Deep Learning Introduction

A Brief Introduction to Modern Neural Networks

Note: This notebook is provided to give you a glimpse of how the concepts you've learned scale up to real-world deep learning.

What You've Learned So Far

In Labs 1 and 2, you built neural networks **from scratch**:

- Perceptrons for linearly separable problems (AND, OR)
- Multi-Layer Perceptrons (MLPs) with backpropagation
- Evaluation methods (confusion matrices, cross-validation)

Building from scratch gave you deep understanding of:

- How neurons compute weighted sums and apply activation functions
- How backpropagation adjusts weights to minimise error
- Why normalisation and architecture choices matter

What This Notebook Covers

Now we'll see how these same concepts scale up using **modern frameworks**:

- **Keras/TensorFlow** - Industry-standard deep learning library
- **Digits Dataset** - Handwritten digit images (similar to the LED letter example in the slides)
- **Deep MLPs** - Multiple hidden layers learning hierarchical features
- **Convolutional Neural Networks (CNNs)** - Specialised architecture for images

---

In [ ]:
```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt

# Suppress warnings for cleaner output
import warnings
warnings.filterwarnings('ignore')

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Suppress TensorFlow logs
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'  # Disable oneDNN messages

import tensorflow as tf
tf.get_logger().setLevel('ERROR')  # Only show errors, not warnings

from tensorflow import keras
from tensorflow.keras import layers

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

print(f"TensorFlow version: {tf.__version__}")
print("\nLibraries loaded successfully!")
```

```
TensorFlow version: 2.10.0

Libraries loaded successfully!
```

---

1. The Digits Dataset

This dataset contains 1,797 grayscale images of handwritten digits (0-9):

- Each image is 8×8 pixels (64 features)
- Similar to the LED letter recognition example in your lecture slides

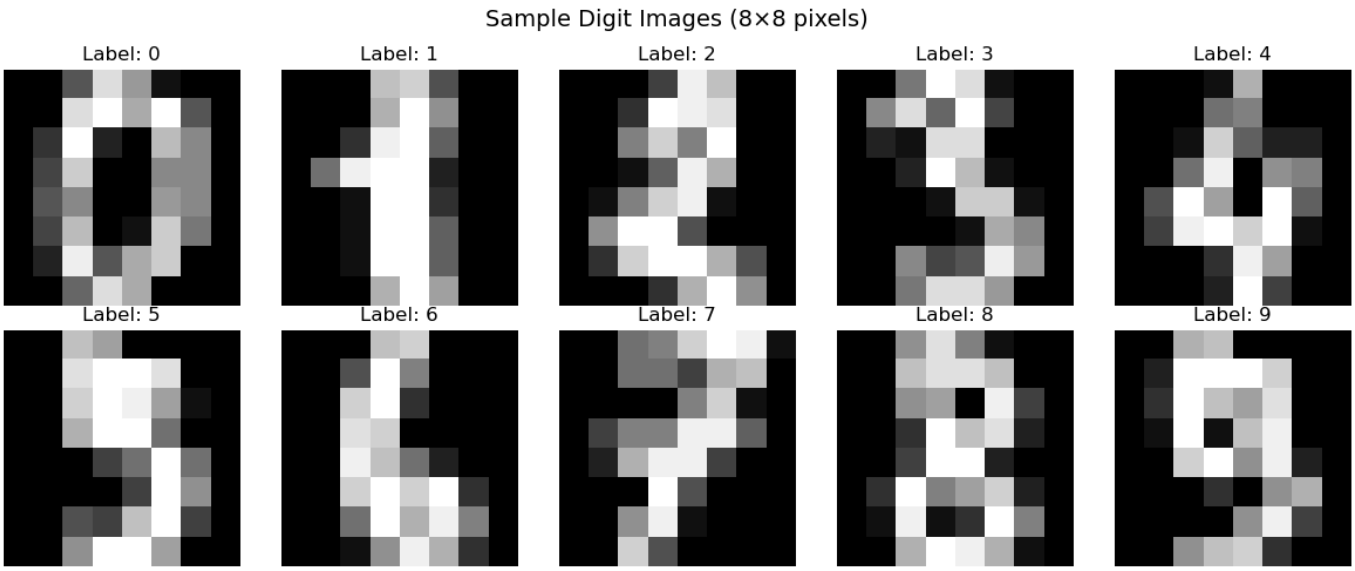
This is a smaller version of the famous MNIST dataset, good for quick experimentation.

In [ ]:
```
# Load the digits dataset
digits = load_digits()
X = digits.data
y = digits.target

print("Digits Dataset")
print("=" * 40)
print(f"Total samples: {X.shape[0]}")
print(f"Features per sample: {X.shape[1]} (8×8 = 64 pixels)")
print(f"Classes: {np.unique(y)} (digits 0-9)")
print(f"\nPixel value range: [{X.min():.0f}, {X.max():.0f}]")
```

```
Digits Dataset
========================================
Total samples: 1797
Features per sample: 64 (8×8 = 64 pixels)
Classes: [0 1 2 3 4 5 6 7 8 9] (digits 0-9)

Pixel value range: [0, 16]
```

In [ ]:
```
# Visualise some sample images
fig, axes = plt.subplots(2, 5, figsize=(12, 5))

for i, ax in enumerate(axes.flat):
```

```
    ax.imshow(digits.images[i], cmap='gray')
    ax.set_title(f'Label: {y[i]}', fontsize=12)
    ax.axis('off')

plt.suptitle('Sample Digit Images (8×8 pixels)', fontsize=14)
plt.tight_layout()
plt.savefig('digits_samples.png', dpi=150)
plt.show()

print("\nEach image is 8×8 = 64 pixels.")
print("This matches the LED letter example from your slides")
print("Compare this to Iris with just 4 features.")
```




```
Each image is 8×8 = 64 pixels.
This matches the LED letter example from your slides
Compare this to Iris with just 4 features.
```

In [ ]:
```
# Preprocess the data

# Normalise pixel values to [0, 1] (same principle as Lab 2)
scaler = MinMaxScaler()
X_norm = scaler.fit_transform(X)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X_norm, y, test_size=0.2, random_state=42, stratify=y
)

# For CNN, reshape to include channel dimension (8×8 → 8×8×1)
X_train_cnn = X_train.reshape(-1, 8, 8, 1)
X_test_cnn = X_test.reshape(-1, 8, 8, 1)

print("Preprocessed Data")
print("=" * 40)
print(f"Training samples: {X_train.shape[0]}")
print(f"Test samples:     {X_test.shape[0]}")
print(f"\nFor MLP (flat):    {X_train.shape}")
print(f"For CNN (spatial): {X_train_cnn.shape}")
print(f"\nNormalised range: [{X_norm.min():.2f}, {X_norm.max():.2f}]")
```

```
Preprocessed Data
========================================
Training samples: 1437
Test samples:     360

For MLP (flat):    (1437, 64)
For CNN (spatial): (1437, 8, 8, 1)

Normalised range: [0.00, 1.00]
```

2. Deep MLP with Keras

First, let's build a **deep MLP** - similar to the architecture you implemented from scratch, but with multiple hidden layers.

