense
19. softmax         - Activation
```

In [7]:
```
# Create models that output intermediate layer activations
layer_outputs = [layer.output for layer in model.layers]
activation_model = Model(inputs=model.input, outputs=layer_outputs)

# Get activations for our sample image
activations = activation_model.predict(np.expand_dims(sample_image, axis=0), verbose=0)

print(f"Number of layer outputs: {len(activations)}")
print(f"\nShape of each layer output:")
for i, (name, activation) in enumerate(zip(layer_names, activations)):
    print(f"{i:2d}. {name:15s}: {activation.shape}")
```

```
Number of layer outputs: 20

Shape of each layer output:
 0. input          : (1, 32, 32, 3)
 1. conv1          : (1, 32, 32, 32)
 2. bn1            : (1, 32, 32, 32)
 3. leaky_relu1    : (1, 32, 32, 32)
 4. conv2          : (1, 16, 16, 32)
 5. bn2            : (1, 16, 16, 32)
 6. leaky_relu2    : (1, 16, 16, 32)
 7. conv3          : (1, 16, 16, 64)
 8. bn3            : (1, 16, 16, 64)
 9. leaky_relu3    : (1, 16, 16, 64)
10. conv4          : (1, 8, 8, 64)
11. bn4            : (1, 8, 8, 64)
12. leaky_relu4    : (1, 8, 8, 64)
13. flatten        : (1, 4096)
14. dense1         : (1, 128)
15. bn5            : (1, 128)
16. leaky_relu5    : (1, 128)
17. dropout        : (1, 128)
18. dense2         : (1, 10)
19. softmax        : (1, 10)
```

4.1 Visualize Convolutional Layer Outputs

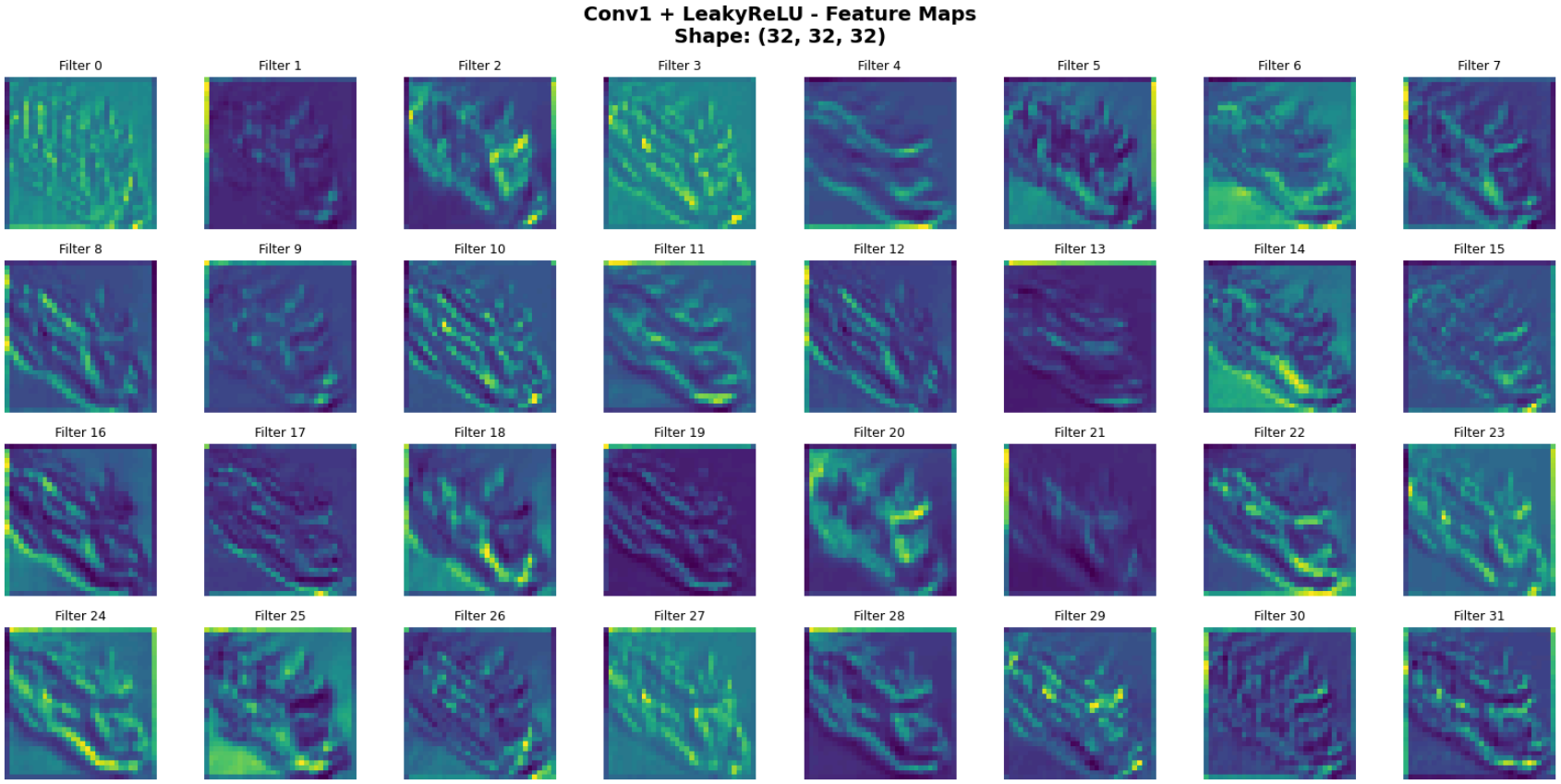
In [8]:

```python
def visualize_conv_layer(activation, layer_name, n_features=32, cols=8):
    """
    Visualize feature maps from a convolutional layer.

    Args:
        activation: Output from the layer (shape: batch, height, width, channels)
        layer_name: Name of the layer
        n_features: Number of feature maps to display
        cols: Number of columns in the grid
    """
    # Remove batch dimension
    if len(activation.shape) == 4:
        activation = activation[0]

    n_features = min(n_features, activation.shape[-1])
```

```
    rows = int(np.ceil(n_features / cols))

    fig, axes = plt.subplots(rows, cols, figsize=(cols * 2, rows * 2))
    fig.suptitle(f'{layer_name} - Feature Maps\nShape: {activation.shape}',
                 fontsize=14, weight='bold')

    axes = axes.flatten() if n_features > 1 else [axes]

    for i in range(rows * cols):
        if i < n_features:
            axes[i].imshow(activation[:, :, i], cmap='viridis')
            axes[i].set_title(f'Filter {i}', fontsize=9)
        axes[i].axis('off')

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

Conv Layer 1 Output (after LeakyReLU)

In [9]:
```
# Visualize first conv layer output (after activation)
conv1_output = activations[layer_names.index('leaky_relu1')]
visualize_conv_layer(conv1_output, 'Conv1 + LeakyReLU', n_features=32, cols=8)
```

