Guide 3 — Convolutional Neural Networks

From Pixels to Patterns: CNNs on MNIST

Written by: Pierre Chambet
Date: June 2025

Quote of the Journey

"The eye sees only what the mind is prepared to comprehend." — Robertson Davies

Introduction

In Guide 2, we built a neural network that could read numbers — but only by flattening them, destroying their shape, and hoping the model could piece them back together.

That approach worked — surprisingly well. But it ignored something crucial: **space**. In images, proximity matters. Edges matter. Orientation, contours, symmetry — they all matter.

In this guide, we step into the spatial domain. We give our models eyes — not just memory. Using **convolutional neural networks (CNNs)**, we'll teach the machine to recognize not just pixels, but *patterns*. Filters, features, activations — this is deep learning that feels more visual, more organic, and more human.

This document is your full walkthrough. From reshaping input data to decoding filters and activations. From understanding parameter sharing to training a model that **sees**. Clear, rigorous, practical.

Let's begin.

Step 1 — Why a CNN? (Context and Motivation)

Before jumping into the code, we need to understand **why** we use Convolutional Neural Networks (CNNs) for image data like MNIST.

1.1 — The Limitations of Fully Connected Networks

In previous sections, we used `Dense` (fully connected) layers to classify images from the MNIST dataset. While this works surprisingly well, it suffers from several limitations:

- **No spatial structure:** A fully connected layer treats every input pixel independently. It flattens the 28x28 image into a 784-dimensional vector and ignores the fact that neighboring pixels form patterns (like edges).
- **Too many parameters:** A dense layer connecting 784 input neurons to 128 hidden units requires $784 \times 128 = 100{,}352$ parameters. This number explodes as we increase image resolution.
- **Lack of translation invariance:** Fully connected networks do not generalize well to small shifts in the input image (e.g., a digit shifted by 2 pixels).

1.2 — Convolution: A Natural Fit for Images

CNNs exploit three core ideas:

1. **Local connectivity:** Neurons are only connected to a small region of the input (e.g., a 3x3 patch). This allows the network to learn spatially local patterns such as edges, corners, or textures.
2. **Parameter sharing:** The same set of weights (called a filter or kernel) is applied across the entire image. This greatly reduces the number of parameters and enforces translational invariance.
3. **Downsampling:** Pooling layers reduce the spatial dimensions of intermediate representations, making the network more efficient and robust to small variations.

1.3 — MNIST: A Perfect Playground

The MNIST dataset is ideal for CNNs because:

- Each image is a small, centered 28x28 grayscale digit.
- The dataset is large enough to train a small CNN from scratch.
- The visual patterns (strokes, loops, edges) are ideal for convolutional filters to capture.

1.4 — From Biological Inspiration to Deep Learning

CNNs are inspired by the **visual cortex** of the brain. Neuroscientists have found that cells in the early visual areas (like V1) respond to local patterns in the visual field (like edges at specific orientations). This biological insight led to the first convolutional architectures (e.g., LeNet-5).

In summary, CNNs are designed to respect the structure of image data. Instead of treating an image as a flat vector, they process it as a 2D grid with spatial coherence. This makes them vastly more efficient and powerful for visual recognition tasks.

Next step: *Let's now prepare our data so it fits the expected input format for a CNN. That's the subject of Step 2.*

Step 2 — Preparing the MNIST Data for a CNN

Unlike fully connected networks that accept 1D vectors, CNNs expect 2D (or 3D) inputs that preserve the spatial layout of the image. The MNIST data, when loaded, comes as 28x28 grayscale images, which is perfect — but we still need to:

- Normalize pixel values to [0, 1]
- Add a channel dimension (for grayscale, it's 1)
- Convert labels to one-hot encoding

2.1 — Code: Loading and Preprocessing MNIST

```
import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load MNIST dataset
data = mnist.load_data()
(X_train, y_train), (X_test, y_test) = data

# Reshape to (batch_size, height, width, channels)
X_train = X_train.reshape(-1, 28, 28, 1).astype("float32") / 255.0
X_test  = X_test.reshape(-1, 28, 28, 1).astype("float32") / 255.0

# One-hot encode labels
y_train = to_categorical(y_train, 10)
y_test  = to_categorical(y_test, 10)

# Check shapes
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
```

2.2 — Why Add a Channel Dimension?