From Scratch vs Keras

Remember your Lab 2 MLP code?

```
# From scratch (Lab 2)
self.weights_ih = np.random.uniform(-0.5, 0.5, (n_inputs, n_hidden))
self.z_h = np.dot(X, self.weights_ih) + self.bias_h
self.a_h = self.sigmoid(self.z_h)
```

In Keras, a layer does all of this in one line:

```
# Keras equivalent
layers.Dense(128, activation='sigmoid')
```

Keras handles weight initialisation, matrix multiplication, bias addition, and activation - all optimised for speed.

In [ ]:
```
# Build a Deep MLP

mlp_model = keras.Sequential([
    # Input layer (64 pixels)
    layers.Input(shape=(64,)),

    # Hidden layer 1: 128 neurons with ReLU activation
    layers.Dense(128, activation='relu'),

    # Hidden layer 2: 64 neurons with ReLU activation
    layers.Dense(64, activation='relu'),

    # Hidden layer 3: 32 neurons with ReLU activation
    layers.Dense(32, activation='relu'),

    # Output layer: 10 neurons (one per digit) with softmax
    layers.Dense(10, activation='softmax')
])

# Display model architecture
mlp_model.summary()

print("\n" + "=" * 50)
print("Architecture: 64 → 128 → 64 → 32 → 10")
print("This is a 'deep' network with 3 hidden layers.")
print("=" * 50)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               8320

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 10)                330

=================================================================
Total params: 18,986
Trainable params: 18,986
Non-trainable params: 0
_________________________________________________________________

================================================
Architecture: 64 → 128 → 64 → 32 → 10
This is a 'deep' network with 3 hidden layers.
================================================
```

New Concepts

ReLU Activation: Instead of sigmoid, modern networks often use ReLU (Rectified Linear Unit):

- ReLU(x) = max(0, x)
- Faster to compute than sigmoid
- Helps avoid the "vanishing gradient" problem in deep networks

Softmax Output: For multi-class classification, softmax converts raw scores to probabilities that sum to 1.

Sparse Categorical Crossentropy: A loss function for classification when labels are integers (0-9) rather than one-hot encoded.

- **Note**: Although we use integer labels (0-9) rather than one-hot encoded vectors, the network still has 10 output neurons (one per class) with softmax activation. The "sparse" in the name refers only to how the labels are formatted, not the network architecture. The softmax outputs represent the probability of each class, and the integer label simply tells the loss function which output neuron should have the highest probability.

In [ ]:
```python
# Compile the model
mlp_model.compile(
    optimizer='adam',  # Advanced optimiser (better than basic gradient descent)
    loss='sparse_categorical_crossentropy',  # For integer labels
    metrics=['accuracy']
)

print("Model compiled!")
print("\nOptimizer: Adam (adaptive learning rate)")
print("Loss: Sparse Categorical Crossentropy")
print("Metrics: Accuracy")
```

```
Model compiled!

Optimizer: Adam (adaptive learning rate)
Loss: Sparse Categorical Crossentropy
Metrics: Accuracy
```

In [ ]:

```python
# Train the MLP
print("Training Deep MLP...")
print("=" * 50)

mlp_history = mlp_model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_split=0.1,  # Use 10% of training data for validation
    verbose=1
)

print("\nTraining complete!")
```

```
Training Deep MLP...
==================================================
Epoch 1/50
41/41 [==============================] - 1s 7ms/step - loss: 2.0255 - accuracy: 0.4826 - val_l
oss: 1.5704 - val_accuracy: 0.7708
Epoch 2/50
41/41 [==============================] - 0s 2ms/step - loss: 1.0374 - accuracy: 0.8461 - val_l
oss: 0.6452 - val_accuracy: 0.8472
Epoch 3/50
41/41 [==============================] - 0s 2ms/step - loss: 0.4251 - accuracy: 0.9172 - val_l
oss: 0.3688 - val_accuracy: 0.8750
Epoch 4/50
41/41 [==============================] - 0s 2ms/step - loss: 0.2636 - accuracy: 0.9451 - val_l
oss: 0.2857 - val_accuracy: 0.9236
Epoch 5/50
41/41 [==============================] - 0s 2ms/step - loss: 0.1919 - accuracy: 0.9637 - val_l
oss: 0.1907 - val_accuracy: 0.9514
Epoch 6/50
41/41 [==============================] - 0s 2ms/step - loss: 0.1471 - accuracy: 0.9706 - val_l
oss: 0.2223 - val_accuracy: 0.9167
Epoch 7/50
41/41 [==============================] - 0s 2ms/step - loss: 0.1227 - accuracy: 0.9722 - val_l
oss: 0.1729 - val_accuracy: 0.9444
Epoch 8/50
41/41 [==============================] - 0s 2ms/step - loss: 0.1053 - accuracy: 0.9753 - val_l
oss: 0.1488 - val_accuracy: 0.9653
Epoch 9/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0883 - accuracy: 0.9807 - val_l
oss: 0.1535 - val_accuracy: 0.9514
Epoch 10/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0795 - accuracy: 0.9861 - val_l
oss: 0.1315 - val_accuracy: 0.9583
Epoch 11/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0621 - accuracy: 0.9884 - val_l
oss: 0.1114 - val_accuracy: 0.9722
Epoch 12/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0526 - accuracy: 0.9899 - val_l
oss: 0.1278 - val_accuracy: 0.9514
Epoch 13/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0511 - accuracy: 0.9892 - val_l
oss: 0.1090 - val_accuracy: 0.9653
Epoch 14/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0413 - accuracy: 0.9930 - val_l
oss: 0.1244 - val_accuracy: 0.9653
Epoch 15/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0398 - accuracy: 0.9923 - val_l
oss: 0.1271 - val_accuracy: 0.9583
Epoch 16/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0328 - accuracy: 0.9946 - val_l
oss: 0.1032 - val_accuracy: 0.9722
Epoch 17/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0252 - accuracy: 0.9969 - val_l
oss: 0.1505 - val_accuracy: 0.9375
Epoch 18/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0260 - accuracy: 0.9961 - val_l
oss: 0.0863 - val_accuracy: 0.9792
Epoch 19/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0233 - accuracy: 0.9961 - val_l
oss: 0.1023 - val_accuracy: 0.9653
Epoch 20/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0201 - accuracy: 0.9977 - val_l
oss: 0.0862 - val_accuracy: 0.9583
Epoch 21/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0187 - accuracy: 0.9992 - val_l
oss: 0.1114 - val_accuracy: 0.9514
Epoch 22/50
```