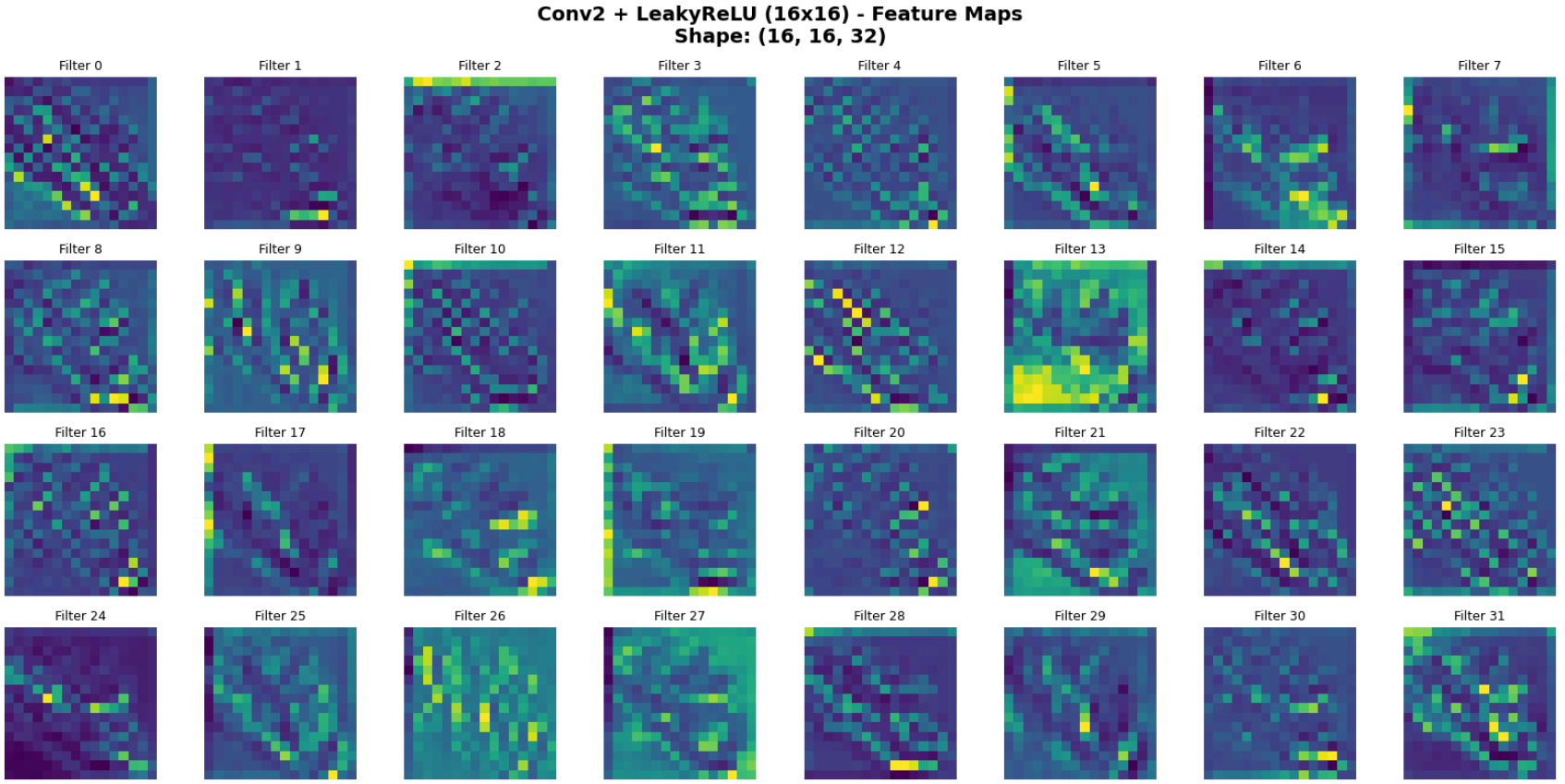


What you're seeing:

- First convolutional layer detects **low-level features** like edges, colors, and simple textures
- Each feature map responds to different patterns
- Bright areas = strong activation (feature detected)
- Dark areas = weak activation (feature not present)

Conv Layer 2 Output (after pooling/stride)

In [10]:
```
# Visualize second conv layer (after stride 2 - spatial reduction)
conv2_output = activations[layer_names.index('leaky_relu2')]
visualize_conv_layer(conv2_output, 'Conv2 + LeakyReLU (16x16)', n_features=32, cols=8)
```