In deep learning frameworks like TensorFlow/Keras, the expected input shape for a grayscale image is (`height`, `width`, `channels`). For MNIST, the images are 28x28 pixels, and each pixel has one intensity value — so we add the final channel dimension: 1.

2.3 — Why Normalize Pixel Values?

Most neural networks perform better when the input features are normalized. MNIST pixel values range from 0 to 255. We scale them to the [0, 1] range to:

- Improve convergence during training
- Reduce numerical instability
- Keep gradients in a reasonable range

2.4 — One-Hot Encoding for Labels

Since the output layer of our CNN will have 10 units (for digits 0–9) with a softmax activation, we need the labels in one-hot format. For example, label `3` becomes:

```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

This format aligns with the cross-entropy loss function we'll use later.

Next step: *Now that our data is ready, we'll design the architecture of our CNN step-by-step. Let's go to Step 3.*

Step 3 — Designing the CNN Architecture

Let's now define the architecture of our convolutional neural network. We will keep it minimal yet expressive — powerful enough to perform well on MNIST while staying didactic.

3.1 — Layers Overview

Our CNN will include the following components:

1. **Input layer:** Accepts 28x28x1 images
2. **Convolutional layer:** Applies 32 filters of size 3x3 with ReLU activation
3. **Pooling layer:** 2x2 max pooling to reduce dimensionality
4. **Second convolution + pooling:** To increase representational power
5. **Flatten layer:** Transforms 3D feature maps into 1D vector
6. **Dense layer:** Fully connected with 128 neurons and ReLU
7. **Output layer:** 10 neurons (one per class) with softmax

3.2 — Code: CNN Model Definition

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(filters=64, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

model.summary()
```

3.3 — Parameter Count and Insights

Each convolutional layer has a learnable set of filters (or kernels). For example:

- The first Conv2D layer with 32 filters of size 3x3 and 1 input channel has $32\times(3\times3\times1+1) = 320$ parameters.
- The second Conv2D layer with 64 filters and 32 input channels has $64 \times (3 \times 3 \times 32 + 1) = 18{,}496$ parameters.

The use of shared filters (convolutions) dramatically reduces the number of parameters compared to a dense layer.

Next step: *Now that our architecture is built, we will compile the model by choosing the optimizer, loss function, and metrics. That's Step 4.*

Step 4 — Compiling the CNN Model

Once the architecture is defined, we need to configure how the model will learn. This includes:

- **Loss function:** Quantifies how wrong the model is.
- **Optimizer:** Updates weights using gradients.
- **Metrics:** Additional indicators to track performance.

4.1 — Code: Compile the Model

```
model.compile(
    loss='categorical_crossentropy',       # Because labels are one-hot encoded
    optimizer='adam',                      # Adaptive learning rate optimizer
    metrics=['accuracy']                   # Track classification accuracy
)
```

4.2 — Why Categorical Crossentropy?

For multi-class classification with one-hot encoded labels, categorical crossentropy is the standard loss. It is defined as:

$$\mathcal{L} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where y_i is the true label (one-hot) and $\hat{y}_i$ is the predicted probability from the softmax output.

4.3 — Why Adam Optimizer?

`Adam` combines the best of `SGD with momentum` and `RMSProp`. It adapts learning rates per parameter and accelerates convergence. It works well out-of-the-box for most problems.

4.4 — Metric: Accuracy

Accuracy gives an interpretable signal: what proportion of the samples are classified correctly. It is not used in backpropagation — it's purely a reporting metric.

Next step: *The model is now ready to train. Let's fit it to the data and watch the learning process. That's Step 5.*