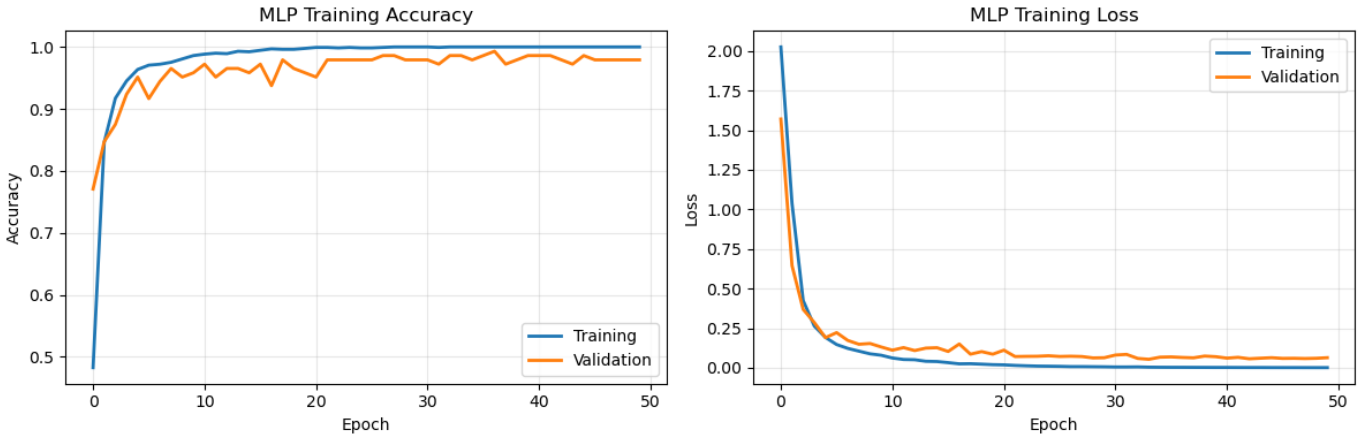
```
41/41 [==============================] - 0s 2ms/step - loss: 0.0152 - accuracy: 0.9992 - val_l
oss: 0.0713 - val_accuracy: 0.9792
Epoch 23/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0128 - accuracy: 0.9985 - val_l
oss: 0.0724 - val_accuracy: 0.9792
Epoch 24/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0109 - accuracy: 0.9992 - val_l
oss: 0.0730 - val_accuracy: 0.9792
Epoch 25/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0102 - accuracy: 0.9985 - val_l
oss: 0.0762 - val_accuracy: 0.9792
Epoch 26/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0093 - accuracy: 0.9985 - val_l
oss: 0.0717 - val_accuracy: 0.9792
Epoch 27/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0077 - accuracy: 0.9992 - val_l
oss: 0.0732 - val_accuracy: 0.9861
Epoch 28/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0077 - accuracy: 1.0000 - val_l
oss: 0.0713 - val_accuracy: 0.9861
Epoch 29/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0071 - accuracy: 1.0000 - val_l
oss: 0.0626 - val_accuracy: 0.9792
Epoch 30/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0064 - accuracy: 1.0000 - val_l
oss: 0.0638 - val_accuracy: 0.9792
Epoch 31/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0053 - accuracy: 1.0000 - val_l
oss: 0.0808 - val_accuracy: 0.9792
Epoch 32/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0054 - accuracy: 0.9992 - val_l
oss: 0.0848 - val_accuracy: 0.9722
Epoch 33/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0059 - accuracy: 1.0000 - val_l
oss: 0.0594 - val_accuracy: 0.9861
Epoch 34/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0041 - accuracy: 1.0000 - val_l
oss: 0.0539 - val_accuracy: 0.9861
Epoch 35/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0036 - accuracy: 1.0000 - val_l
oss: 0.0673 - val_accuracy: 0.9792
Epoch 36/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0032 - accuracy: 1.0000 - val_l
oss: 0.0690 - val_accuracy: 0.9861
Epoch 37/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0031 - accuracy: 1.0000 - val_l
oss: 0.0654 - val_accuracy: 0.9931
Epoch 38/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0028 - accuracy: 1.0000 - val_l
oss: 0.0632 - val_accuracy: 0.9722
Epoch 39/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0027 - accuracy: 1.0000 - val_l
oss: 0.0743 - val_accuracy: 0.9792
Epoch 40/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0024 - accuracy: 1.0000 - val_l
oss: 0.0704 - val_accuracy: 0.9861
Epoch 41/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0024 - accuracy: 1.0000 - val_l
oss: 0.0612 - val_accuracy: 0.9861
Epoch 42/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0021 - accuracy: 1.0000 - val_l
oss: 0.0662 - val_accuracy: 0.9861
Epoch 43/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0021 - accuracy: 1.0000 - val_l
oss: 0.0571 - val_accuracy: 0.9792
Epoch 44/50
```

```
41/41 [==============================] - 0s 2ms/step - loss: 0.0020 - accuracy: 1.0000 - val_l
oss: 0.0607 - val_accuracy: 0.9722
Epoch 45/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0017 - accuracy: 1.0000 - val_l
oss: 0.0640 - val_accuracy: 0.9861
Epoch 46/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0016 - accuracy: 1.0000 - val_l
oss: 0.0595 - val_accuracy: 0.9792
Epoch 47/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0016 - accuracy: 1.0000 - val_l
oss: 0.0602 - val_accuracy: 0.9792
Epoch 48/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0014 - accuracy: 1.0000 - val_l
oss: 0.0586 - val_accuracy: 0.9792
Epoch 49/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0013 - accuracy: 1.0000 - val_l
oss: 0.0602 - val_accuracy: 0.9792
Epoch 50/50
41/41 [==============================] - 0s 2ms/step - loss: 0.0013 - accuracy: 1.0000 - val_l
oss: 0.0640 - val_accuracy: 0.9792

Training complete!
```

In [ ]:
```python
# Evaluate on test set
mlp_loss, mlp_accuracy = mlp_model.evaluate(X_test, y_test, verbose=0)

print("Deep MLP Results")
print("=" * 40)
print(f"Test Loss:     {mlp_loss:.4f}")
print(f"Test Accuracy: {mlp_accuracy * 100:.2f}%")
print(f"\nCorrectly classified: {int(mlp_accuracy * len(y_test))} / {len(y_test)} images")
```

```
Deep MLP Results
========================================
Test Loss:     0.0741
Test Accuracy: 98.33%

Correctly classified: 354 / 360 images
```