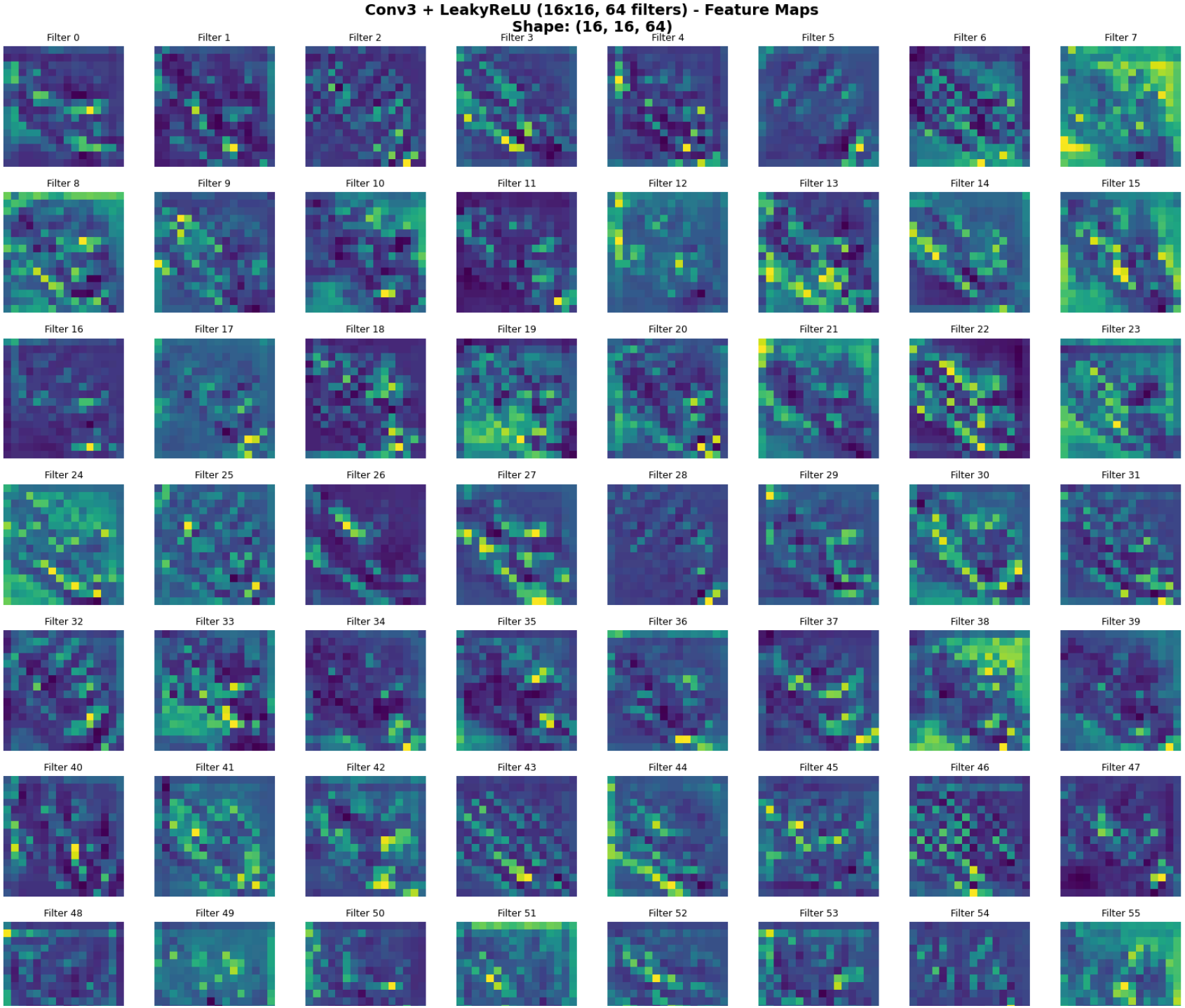


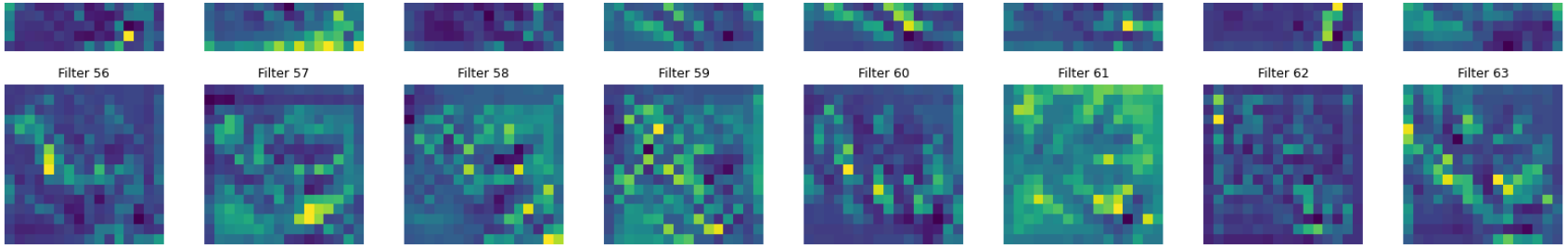
What you're seeing:

- Image is now 16×16 (downsampled from 32×32)
- Detects **mid-level features** like corners, simple shapes
- More abstract than layer 1

Conv Layer 3 Output

In [11]:
```
# Visualize third conv layer (64 filters)
conv3_output = activations[layer_names.index('leaky_relu3')]
visualize_conv_layer(conv3_output, 'Conv3 + LeakyReLU (16x16, 64 filters)', n_features=64, cols=8)
```


Conv3 + LeakyReLU (16x16, 64 filters) - Feature Maps
Shape: (16, 16, 64)
Filter 0
Filter 1
Filter 2
Filter 3
Filter 4
Filter 5
Filter 6
Filter 7
Filter 8
Filter 9
Filter 10
Filter 11
Filter 12
Filter 13
Filter 14
Filter 15
Filter 16
Filter 17
Filter 18
Filter 19
Filter 20
Filter 21
Filter 22
Filter 23
Filter 24
Filter 25
Filter 26
Filter 27
Filter 28
Filter 29
Filter 30
Filter 31
Filter 32
Filter 33
Filter 34
Filter 35
Filter 36
Filter 37
Filter 38
Filter 39
Filter 40
Filter 41
Filter 42
Filter 43
Filter 44
Filter 45
Filter 46
Filter 47
Filter 48
Filter 49
Filter 50
Filter 51
Filter 52
Filter 53
Filter 54
Filter 55

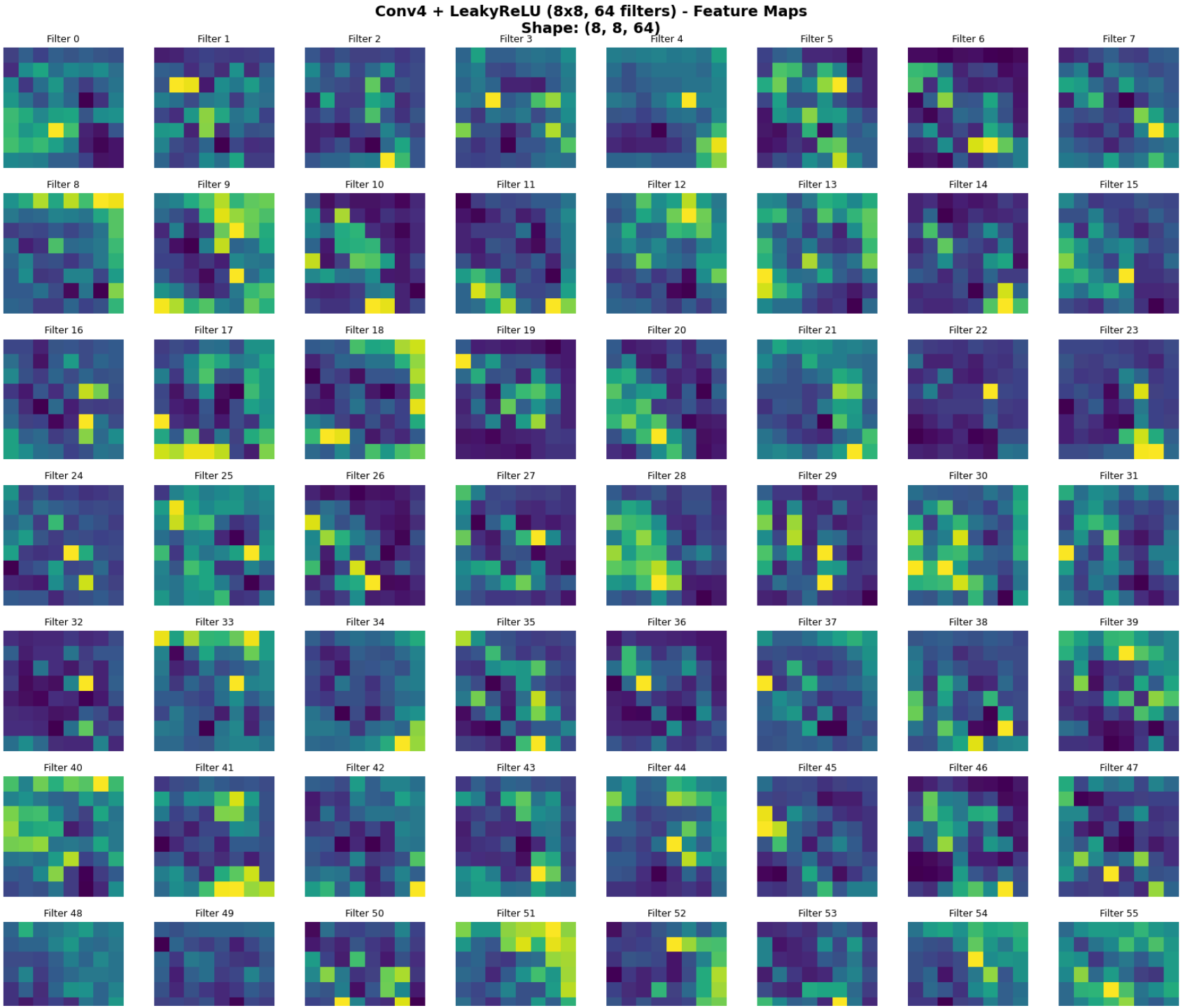


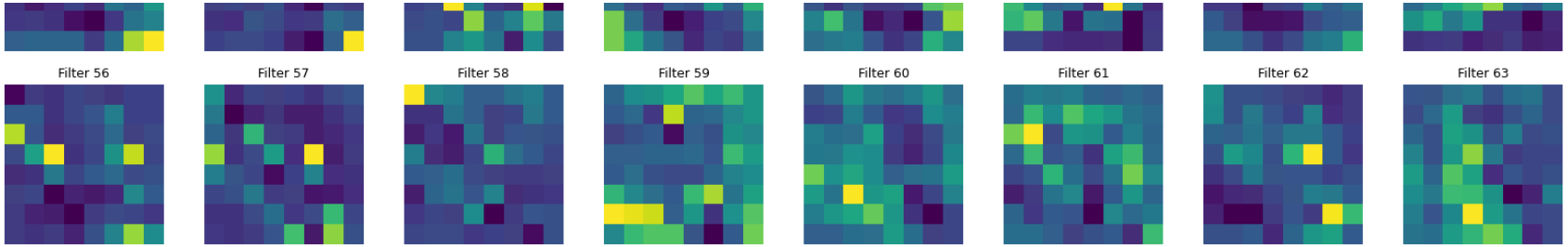
What you're seeing:

- 64 filters now (more capacity to detect patterns)
- Detects **complex patterns** and object parts
- Starting to recognize object-specific features

Conv Layer 4 Output (deepest conv layer)

In [12]:
```
# Visualize fourth conv layer (8x8 spatial size)
conv4_output = activations[layer_names.index('leaky_relu4')]
visualize_conv_layer(conv4_output, 'Conv4 + LeakyReLU (8x8, 64 filters)', n_features=64, cols=8)
```


Conv4 + LeakyReLU (8x8, 64 filters) - Feature Maps
Shape: (8, 8, 64)
Filter 0
Filter 1
Filter 2
Filter 3
Filter 4
Filter 5
Filter 6
Filter 7
Filter 8
Filter 9
Filter 10
Filter 11
Filter 12
Filter 13
Filter 14
Filter 15
Filter 16
Filter 17
Filter 18
Filter 19
Filter 20
Filter 21
Filter 22
Filter 23
Filter 24
Filter 25
Filter 26
Filter 27
Filter 28
Filter 29
Filter 30
Filter 31
Filter 32
Filter 33
Filter 34
Filter 35
Filter 36
Filter 37
Filter 38
Filter 39
Filter 40
Filter 41
Filter 42
Filter 43
Filter 44
Filter 45
Filter 46
Filter 47
Filter 48
Filter 49
Filter 50
Filter 51
Filter 52
Filter 53
Filter 54
Filter 55



What you're seeing:

- Highly abstract representations (8×8 spatial size)
- Detects **high-level semantic features**
- These features are used for classification
- Individual feature maps may respond to specific object parts or combinations

5. Visualize Learned Filters (Kernels)

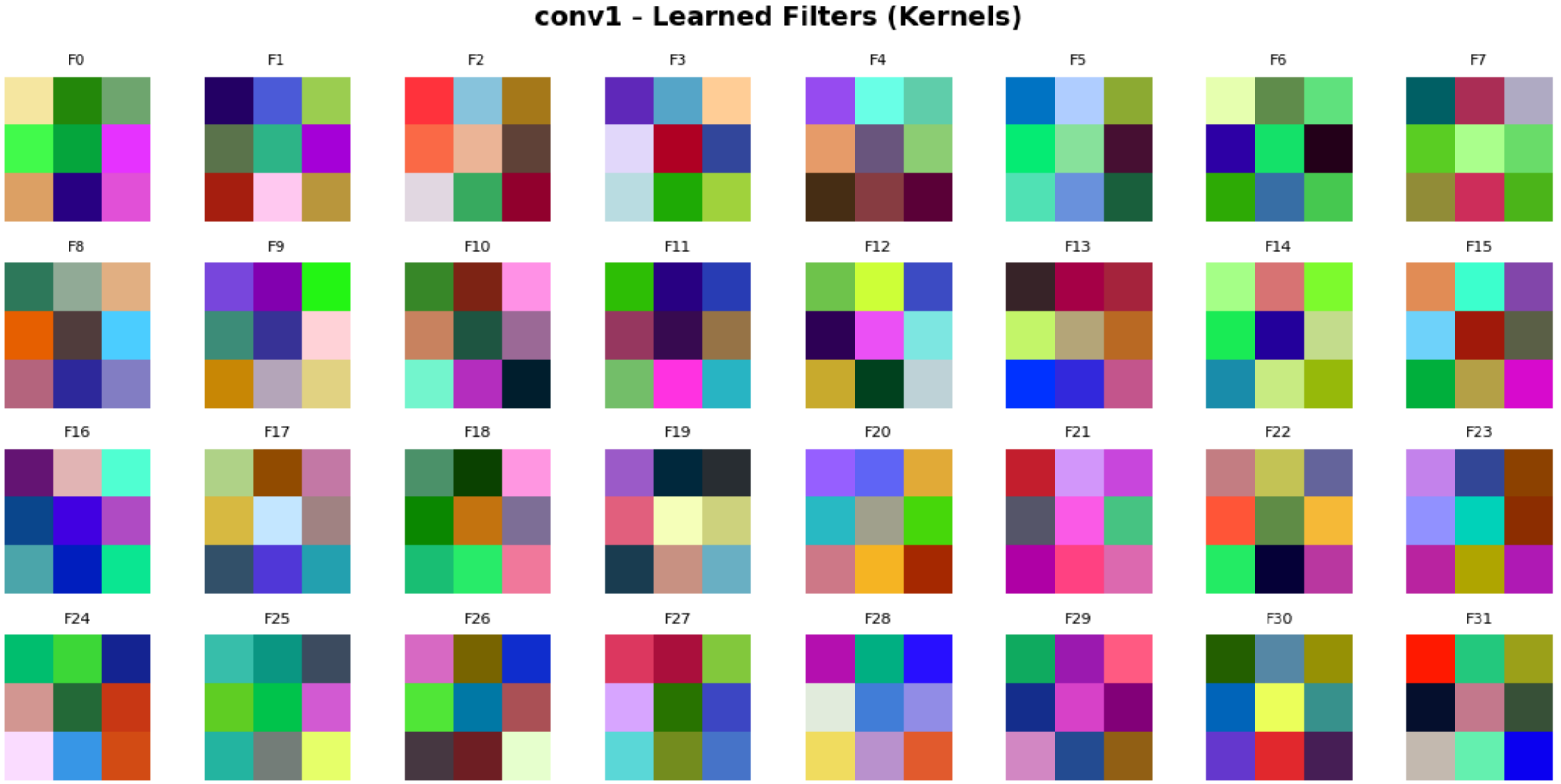
Let's look at what patterns the convolutional filters have learned.