In [ ]:
```python
# Plot training history
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Accuracy
axes[0].plot(mlp_history.history['accuracy'], label='Training', linewidth=2)
axes[0].plot(mlp_history.history['val_accuracy'], label='Validation', linewidth=2)
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Accuracy')
axes[0].set_title('MLP Training Accuracy')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Loss
axes[1].plot(mlp_history.history['loss'], label='Training', linewidth=2)
axes[1].plot(mlp_history.history['val_loss'], label='Validation', linewidth=2)
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Loss')
axes[1].set_title('MLP Training Loss')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('mlp_training_history.png', dpi=150)
plt.show()
```



---

3. Convolutional Neural Network (CNN)

MLPs treat images as flat vectors, losing spatial information. **CNNs** are designed specifically for images:

- **Convolutional layers** slide small filters across the image to detect local patterns (edges, corners, textures)
- **Pooling layers** reduce spatial dimensions while keeping important features
- **Feature hierarchy** emerges: early layers detect simple patterns, deeper layers detect complex objects

This matches what your lecture slides describe as "hierarchical feature representations" (slide 68).

In [ ]:
```python
# Build a CNN

cnn_model = keras.Sequential([
    # Input: 8×8×1 image
    layers.Input(shape=(8, 8, 1)),

    # Convolutional layer 1: 32 filters, 3×3 kernel
    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),

    # Convolutional layer 2: 64 filters, 3×3 kernel
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2)),  # Reduce to 4×4

    # Flatten and dense layers
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.3),  # Regularisation to prevent overfitting

    # Output layer
    layers.Dense(10, activation='softmax')
])

cnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 8, 8, 32)          320

 conv2d_1 (Conv2D)           (None, 8, 8, 64)          18496

 max_pooling2d (MaxPooling2D  (None, 4, 4, 64)         0
 )

 flatten (Flatten)           (None, 1024)              0

 dense_4 (Dense)             (None, 64)                65600

 dropout (Dropout)           (None, 64)                0

 dense_5 (Dense)             (None, 10)                650

=================================================================
Total params: 85,066
Trainable params: 85,066
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

```python
# Compile and train the CNN
cnn_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

print("Training CNN...")
print("=" * 50)

cnn_history = cnn_model.fit(
    X_train_cnn, y_train,
    epochs=30,
    batch_size=32,
    validation_split=0.1,
    verbose=1
)

print("\nTraining complete!")
```

```
Training CNN...
==================================================
Epoch 1/30
41/41 [==============================] - 1s 8ms/step - loss: 1.7922 - accuracy: 0.4517 - val_l
oss: 0.8824 - val_accuracy: 0.8750
Epoch 2/30
41/41 [==============================] - 0s 4ms/step - loss: 0.6773 - accuracy: 0.8074 - val_l
oss: 0.2874 - val_accuracy: 0.9236
Epoch 3/30
41/41 [==============================] - 0s 4ms/step - loss: 0.3515 - accuracy: 0.8910 - val_l
oss: 0.1777 - val_accuracy: 0.9514
Epoch 4/30
41/41 [==============================] - 0s 4ms/step - loss: 0.2607 - accuracy: 0.9188 - val_l
oss: 0.1653 - val_accuracy: 0.9444
Epoch 5/30
41/41 [==============================] - 0s 4ms/step - loss: 0.1903 - accuracy: 0.9459 - val_l
oss: 0.1091 - val_accuracy: 0.9722
Epoch 6/30
41/41 [==============================] - 0s 3ms/step - loss: 0.1629 - accuracy: 0.9544 - val_l
oss: 0.0842 - val_accuracy: 0.9722
Epoch 7/30
41/41 [==============================] - 0s 3ms/step - loss: 0.1346 - accuracy: 0.9567 - val_l
oss: 0.0753 - val_accuracy: 0.9861
Epoch 8/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0998 - accuracy: 0.9675 - val_l
oss: 0.0982 - val_accuracy: 0.9792
Epoch 9/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0917 - accuracy: 0.9753 - val_l
oss: 0.0862 - val_accuracy: 0.9653
Epoch 10/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0816 - accuracy: 0.9737 - val_l
oss: 0.0816 - val_accuracy: 0.9792
Epoch 11/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0643 - accuracy: 0.9807 - val_l
oss: 0.0761 - val_accuracy: 0.9792
Epoch 12/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0549 - accuracy: 0.9838 - val_l
oss: 0.0341 - val_accuracy: 1.0000
Epoch 13/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0667 - accuracy: 0.9783 - val_l
oss: 0.0389 - val_accuracy: 0.9931
Epoch 14/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0487 - accuracy: 0.9861 - val_l
oss: 0.0312 - val_accuracy: 0.9931
Epoch 15/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0490 - accuracy: 0.9838 - val_l
oss: 0.0474 - val_accuracy: 0.9792
Epoch 16/30
41/41 [==============================] - 0s 4ms/step - loss: 0.0523 - accuracy: 0.9838 - val_l
oss: 0.0531 - val_accuracy: 0.9861
Epoch 17/30
41/41 [==============================] - 0s 4ms/step - loss: 0.0452 - accuracy: 0.9869 - val_l
oss: 0.0330 - val_accuracy: 0.9861
Epoch 18/30
41/41 [==============================] - 0s 4ms/step - loss: 0.0359 - accuracy: 0.9876 - val_l
oss: 0.0330 - val_accuracy: 0.9861
Epoch 19/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0372 - accuracy: 0.9876 - val_l
oss: 0.0380 - val_accuracy: 0.9861
Epoch 20/30
41/41 [==============================] - 0s 4ms/step - loss: 0.0364 - accuracy: 0.9830 - val_l
oss: 0.0244 - val_accuracy: 0.9931
Epoch 21/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0384 - accuracy: 0.9876 - val_l
oss: 0.0431 - val_accuracy: 0.9861
Epoch 22/30
```

```
41/41 [==============================] - 0s 4ms/step - loss: 0.0254 - accuracy: 0.9938 - val_l
oss: 0.0276 - val_accuracy: 0.9931
Epoch 23/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0281 - accuracy: 0.9923 - val_l
oss: 0.0188 - val_accuracy: 1.0000
Epoch 24/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0248 - accuracy: 0.9930 - val_l
oss: 0.0331 - val_accuracy: 0.9861
Epoch 25/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0193 - accuracy: 0.9954 - val_l
oss: 0.0218 - val_accuracy: 0.9931
Epoch 26/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0185 - accuracy: 0.9954 - val_l
oss: 0.0383 - val_accuracy: 0.9792
Epoch 27/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0169 - accuracy: 0.9954 - val_l
oss: 0.0317 - val_accuracy: 0.9861
Epoch 28/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0153 - accuracy: 0.9961 - val_l
oss: 0.0495 - val_accuracy: 0.9792
Epoch 29/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0243 - accuracy: 0.9907 - val_l
oss: 0.0298 - val_accuracy: 0.9722
Epoch 30/30
41/41 [==============================] - 0s 3ms/step - loss: 0.0160 - accuracy: 0.9969 - val_l
oss: 0.0229 - val_accuracy: 0.9931