In [13]:
```
def visualize_filters(layer_name, n_filters=32, normalize=True):
    """
    Visualize the learned convolutional filters.

    Args:
        layer_name: Name of the convolutional layer
        n_filters: Number of filters to display
        normalize: Whether to normalize filter values for visualization
    """
    layer = model.get_layer(layer_name)
    filters = layer.get_weights()[0]  # Shape: (height, width, input_channels, output_channels)

    print(f"Layer: {layer_name}")
    print(f"Filter shape: {filters.shape}")
    print(f"  - Kernel size: {filters.shape[0]}×{filters.shape[1]}")
    print(f"  - Input channels: {filters.shape[2]}")
    print(f"  - Output filters: {filters.shape[3]}")
    print()
```

```
    n_filters = min(n_filters, filters.shape[3])
    cols = 8
    rows = int(np.ceil(n_filters / cols))

    fig, axes = plt.subplots(rows, cols, figsize=(cols * 1.5, rows * 1.5))
    fig.suptitle(f'{layer_name} - Learned Filters (Kernels)', fontsize=14, weight='bold')

    axes = axes.flatten() if n_filters > 1 else [axes]

    for i in range(rows * cols):
        if i < n_filters:
            # Get filter for channel i
            f = filters[:, :, :, i]

            # If RGB input (3 channels), visualize as RGB
            if f.shape[2] == 3:
                # Normalize to [0, 1] for RGB display
                if normalize:
                    f_min, f_max = f.min(), f.max()
                    if f_max > f_min:
                        f = (f - f_min) / (f_max - f_min)
                axes[i].imshow(f)
            else:
                # For grayscale or multi-channel, show first channel
                f_channel = f[:, :, 0]
                if normalize:
                    f_min, f_max = f_channel.min(), f_channel.max()
                    if f_max > f_min:
                        f_channel = (f_channel - f_min) / (f_max - f_min)
                axes[i].imshow(f_channel, cmap='gray')

            axes[i].set_title(f'F{i}', fontsize=8)
        axes[i].axis('off')

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

In [14]:
```
# Visualize filters from Conv1 (operates on RGB input)
visualize_filters('conv1', n_filters=32)
```

```
Layer: conv1
Filter shape: (3, 3, 3, 32)
  - Kernel size: 3×3
  - Input channels: 3
  - Output filters: 32
```

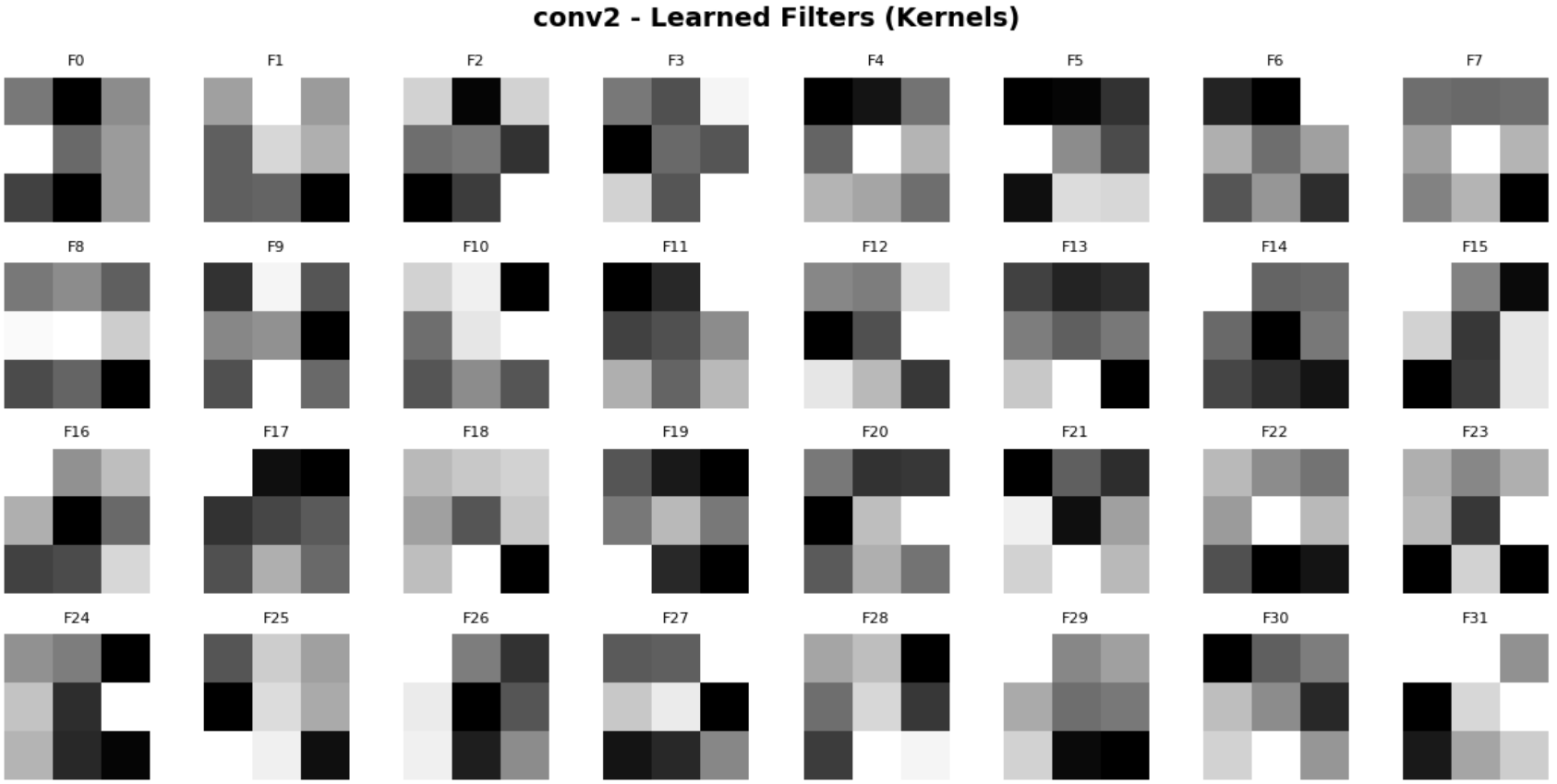


What you're seeing:

- These are 3×3 filters learned by the first convolutional layer
- Each filter detects a specific pattern (edges, colors, textures)
- RGB visualization shows color sensitivities
- Different filters specialize in different features

In [15]:
```
# Visualize filters from Conv2
visualize_filters('conv2', n_filters=32)
```

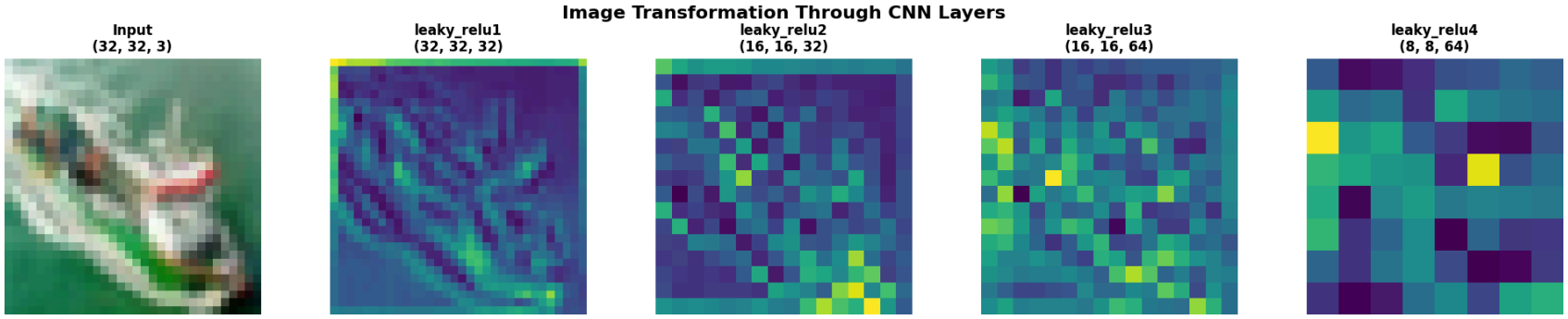
```
Layer: conv2
Filter shape: (3, 3, 32, 32)
  - Kernel size: 3×3
  - Input channels: 32
  - Output filters: 32
```



6. Layer-by-Layer Transformation

Let's see how the image transforms at each major stage.

In [16]:
```
# Select key layers to visualize
key_layers = ['input', 'leaky_relu1', 'leaky_relu2', 'leaky_relu3', 'leaky_relu4']

fig, axes = plt.subplots(1, len(key_layers), figsize=(20, 4))
fig.suptitle('Image Transformation Through CNN Layers', fontsize=16, weight='bold')

for idx, layer_name in enumerate(key_layers):
    if layer_name == 'input':
        # Show original image
        axes[idx].imshow(sample_image)
        axes[idx].set_title(f'Input\n{sample_image.shape}', fontsize=12, weight='bold')
    else:
        # Show average across all feature maps
        layer_idx = layer_names.index(layer_name)
        activation = activations[layer_idx][0]  # Remove batch dimension

        # Average across channels
        avg_activation = np.mean(activation, axis=-1)

        axes[idx].imshow(avg_activation, cmap='viridis')
        axes[idx].set_title(f'{layer_name}\n{activation.shape}', fontsize=12, weight='bold')

    axes[idx].axis('off')

plt.tight_layout()
plt.show()

print("\n📊 Spatial Size Progression:")
print("Input:  32×32 → Conv1: 32×32 → Conv2: 16×16 → Conv3: 16×16 → Conv4: 8×8")
print("\n📊 Channel Progression:")
print("Input:  3 channels → Conv1: 32 → Conv2: 32 → Conv3: 64 → Conv4: 64")
```

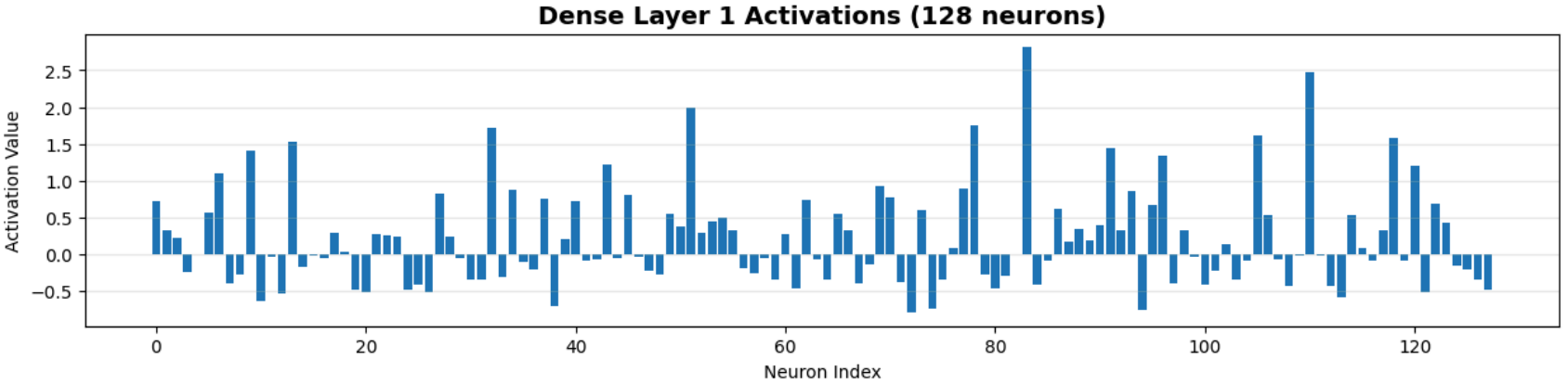


📊 Spatial Size Progression:
Input: 32×32 → Conv1: 32×32 → Conv2: 16×16 → Conv3: 16×16 → Conv4: 8×8

📊 Channel Progression:
Input: 3 channels → Conv1: 32 → Conv2: 32 → Conv3: 64 → Conv4: 64

7. Analyze Dense Layers

In [17]:
```
# Visualize Dense layer activations
dense1_output = activations[layer_names.index('leaky_relu5')][0]  # Shape: (128,)

plt.figure(figsize=(15, 3))
plt.bar(range(128), dense1_output)
plt.title('Dense Layer 1 Activations (128 neurons)', fontsize=14, weight='bold')
plt.xlabel('Neuron Index')
plt.ylabel('Activation Value')
plt.grid(axis='y', alpha=0.3)
plt.show()

print(f"Dense layer statistics:")
print(f"  Min: {dense1_output.min():.4f}")
print(f"  Max: {dense1_output.max():.4f}")
print(f"  Mean: {dense1_output.mean():.4f}")
print(f"  Std: {dense1_output.std():.4f}")
print(f"  Active neurons (>0.1): {(dense1_output > 0.1).sum()}/128")
```

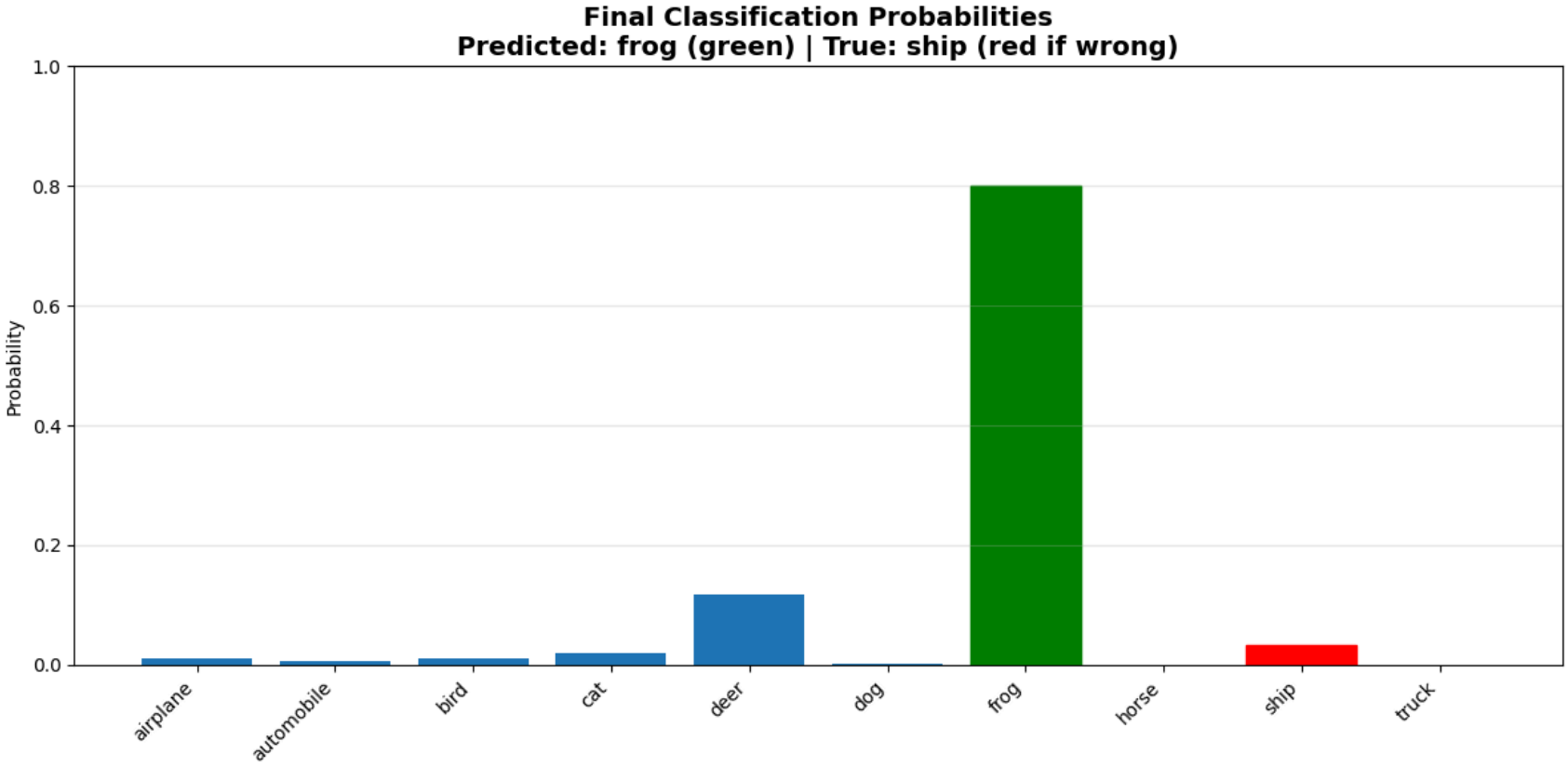