Training complete!
```

In [ ]:
```python
# Evaluate CNN
cnn_loss, cnn_accuracy = cnn_model.evaluate(X_test_cnn, y_test, verbose=0)

print("CNN Results")
print("=" * 40)
print(f"Test Loss:     {cnn_loss:.4f}")
print(f"Test Accuracy: {cnn_accuracy * 100:.2f}%")
print(f"\nCorrectly classified: {int(cnn_accuracy * len(y_test))} / {len(y_test)} images")
```

```
CNN Results
========================================
Test Loss:     0.0341
Test Accuracy: 98.61%

Correctly classified: 354 / 360 images
```

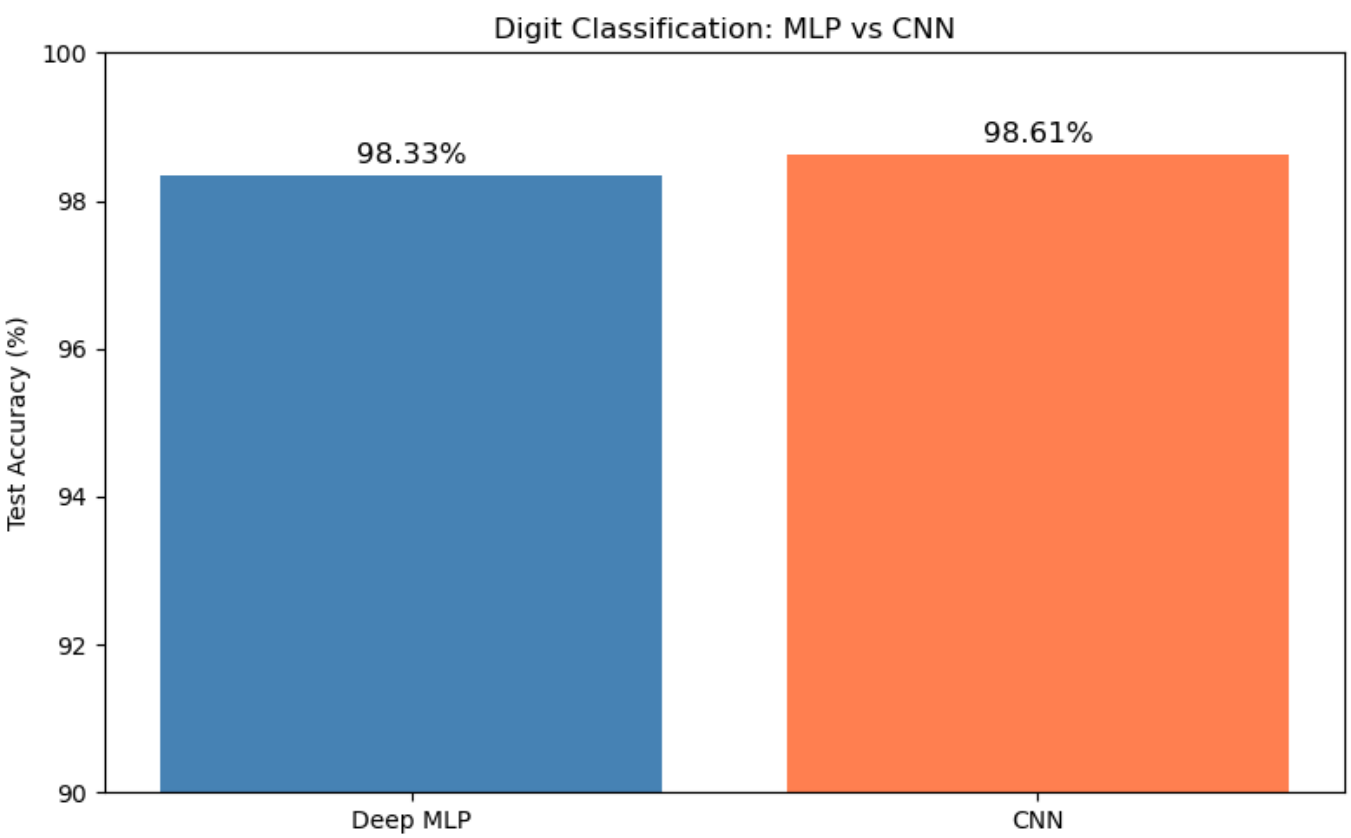
---

4. Comparison: MLP vs CNN

In [ ]:
```python
# Compare results

print("\n" + "=" * 55)
print("COMPARISON: Deep MLP vs CNN on Digits Dataset")
print("=" * 55)

print(f"\n{'Model':<20} {'Test Accuracy':>15} {'Parameters':>15}")
print("-" * 55)
print(f"{'Deep MLP':<20} {mlp_accuracy*100:>14.2f}% {mlp_model.count_params():>15,}")
print(f"{'CNN':<20} {cnn_accuracy*100:>14.2f}% {cnn_model.count_params():>15,}")

print("\n" + "=" * 55)
print("Key Observations:")
print("-" * 55)
print("• Both achieve high accuracy on this small dataset")
print("• CNN exploits spatial structure of images")
```

```python
print("• MLP treats each pixel independently (loses structure)")
print("• On larger images (like MNIST 28×28), CNN advantage is clearer")
print("=" * 55)

# Bar chart comparison
fig, ax = plt.subplots(figsize=(8, 5))

models = ['Deep MLP', 'CNN']
accuracies = [mlp_accuracy * 100, cnn_accuracy * 100]
colors = ['steelblue', 'coral']

bars = ax.bar(models, accuracies, color=colors)

ax.set_ylabel('Test Accuracy (%)')
ax.set_title('Digit Classification: MLP vs CNN')
ax.set_ylim(90, 100)

# Add value labels
for bar, acc in zip(bars, accuracies):
    ax.annotate(f'{acc:.2f}%',
                xy=(bar.get_x() + bar.get_width() / 2, bar.get_height()),
                xytext=(0, 3),
                textcoords="offset points",
                ha='center', va='bottom', fontsize=12)

plt.tight_layout()
plt.savefig('mlp_vs_cnn_comparison.png', dpi=150)
plt.show()
```

```
============================================================
COMPARISON: Deep MLP vs CNN on Digits Dataset
============================================================

Model                  Test Accuracy      Parameters
------------------------------------------------------------
Deep MLP                       98.33%          18,986
CNN                            98.61%          85,066

============================================================
Key Observations:
------------------------------------------------------------
• Both achieve high accuracy on this small dataset
• CNN exploits spatial structure of images
• MLP treats each pixel independently (loses structure)
• On larger images (like MNIST 28×28), CNN advantage is clearer
============================================================
```



---

5. Visualise Predictions

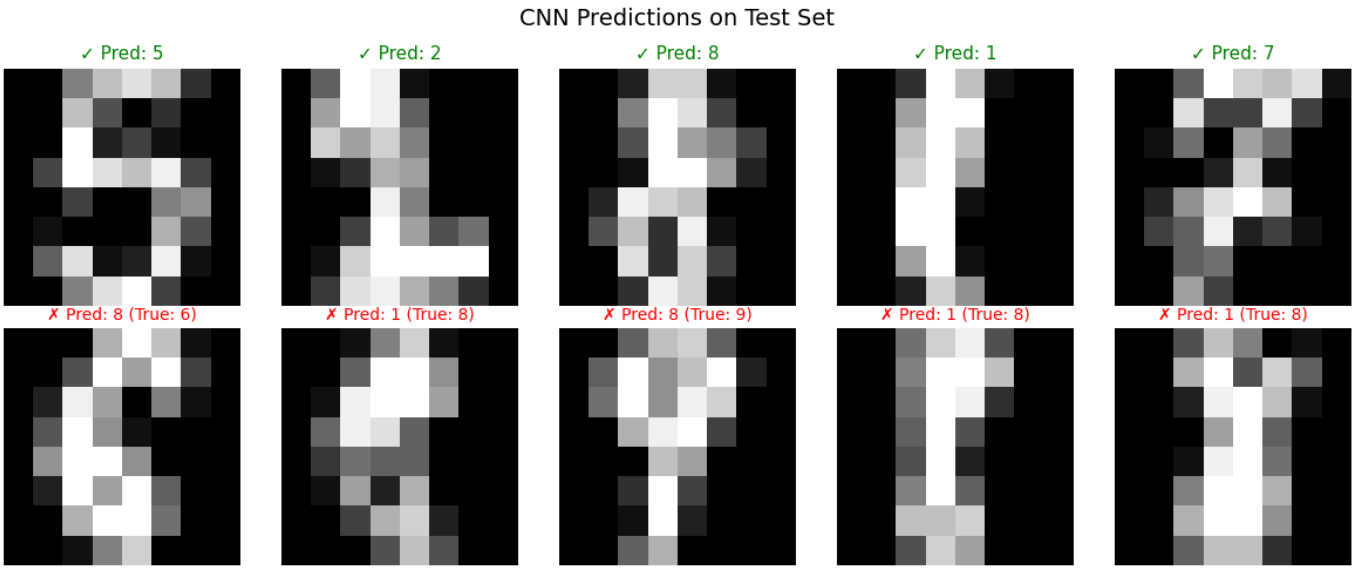
Let's see what the models actually predict, including some mistakes.

In [ ]:
```
# Get predictions
cnn_predictions = cnn_model.predict(X_test_cnn, verbose=0)
cnn_pred_labels = np.argmax(cnn_predictions, axis=1)

# Find correct and incorrect predictions
correct_indices = np.where(cnn_pred_labels == y_test)[0]
incorrect_indices = np.where(cnn_pred_labels != y_test)[0]

print(f"Correct predictions: {len(correct_indices)}")
print(f"Incorrect predictions: {len(incorrect_indices)}")
```

```
Correct predictions: 355
Incorrect predictions: 5
```

In [ ]:
```
# Visualise some correct and incorrect predictions
fig, axes = plt.subplots(2, 5, figsize=(12, 5))

# Top row: Correct predictions
for i, ax in enumerate(axes[0]):
    idx = correct_indices[i]
    ax.imshow(X_test[idx].reshape(8, 8), cmap='gray')
    ax.set_title(f'✓ Pred: {cnn_pred_labels[idx]}', fontsize=11, color='green')
    ax.axis('off')

# Bottom row: Incorrect predictions (if any)
for i, ax in enumerate(axes[1]):
    if i < len(incorrect_indices):
        idx = incorrect_indices[i]
        ax.imshow(X_test[idx].reshape(8, 8), cmap='gray')
        ax.set_title(f'✗ Pred: {cnn_pred_labels[idx]} (True: {y_test[idx]})',
                     fontsize=10, color='red')
    else:
```

```
        ax.set_title('(No more errors)', fontsize=10, color='gray')
    ax.axis('off')

plt.suptitle('CNN Predictions on Test Set', fontsize=14)
plt.tight_layout()
plt.savefig('cnn_predictions.png', dpi=150)
plt.show()

print("\nNotice: The errors are often on ambiguous or poorly written digits.")
print("Even humans might struggle with some of these!")
```




```
Notice: The errors are often on ambiguous or poorly written digits.
Even humans might struggle with some of these!
```

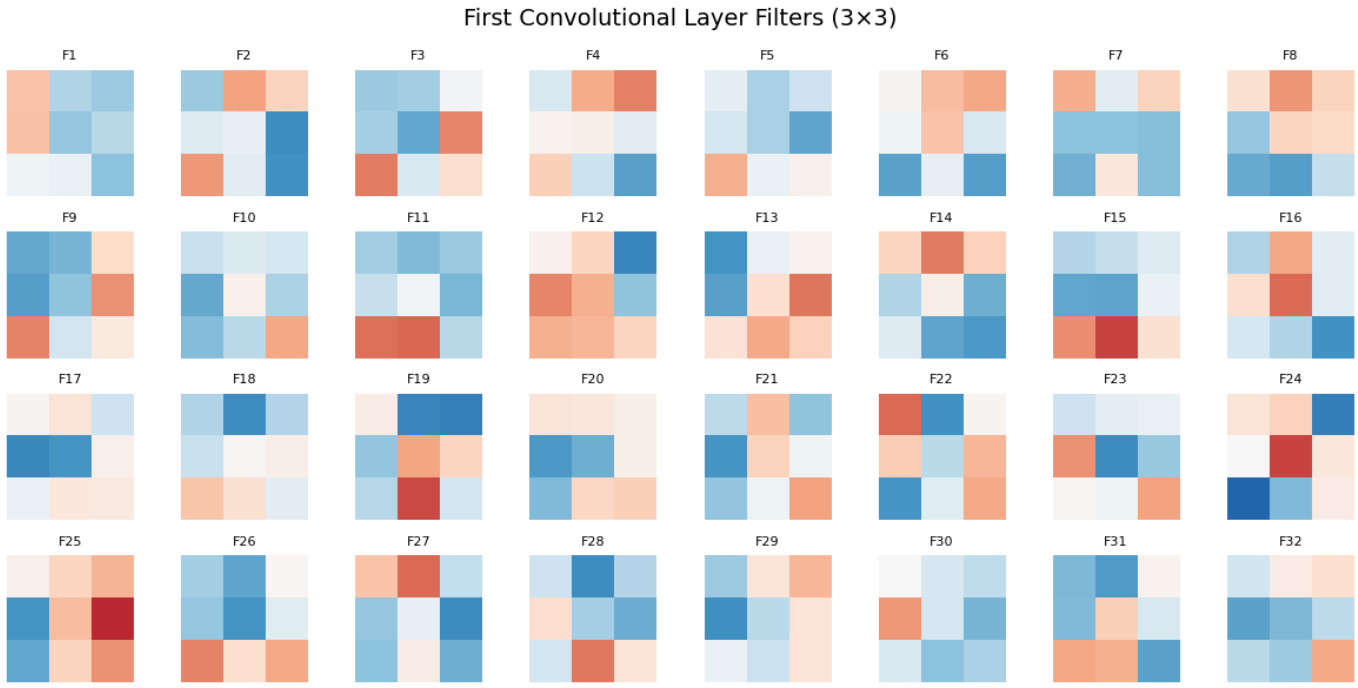
---

6. What Does the Network "See"?

One advantage of CNNs is that we can visualise what the convolutional filters have learned.

In [ ]:
```
# Visualise first layer filters
first_layer_weights = cnn_model.layers[0].get_weights()[0]
print(f"First convolutional layer filter shape: {first_layer_weights.shape}")
print("(3×3 kernel, 1 input channel, 32 filters)")

# Plot some filters
fig, axes = plt.subplots(4, 8, figsize=(12, 6))

for i, ax in enumerate(axes.flat):
    if i < 32:
        # Get the i-th filter (3×3)
        filt = first_layer_weights[:, :, 0, i]
        ax.imshow(filt, cmap='RdBu', vmin=-0.5, vmax=0.5)
        ax.axis('off')
        ax.set_title(f'F{i+1}', fontsize=8)

plt.suptitle('First Convolutional Layer Filters (3×3)', fontsize=14)
plt.tight_layout()
plt.savefig('cnn_filters.png', dpi=150)
plt.show()

print("\nThese 3×3 filters detect basic features like edges and gradients.")
print("Red = positive weights, Blue = negative weights.")
print("Deeper layers combine these to detect more complex patterns.")
```

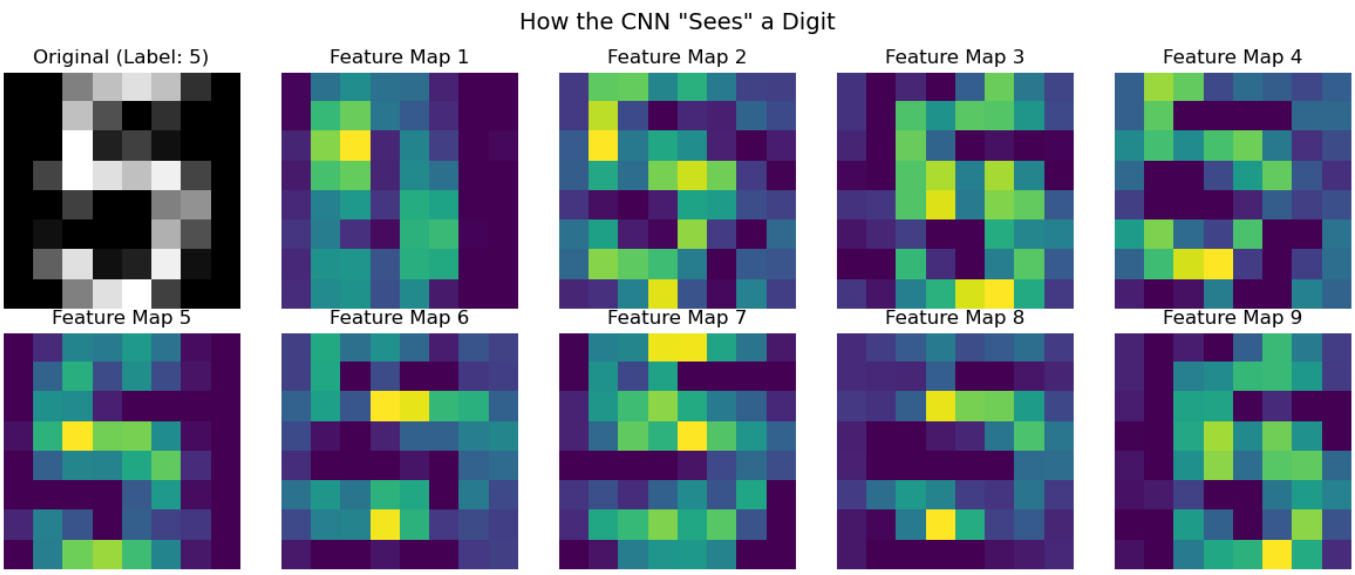
```
First convolutional layer filter shape: (3, 3, 1, 32)
(3×3 kernel, 1 input channel, 32 filters)
```



These 3×3 filters detect basic features like edges and gradients.
Red = positive weights, Blue = negative weights.
Deeper layers combine these to detect more complex patterns.

In [ ]:
```
# Visualise feature maps for a sample image

# Create a model that outputs first conv layer activations
feature_model = keras.Model(
    inputs=cnn_model.inputs,
    outputs=cnn_model.layers[0].output
)

# Get feature maps for first test image
sample_idx = 0
sample_image = X_test_cnn[sample_idx:sample_idx+1]
feature_maps = feature_model.predict(sample_image, verbose=0)

print(f"Feature maps shape: {feature_maps.shape}")
print("(1 sample, 8×8 spatial, 32 feature maps)")

# Plot original image and some feature maps
fig, axes = plt.subplots(2, 5, figsize=(12, 5))

# Original image
axes[0, 0].imshow(X_test[sample_idx].reshape(8, 8), cmap='gray')
axes[0, 0].set_title(f'Original (Label: {y_test[sample_idx]})')
axes[0, 0].axis('off')

# Feature maps
for i, ax in enumerate(axes.flat[1:]):
    if i < 9:
        ax.imshow(feature_maps[0, :, :, i], cmap='viridis')
        ax.set_title(f'Feature Map {i+1}')
        ax.axis('off')

plt.suptitle('How the CNN "Sees" a Digit', fontsize=14)
plt.tight_layout()
plt.savefig('cnn_feature_maps.png', dpi=150)
plt.show()

print("\nEach feature map highlights different aspects of the image.")
print("The network learns which features are useful for classification.")
```

```
Feature maps shape: (1, 8, 8, 32)
(1 sample, 8×8 spatial, 32 feature maps)
```




```
Each feature map highlights different aspects of the image.
The network learns which features are useful for classification.
```

---

7. Compare with Your From-Scratch MLP

Let's see how the Keras deep MLP compares to the MLP you built from scratch in Lab 2.

In [ ]:
```python
# Your Lab 2 MLP implementation (simplified)

class MLPFromScratch:
    """Your Lab 2 MLP with one hidden layer."""

    def __init__(self, n_inputs, n_hidden, n_outputs, learning_rate=0.5):
        self.learning_rate = learning_rate
        self.weights_ih = np.random.uniform(-0.5, 0.5, (n_inputs, n_hidden))
        self.bias_h = np.random.uniform(-0.5, 0.5, n_hidden)
        self.weights_ho = np.random.uniform(-0.5, 0.5, (n_hidden, n_outputs))
        self.bias_o = np.random.uniform(-0.5, 0.5, n_outputs)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))

    def forward(self, X):
        self.z_h = np.dot(X, self.weights_ih) + self.bias_h
        self.a_h = self.sigmoid(self.z_h)
        self.z_o = np.dot(self.a_h, self.weights_ho) + self.bias_o
        self.a_o = self.sigmoid(self.z_o)
        return self.a_o

    def backward(self, X, y):
        n_samples = X.shape[0]
        output_error = y - self.a_o
        output_delta = output_error * self.a_o * (1 - self.a_o)
        hidden_error = np.dot(output_delta, self.weights_ho.T)
        hidden_delta = hidden_error * self.a_h * (1 - self.a_h)

        self.weights_ho += self.learning_rate * np.dot(self.a_h.T, output_delta) / n_samples
        self.bias_o += self.learning_rate * np.mean(output_delta, axis=0)
        self.weights_ih += self.learning_rate * np.dot(X.T, hidden_delta) / n_samples
        self.bias_h += self.learning_rate * np.mean(hidden_delta, axis=0)

    def train(self, X, y, epochs):
        for epoch in range(epochs):
            self.forward(X)
            self.backward(X, y)
```

```
    def predict(self, X):
        output = self.forward(X)
        return np.argmax(output, axis=1)

# One-hot encode labels for from-scratch MLP
y_train_onehot = np.zeros((len(y_train), 10))
for i, label in enumerate(y_train):
    y_train_onehot[i, label] = 1

# Train from-scratch MLP
print("Training your Lab 2 MLP (from scratch)...")
np.random.seed(42)
scratch_mlp = MLPFromScratch(n_inputs=64, n_hidden=32, n_outputs=10, learning_rate=0.5)
scratch_mlp.train(X_train, y_train_onehot, epochs=500)

# Evaluate
scratch_pred = scratch_mlp.predict(X_test)
scratch_accuracy = np.mean(scratch_pred == y_test)

print(f"\nFrom-Scratch MLP Test Accuracy: {scratch_accuracy * 100:.2f}%")
```

```
Training your Lab 2 MLP (from scratch)...

From-Scratch MLP Test Accuracy: 83.33%
```

In [ ]:
```
# Final comparison

print("\n" + "=" * 60)
print("FINAL COMPARISON: All Models on Digits Dataset")
print("=" * 60)

print(f"\n{'Model':<30} {'Test Accuracy':>15}")
print("-" * 50)
print(f"{'Your Lab 2 MLP (from scratch)':<30} {scratch_accuracy*100:>14.2f}%")
print(f"{'Keras Deep MLP (3 hidden)':<30} {mlp_accuracy*100:>14.2f}%")
print(f"{'Keras CNN':<30} {cnn_accuracy*100:>14.2f}%")

print("\n" + "=" * 60)
print("What This Shows:")
print("-" * 60)
print("• Your from-scratch implementation works correctly")
print("• Deep networks and CNNs achieve higher accuracy")
print("• Keras handles optimisation details (Adam, better init)")
print("• The concepts are the same - scale and efficiency differ")
print("=" * 60)
```

```
============================================================
FINAL COMPARISON: All Models on Digits Dataset
============================================================

Model                          Test Accuracy
--------------------------------------------------
Your Lab 2 MLP (from scratch)          83.33%
Keras Deep MLP (3 hidden)              98.33%
Keras CNN                              98.61%

============================================================
What This Shows:
------------------------------------------------------------
• Your from-scratch implementation works correctly!
• Deep networks and CNNs achieve higher accuracy
• Keras handles optimisation details (Adam, better init)
• The concepts are the same - scale and efficiency differ
============================================================
```

8. Connecting Back to What You've Learned

Same Principles, Different Scale

Concept	Your Lab Implementation	Deep Learning
Neuron	Weighted sum + sigmoid	Same (different activations)
Training	Backpropagation	Same (more efficient implementation)
Learning rate	Fixed, manual	Adaptive (Adam optimiser)
Architecture	1 hidden layer	Many layers (feature hierarchy)
Data size	~150 samples (Iris)	1,000s to millions of samples

What Makes Deep Learning "Deep"?

1. **Multiple hidden layers** - Each layer learns increasingly abstract features
    - Layer 1: edges
    - Layer 2: shapes formed by edges
    - Deeper layers: complex objects
2. **Specialised architectures** - CNNs for images, RNNs for sequences, Transformers for language
3. **Scale** - Millions/billions of parameters, massive datasets
4. **Hardware** - GPUs/TPUs for parallel computation

The Foundation You've Built

Understanding how neural networks work from scratch means you can:

- Debug when things go wrong
- Make informed architecture decisions
- Understand research papers
- Adapt techniques to new problems

Frameworks like Keras handle the implementation details, but the concepts remain the same.

Summary

In this notebook, you've seen:

1. **Digits Dataset** - 1,797 handwritten digit images (8×8 pixels), similar to your lecture slides' LED letter example
2. **Deep MLP with Keras** - Multiple hidden layers in a few lines of code
    - Same concepts as your Lab 2 implementation
    - Better optimisation (Adam) and initialisation
3. **Convolutional Neural Network (CNN)** - Specialised architecture for images
    - Exploits spatial structure

    - Convolutional filters detect local patterns
    - Feature hierarchy: simple → complex
4. **Feature Visualisation** - What the network "sees"
    - First layer detects edges and gradients
    - Feature maps show how the image is processed
5. **Comparison with Your Implementation** - Same principles, different scale

If you want to explore deep learning further:

- Keras Documentation
- TensorFlow Tutorials
- Stanford CS231n: CNNs for Visual Recognition
- 3Blue1Brown Neural Network Videos