```
Dense layer statistics:
  Min: -0.7972
  Max: 2.8139
  Mean: 0.1984
  Std: 0.6721
  Active neurons (>0.1): 58/128
```

8. Final Classification Output

In [18]:
```python
# Visualize final softmax probabilities
final_output = activations[-1][0]  # Shape: (10,)

plt.figure(figsize=(12, 6))
bars = plt.bar(range(10), final_output)

# Color the predicted class
predicted_idx = np.argmax(final_output)
bars[predicted_idx].set_color('green')

# Color the true class (if different)
true_idx = np.argmax(y_test[img_index])
if true_idx != predicted_idx:
    bars[true_idx].set_color('red')

plt.xticks(range(10), CLASSES, rotation=45, ha='right')
plt.ylabel('Probability')
plt.title(f'Final Classification Probabilities\nPredicted: {predicted_label} (green) | True: {true_label}
          fontsize=14, weight='bold')
```

```
plt.grid(axis='y', alpha=0.3)
plt.ylim(0, 1)
plt.tight_layout()
plt.show()

print("\nClass probabilities:")
for i, (cls, prob) in enumerate(zip(CLASSES, final_output)):
    marker = "👉" if i == predicted_idx else "  "
    print(f"{marker} {cls:12s}: {prob*100:5.2f}%")
```



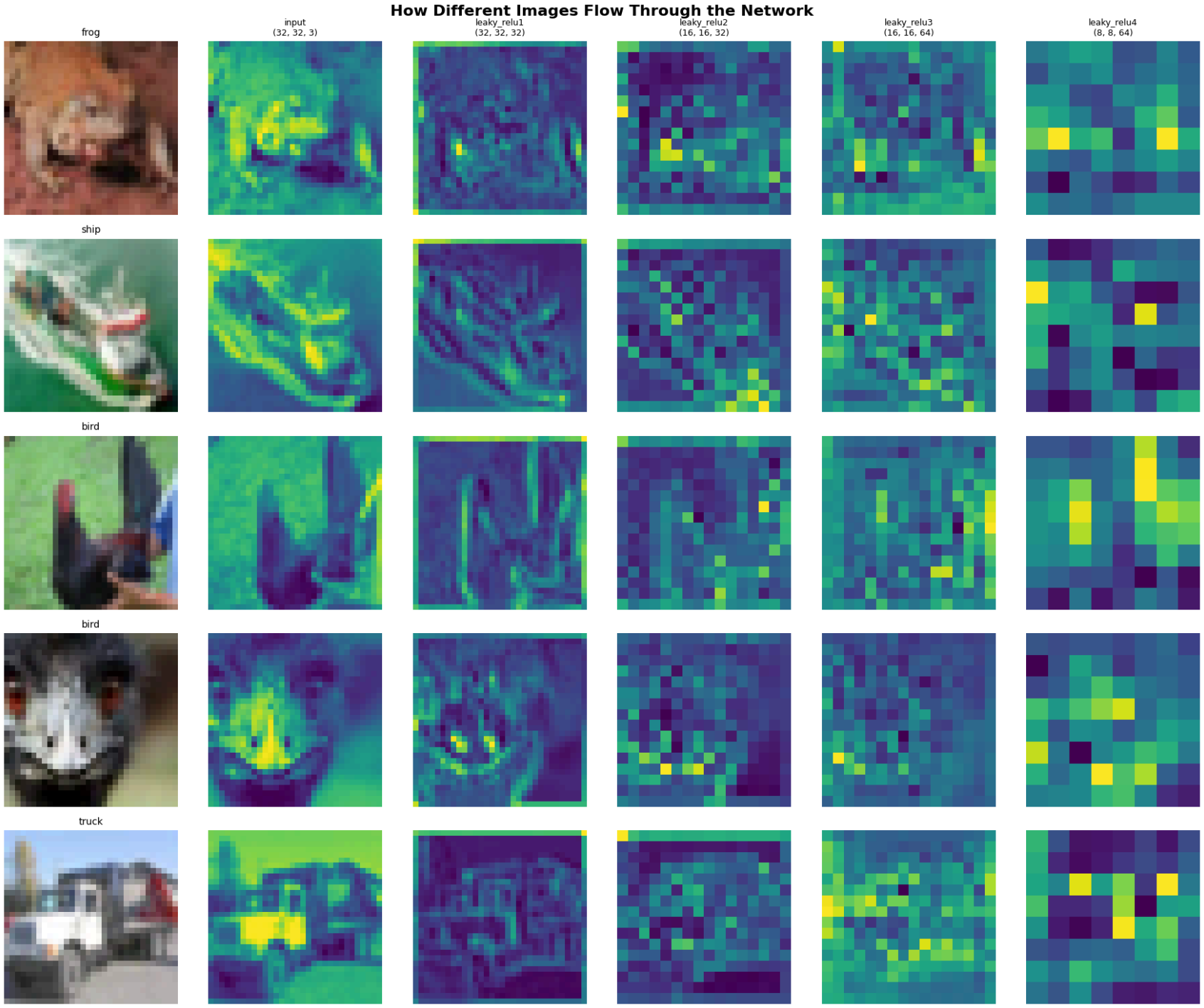
```
Class probabilities:
   airplane    :  1.19%
   automobile  :  0.60%
   bird        :  1.00%
   cat         :  1.88%
   deer        : 11.68%
   dog         :  0.27%
👉 frog        : 80.05%
   horse       :  0.01%
   ship        :  3.24%
   truck       :  0.08%
```

9. Compare Multiple Images

Let's see how different images activate the same layer.

In [19]:
```
# Select multiple images
n_images = 5
image_indices = [5, 15, 25, 35, 45]  # You can change these

fig, axes = plt.subplots(n_images, 6, figsize=(18, n_images * 3))
fig.suptitle('How Different Images Flow Through the Network', fontsize=16, weight='bold')

layer_to_show = ['input', 'leaky_relu1', 'leaky_relu2', 'leaky_relu3', 'leaky_relu4']

for row, img_idx in enumerate(image_indices):
    img = x_test[img_idx]
    true_label = CLASSES[np.argmax(y_test[img_idx])]

    # Get activations for this image
    img_activations = activation_model.predict(np.expand_dims(img, axis=0), verbose=0)

    # Show original image
    axes[row, 0].imshow(img)
    axes[row, 0].set_title(f'{true_label}', fontsize=10)
    axes[row, 0].axis('off')

    # Show activations at each layer
    for col, layer_name in enumerate(layer_to_show, start=1):
        layer_idx = layer_names.index(layer_name)
```

```
        activation = img_activations[layer_idx][0]

        # Average across channels for visualization
        avg_activation = np.mean(activation, axis=-1)

        axes[row, col].imshow(avg_activation, cmap='viridis')
        if row == 0:
            axes[row, col].set_title(f'{layer_name}\n{activation.shape}', fontsize=9)
        axes[row, col].axis('off')

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


How Different Images Flow Through the Network
frog
input
(32, 32, 3)
leaky_relu1
(32, 32, 32)
leaky_relu2
(16, 16, 32)
leaky_relu3
(16, 16, 64)
leaky_relu4
(8, 8, 64)
ship
bird
bird
truck

10. Statistical Analysis of Layers

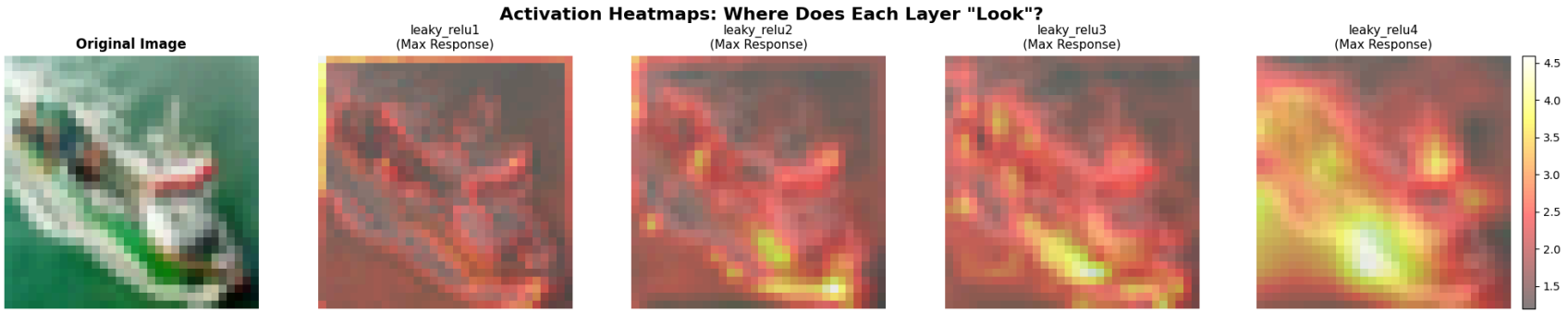
In [20]:
```
# Analyze activation statistics across layers
conv_layers = ['leaky_relu1', 'leaky_relu2', 'leaky_relu3', 'leaky_relu4']

stats = []
for layer_name in conv_layers:
    layer_idx = layer_names.index(layer_name)
    activation = activations[layer_idx][0]

    stats.append({
        'Layer': layer_name,
        'Shape': activation.shape,
        'Total Activations': np.prod(activation.shape),
        'Mean': activation.mean(),
        'Std': activation.std(),
        'Min': activation.min(),
        'Max': activation.max(),
        'Sparsity (% zeros)': (activation == 0).sum() / np.prod(activation.shape) * 100,
    })

import pandas as pd
df_stats = pd.DataFrame(stats)
print("\n Layer Statistics:")
print(df_stats.to_string(index=False))
```

```
 Layer Statistics:
      Layer        Shape  Total Activations     Mean      Std       Min      Max  Sparsity (% zeros)
leaky_relu1 (32, 32, 32)              32768 0.238628 0.671575 -2.827088 7.060660                 0.0
leaky_relu2 (16, 16, 32)               8192 0.216428 0.737777 -2.099273 6.728913                 0.0
leaky_relu3 (16, 16, 64)              16384 0.214104 0.707115 -1.704782 6.530526                 0.0
leaky_relu4   (8, 8, 64)               4096 0.204519 0.740229 -1.680109 4.715580                 0.0
```

11. Activation Heatmaps

Create heatmaps showing which parts of the image activate each layer most.

In [21]:
```
fig, axes = plt.subplots(1, 5, figsize=(20, 4))
fig.suptitle('Activation Heatmaps: Where Does Each Layer "Look"?', fontsize=16, weight='bold')

# Show original
axes[0].imshow(sample_image)
axes[0].set_title('Original Image', fontsize=12, weight='bold')
axes[0].axis('off')

# Show max activation across channels for each conv layer
for idx, layer_name in enumerate(conv_layers, start=1):
    layer_idx = layer_names.index(layer_name)
    activation = activations[layer_idx][0]

    # Take maximum across all channels (shows strongest response)
    max_activation = np.max(activation, axis=-1)

    # Upsample to original size for comparison
    from scipy.ndimage import zoom
    scale = 32 / max_activation.shape[0]
    max_activation_upsampled = zoom(max_activation, scale, order=1)

    # Overlay on original image
    axes[idx].imshow(sample_image, alpha=0.5)
    im = axes[idx].imshow(max_activation_upsampled, cmap='hot', alpha=0.5)
    axes[idx].set_title(f'{layer_name}\n(Max Response)', fontsize=11)
    axes[idx].axis('off')

plt.colorbar(im, ax=axes[-1], fraction=0.046, pad=0.04)
plt.tight_layout()
plt.show()
```



12. Summary: What We Learned

The CNN Processing Pipeline:

1. **Input Layer**: Original RGB image (32×32×3)
2. **Conv1 + LeakyReLU**:
   - Detects low-level features (edges, colors, simple textures)
   - Output: 32×32×32
   - Filters learned: edge detectors, color gradients
3. **Conv2 + LeakyReLU** (stride 2):
   - Spatial downsampling (32×32 → 16×16)
   - Mid-level features (corners, simple shapes)
   - Output: 16×16×32
4. **Conv3 + LeakyReLU**:
   - More filters (64) for complex patterns
   - Object-part detection
   - Output: 16×16×64
5. **Conv4 + LeakyReLU** (stride 2):
   - Further downsampling (16×16 → 8×8)
   - High-level semantic features
   - Output: 8×8×64
6. **Flatten**: Convert 2D feature maps to 1D vector (4,096 values)
7. **Dense + Dropout**:
   - Combine features for classification
   - Output: 128 neurons
8. **Output + Softmax**:

- Final classification probabilities
- Output: 10 classes

Key Observations:

- **Hierarchical Learning**: Early layers detect simple patterns, deeper layers detect complex concepts
- **Spatial Reduction**: Image gets smaller (32→16→8) but channels increase (3→32→64)
- **Feature Specialization**: Each filter learns different patterns
- **Activation Sparsity**: Not all neurons activate for every image

13. Interactive Exploration (Try These!)

Change these values and re-run cells to explore:

1. **Different images**: Change `img_index` in Section 3
2. **More filters**: Increase `n_features` in visualization functions
3. **Different layers**: Modify `layer_to_show` lists
4. **Color maps**: Try different `cmap` values ('viridis', 'hot', 'gray', 'jet')

Exercises:

1. Find an image that the network misclassifies - how do its activations differ?
2. Compare activations for similar classes (e.g., cat vs dog)
3. Which layer shows the most distinctive patterns for different classes?
4. How do activation patterns change between correct and incorrect predictions?

In [ ]: