Guide 2 — Multilayer Perceptron

Teaching a Network to Think: MLP on MNIST

Written by: Pierre Chambet
Date: June 2025

Quote of the Journey

"All models are wrong. Some are useful." — George Box

Introduction

This is where it truly begins.

In this guide, we build our first complete neural network — from input to output, from pixels to predictions. Not a toy. Not a metaphor. A real model, trained on real data, evaluated with real metrics.

The dataset is MNIST — a collection of handwritten digits. Famous, clean, iconic. But this guide isn't about digits. It's about modeling: building a system that transforms observations into decisions.

We're not yet thinking in space. We're thinking in numbers. We flatten. We connect. We optimize. We build a mental model of how learning emerges through weight updates and loss gradients.

This guide is your entry point to function approximation. You'll understand how layers, activations, and loss functions come together to form a decision-making machine.

Let's begin.

Roadmap — MLP on MNIST

The Journey Ahead

This roadmap captures the key conceptual milestones in building your first neural network. Each step corresponds to a foundational block in your understanding of deep learning — not just how it works, but why.

1. Loading the Data

- **Objective:** Access and inspect the raw MNIST images.
- **Skills:** Use `keras.datasets`, visualize images, verify shapes and labels.
- **Concepts:** Image tensors, data splits, pixel distributions.

2. Preparing the Inputs

- **Objective:** Transform raw images into a form the model can understand.
- **Steps:**
  - Normalize pixel values to [0, 1]
  - Flatten 2D images to 1D vectors
  - Convert labels to one-hot encoded vectors
- **Why it matters:** Input stability is key to efficient learning.

3. Designing the MLP Architecture

- **Objective:** Create a basic feedforward neural network.
- **Structure:** Input layer → Hidden layers → Output layer
- **Choices:** Number of units, activation functions, initialization
- **Perspective:** The MLP as a universal function approximator.

4. Loss Functions and Softmax

- **Objective:** Measure how wrong the model is.
- **Topics:** Softmax for probabilities, crossentropy for loss
- **Mathematical Clarity:** Why their combination simplifies gradients and improves stability.

5. Training the Model

- **Objective:** Use gradient descent to minimize the loss.
- **Tools:** `model.compile()`, `fit()`, optimizers, learning rate
- **Outcomes:** Training/validation loss and accuracy curves

6. Evaluating Generalization

- **Objective:** Test the model on unseen data.
- **Checks:** Overfitting, underfitting, final accuracy
- **Reflection:** What the model learned — and what it didn't.

7. Understanding the Limits of MLPs

- **Objective:** Realize what's missing.
- **Key Insight:** The model sees pixels, not patterns. It has no concept of space.
- **What's next:** CNNs — networks that learn to see.

1. Loading and Exploring the MNIST Dataset

Before we even think about training a model, we must start with the most basic, yet most overlooked step: **understanding our data**.

MNIST is not just "some images of digits". It's a well-curated dataset of **70,000 images** of handwritten numbers — split into **60,000 for training** and **10,000 for testing**.

Each image:

- is grayscale,
- has a fixed size of **28 × 28 pixels**,
- represents one digit from **0 to 9**.

Load it — don't just trust it

We'll use the built-in datasets from libraries like `tensorflow.keras.datasets` or `torchvision.datasets`. These are not magical imports — they are curated loaders that return structured NumPy arrays.

In Python (TensorFlow):

```
from tensorflow.keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

This gives us:

- `X_train.shape = (60000, 28, 28)` — training images
- `y_train.shape = (60000,)` — training labels
- `X_test.shape = (10000, 28, 28)` — test images
- `y_test.shape = (10000,)` — test labels

Each image is stored as a 2D array — not a flattened vector. That's important.

Visualize what you're dealing with

Let's take a look at a few samples.

In Python:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 2))
for i in range(10):
    plt.subplot(1, 10, i+1)
    plt.imshow(X_train[i], cmap="gray")
    plt.title(str(y_train[i]))
    plt.axis("off")
plt.show()
```

Already, we can tell: some digits are clean. Some are tilted. Some are thick. Some are... a little ugly.

And that's the point.

This dataset reflects the messiness of human handwriting — but in a clean, structured format. That makes it ideal to study **both robustness and performance**.

Distribution of labels

Is the dataset balanced? Let's count the digits.

```
import numpy as np

unique, counts = np.unique(y_train, return_counts=True)
print(dict(zip(unique, counts)))
```

You should get something close to:

```
{0: 5923, 1: 6742, 2: 5958, ..., 9: 5949}
```

That's quite balanced — not perfect, but close enough for fair training.

We've now loaded and visualized the data. We've confirmed its shape, its structure, and its diversity.

Now, let's move to the next step: **preprocessing the inputs to make them ready for our models**.

2. Preparing the Input: From Pixels to Meaning

Before we ask a machine to learn, we must first learn how to speak its language.

Raw data — as beautiful and structured as MNIST may be — is not yet suitable for training. A model needs inputs that are scaled, shaped, and semantically meaningful.

In this section, we take our first act of care: we prepare the data.

2.1. Normalize the pixel values

Each image in MNIST is composed of 784 pixels, stored as integers between `0` (black) and `255` (white). These values are meaningful for our eyes — but not for gradient descent.

Large input values lead to large gradients. Large gradients lead to instability.

So we rescale every pixel to the range $[0, 1]$. Not based on theory — but on reality:

$$x_{\text{normalized}} = \frac{x}{\max(X_{\text{train}})}$$

```
X_train = X_train / X_train.max()
X_test  = X_test  / X_test.max()
```

This isn't just a trick — it's a foundational step in deep learning. Neural networks operate best in a numerically stable regime. Normalization ensures just that.

Respect the scale of things. Models do.

2.2. Flatten the images (for now)

Each MNIST image is a 28×28 grid — a spatial structure. But the first model we'll build doesn't yet understand space. It only understands vectors.

Enter the MLP: Multi-Layer Perceptron. An MLP is the most fundamental type of neural network. Remember, we built one from scratch in the first guide. It's a **fully connected network** :

- It takes a **flat vector** as input,
- Applies a series of **linear transformations** (remember the neuron/layer from guide 1) and **non-linearities** (remember activation functions),
- And produces a vector of **class scores or probabilities**.

So let's flatten our images to make them compatible with the MLP as a flat vector. We start with the original shapes:

```
print(X_train.shape)  # (60000, 28, 28)
print(X_test.shape)   # (10000, 28, 28)
```

To be clear:

```
X_train.shape[0] = 60000  # Number of training images
X_train.shape[1] = 28     # Height of each image
X_train.shape[2] = 28     # Width of each image
```

But our first model — a dense neural network, or MLP — expects each image as a flat vector. So we reshape the data like this:

```
X_train_flat = X_train.reshape(X_train.shape[0], X_train.shape[1] * X_train.shape[2])
X_test_flat  = X_test.reshape(X_test.shape[0], X_test.shape[1] * X_test.shape[2])
```

This gives:

- `X_train_flat.shape = (60000, 784)`
- `X_test_flat.shape = (10000, 784)`

Now, each image is a flat vector of 784 pixels.

Each row becomes one image. Each column, one pixel.

This is not a loss — it's a design choice. Flattening discards spatial locality, but preserves pixel content — which is enough for a baseline.

We flatten the image — not the information.

2.3. Encode the labels — From Numbers to Meaning

The labels - or ground truth - in MNIST are integers from `0` to `9`. They tell us which digit each image represents.

For example, let's display the first few labels of the training set:

```
print(y_train[:5])   # Output: [5 0 4 1 9]
```

This is perfect for humans — but not enough for our model.

Why? Because the model won't just output a single number. It will produce a **vector of probabilities**, like:

$$\hat{y} = [0.01, 0.01, 0.02, \mathbf{0.95}, 0.00, \dots, 0.00]$$

This vector tells us how confident the model is for each class. To compare this prediction with the true answer, we need the label to have the same shape.

In other words, we are building a model that will output **a distribution over classes** — not a single number.

So instead of comparing:

$$\texttt{label} = 3 \quad \text{vs.} \quad \hat{y} = 3$$

We compare:

$$y = [0, 0, 0, \mathbf{1}, 0, ..., 0] \quad \text{vs.} \quad \hat{y} = [0.01, 0.01, 0.02, \mathbf{0.95}, ..., 0.00]$$

This requires three new concepts:

1. Softmax — how to turn scores into probabilities : For the last activation function, the one at the final layer, the model outputs 10 raw scores (logits). We apply the **softmax function** to transform them into a valid probability distribution:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}}$$

This tells us how confident the model is for each class. We expect the highest probability corresponds to the true digit.

2. One-hot encoding — how to shape certainty : We encode the true label as a one-hot vector — a 10-dimensional vector with a single 1 in the correct position.

$$\text{Ex : True label for 3} \quad \Rightarrow \quad [0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0]$$

This matches the shape of the model's output — and allows us to compute the error.

3. Cross-entropy — the cost of being wrong : The **cross-entropy loss** compares the model's softmax output to the one-hot encoded truth. It penalizes confident wrong predictions more heavily than hesitant ones.

$$\mathcal{L} = -\sum_{i=1}^{10} y_i \log(\hat{y}_i)$$

One-hot encoding in Code : (We code this function from scratch in the Google Colab notebook, but here's how to do it with TensorFlow/Keras.)

```
from tensorflow.keras.utils import to_categorical

# Convert labels to one-hot encoded format
y_train_cat = to_categorical(y_train, num_classes=10)
y_test_cat  = to_categorical(y_test, num_classes=10)

# Check the result
```

```
print(y_train_cat[:2])
# Output:
# [[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
#  [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Each row now has:

- 10 values (one per class),
- A single 1 indicating the correct digit.

From a number... to a structured certainty.

Note. If we use a different loss function — like `sparse_categorical_crossentropy` — we can keep the labels as integers. But in this guide, we'll use one-hot encoding explicitly, so we can visualize and understand everything the model is doing.

Summary: A clean dataset, ready to learn

We now have:

- Normalized pixel values from $[0, 255]$ to $[0, 1]$,
- Flattened images of shape `(60000, 784)`,
- Labels encoded as one-hot vectors of shape `(60000, 10)`.

The data is clean. The format is correct. The inputs are meaningful.

It's time to build our first neural network — and teach it how to read.

3. Our First MLP — Building the Baseline

Let's put our theory into practice.

We now have:

- Clean, normalized input data,
- Flattened images, ready to be processed as vectors,
- One-hot encoded labels that define what is true.

Our goal is simple: build a model that takes one image — and predicts which digit it is.

3.1. Architecture — From pixels to predictions

We'll construct a very simple Multi-Layer Perceptron with the following architecture:

$$784 \to 128 \to 64 \to 10$$

In words:

- The input layer has 784 nodes — one for each pixel.
- The first hidden layer has 128 neurons, each connected to all inputs.
- The second hidden layer has 64 neurons.
- The output layer has 10 neurons — one per digit.

Each hidden layer is followed by a **ReLU activation function**, and the output layer ends with a **softmax activation function** to produce probabilities.

3.2. Building the Model — Layer by Layer

We'll now define our first neural network using Keras — a high-level, intuitive API built on top of TensorFlow.

Here is the full definition of our Multi-Layer Perceptron:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(128, activation='relu', input_shape=(784,)),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

What's happening here? Let's break it down:

- `Sequential([...])` Creates a linear stack of layers — the output of one becomes the input of the next. Perfect for a feedforward network like an MLP.
- `Dense(128, activation='relu')` Adds a **fully connected layer** with 128 neurons. Each neuron computes a weighted sum of all inputs and applies a ReLU activation:

$$\mathrm{ReLU}(x) = \max(0, x)$$

  ReLU introduces non-linearity — without it, the network would collapse into a simple linear transformation.
- `input_shape=(784,)` Specifies the shape of each input vector. Here, each image is a flat vector of 784 pixels. This argument is only needed for the first layer.
- `Dense(64, activation='relu')` A second hidden layer, smaller — 64 neurons. It extracts higher-level features from the previous representation.
- `Dense(10, activation='softmax')` The output layer. It has 10 neurons — one for each digit (0–9). The softmax activation transforms the raw outputs (”logits”) into a **probability distribution** over the 10 classes.

Let's look at the architecture:

```
model.summary()
```

Typical output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               100480
 dense_1 (Dense)             (None, 64)                8256
 dense_2 (Dense)             (None, 10)                650
=================================================================
Total params: 109,386
Trainable params: 109,386
Non-trainable params: 0
```

How do we read this?

- `Output Shape` shows the dimension of each layer's output.
- `Param #` is the number of **trainable parameters** in the layer:

  $$\text{Parameters} = (\text{input size}) \times (\text{output size}) + (\text{biases})$$

  For example, the first layer has:

  $$784 \times 128 + 128 = 100{,}480$$

  Of course, we have one weight for each connection, plus one bias for each neuron.
- `Trainable params` shows the number of **trainable parameters** — the weights and biases across all layers. This is the total number of values the model will update during training.

  This is your model's skeleton. Each number here tells a story — about capacity, flexibility, and complexity.

That's it. Three layers. Fully connected. We've defined the computation that transforms input pixels into confidence scores for each digit.

3.3. Compilation — Preparing for training

Before training, we need to tell the model:

- **How to measure the error** → using the `categorical_crossentropy` loss.
- **How to optimize the weights** → using the `adam` optimizer.
- **How to track performance** → using the `accuracy` metric.

```
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

The `adam` optimizer is a modern variant of gradient descent. It adapts the learning rate for each parameter over time — which makes it fast, stable, and widely used.

3.3 bis — Why do we choose the loss function `categorical_crossentropy`?

When training a neural network to classify images—such as identifying handwritten digits in MNIST—it's not enough to simply make predictions. The model needs to **learn from its mistakes**, and this is exactly the role of a **loss function**.

So:

- Why do we choose `categorical_crossentropy`?
- What does it really measure?
- And what makes it *mathematically special* compared to other loss functions?

Let's unfold the answer.

What is `categorical_crossentropy`?

It's a function that measures the **discrepancy** between two probability distributions:

1. The **true label** y, represented as a **one-hot vector**, e.g. $[0, 0, 1, 0, 0]$ for class 2.
2. The **predicted probabilities** $\hat{y}$, usually the output of a softmax, e.g. $[0.01, 0.04, 0.89, 0.03, 0.03]$.

The loss is computed as:

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

But since only one $y_i = 1$ (true class), this simplifies to:

$$\mathcal{L}(y, \hat{y}) = -\log(\hat{y}_{\text{true class}})$$

It penalizes the model only based on how confident it is about the correct class.

Why is this loss meaningful?

Suppose the true class is 2:

Predicted probability for class 2	Loss
0.95	$-\log(0.95) \approx 0.051$ (good)
0.50	$-\log(0.50) = 0.693$ (uncertain)
0.01	$-\log(0.01) = 4.605$ (bad)

The lower the predicted probability for the correct class, the higher the loss. The logarithm naturally **dampens overconfidence** and **emphasizes rare errors**.

What makes `softmax + categorical_crossentropy` so powerful?

Here's the magic: when we use a softmax activation function in the final layer (output layer), followed by categorical cross-entropy, the gradient of the loss simplifies beautifully. And where do we need gradients? During backpropagation, to update the model's weights. You see where I'm going?

Let:

- z_i be the logit (score before softmax) for class i,
- $\hat{y}_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$ be the softmax output,
- $y_i \in \{0, 1\}$ be the one-hot label.

Then, the derivative of the loss with respect to the input logit is:

$$\boxed{\frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i - y_i}$$

This result is **astonishingly elegant**:

- The gradient is just the difference between the predicted probability and the ground truth.
- No exponentials, no logs, no complicated expressions during backpropagation.
- This makes the combination **fast**, **stable**, and **very effective**.

What if we used something else?

Output Layer	Loss Function	Recommended?
`softmax`	`categorical_crossentropy`	Yes — Standard choice
`softmax`	`mean_squared_error`	No — Poor gradients
`sigmoid`	`binary_crossentropy`	Yes — Multi-label tasks
None (logits)	`SparseCategoricalCrossentropy(from_logits=True)`	Yes — But requires care

Using `mse` with softmax, for instance, results in **weaker gradients** and slower learning.

But this is just the tip of the iceberg.

What's the best loss for your problem? Should you use sigmoid, softmax, or neither? What happens under the hood when you combine activation functions with different loss objectives? These are critical design questions that few tutorials ever dive into with precision.

That's why I'm preparing a full guide — a dedicated exploration of loss functions, activation functions, and the **subtle alchemy behind their combinations**. To date, I haven't encountered a guide that demystifies this topic properly. So, I'll write one.

For now, let's stay focused on the golden standard—the battle-tested duo:

`softmax` output + `categorical_crossentropy` loss.

Summary: An optimal match for classification

The combination of `softmax` and `categorical_crossentropy` is not just a convention—it is a **mathematically optimized pipeline** for multiclass classification.

- The softmax layer produces a **true probability distribution** from unbounded scores.
- The categorical cross-entropy loss measures **how well that distribution aligns with the true label**.
- Their gradients align perfectly:
$$\frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i - y_i$$
which is computationally elegant, numerically stable, and easy to implement.
- This yields **faster convergence**, better generalization, and smoother optimization trajectories.

This synergy is why almost every modern classification pipeline builds on this foundation. And now — you understand why.

Maybe some of you are thinking : how do we found this easy gradient? *Let's prove it.*

Heads-up — No Magic Here

Warning: math zone ahead. Just a few pages, nothing terrifying — but no illusions either. We're about to unpack what really happens when `softmax` meets `crossentropy`. No black box, no mysticism. Just math, truth, and clarity.
It's the good kind of pain. You can do this. Breathe in. Stay sharp. Let's go decode the machine.

3.3 ter — Deriving the gradient: from softmax to cross-entropy

Let us now rigorously derive the famous result:

$$\boxed{\frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i - y_i}$$

which connects the softmax output $\hat{y}_i$ and the categorical cross-entropy loss $\mathcal{L}$.

This result is not just elegant—it's *practical.* It leads to extremely efficient implementations of the backpropagation step when training neural networks for multiclass classification.

Step 1 — Define the softmax and cross-entropy

Let the output vector of the neural network before activation be $\mathbf{z} = (z_1, \ldots, z_C)$ (called the **logits**), for C classes.

The softmax function transforms these scores into a probability distribution:

$$\hat{y}_j = \frac{e^{z_j}}{\sum_{k=1}^{C} e^{z_k}} = \frac{e^{z_j}}{S} \quad \text{where } S = \sum_{k=1}^{C} e^{z_k}$$

Let the true label be encoded as a one-hot vector $y = (y_1, \ldots, y_C)$, where $y_i = 1$ if class i is correct, and $y_j = 0$ for $j \neq i$.

Then, the categorical cross-entropy loss is defined as:

$$\mathcal{L}(y, \hat{y}) = -\sum_{j=1}^{C} y_j \log(\hat{y}_j)$$

Since y is one-hot, this simplifies to:

$$\mathcal{L} = -\log(\hat{y}_t) \quad \text{where } t \text{ is the true class.}$$

Step 2 — Compute the gradient using the chain rule

Our goal is to compute the derivative of the loss with respect to the logits z_i:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \sum_{j=1}^{C} \frac{\partial \mathcal{L}}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_i}$$

Let's compute each term separately.

(a) Derivative of loss w.r.t. softmax output: Our goal is to compute:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_j}$$

Derivative with respect to $\hat{y}_j$

We now differentiate the loss expression with respect to $\hat{y}_j$, for all $j \in \{1, \ldots, C\}$.

Case 1: $j = t$ Only the term $-\log(\hat{y}_j)$ appears in the loss. Thus:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_j} = \frac{d}{d\hat{y}_j}\left(-\log(\hat{y}_j)\right) = -\frac{1}{\hat{y}_j} \quad \text{since } y_j = 1$$

Case 2: $j \neq t$ In this case, the term $-y_j \log(\hat{y}_j)$ does not appear in the loss since $y_j = 0$. Hence:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_j} = 0$$

Unified expression

These two cases can be unified into a single elegant expression:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \hat{y}_j} = -\frac{y_j}{\hat{y}_j}}$$

(b) Derivative of softmax output w.r.t. logits: We want to compute the partial derivative:

$$\frac{\partial \hat{y}_j}{\partial z_i}$$

We distinguish two cases:

Case 1: $i = j$

We differentiate $\hat{y}_j$ with respect to its own input z_j.

Let us rewrite:

$$\hat{y}_j = \frac{e^{z_j}}{S} \quad \text{with} \quad S = \sum_{k=1}^{C} e^{z_k}$$

We apply the quotient rule:

$$\frac{\partial \hat{y}_j}{\partial z_j} = \frac{\frac{d}{dz_j}(e^{z_j}) \cdot S - e^{z_j} \cdot \frac{dS}{dz_j}}{S^2}$$

We compute each derivative:

$$\frac{d}{dz_j}(e^{z_j}) = e^{z_j}$$

$$\frac{dS}{dz_j} = \frac{d}{dz_j}\left(\sum_{k=1}^{C} e^{z_k}\right) = e^{z_j}$$

Therefore:

$$\frac{\partial \hat{y}_j}{\partial z_j} = \frac{e^{z_j} \cdot S - e^{z_j} \cdot e^{z_j}}{S^2} = \frac{e^{z_j}(S - e^{z_j})}{S^2}$$

Recall that:

$$\hat{y}_j = \frac{e^{z_j}}{S} \Rightarrow e^{z_j} = \hat{y}_j \cdot S$$

Substitute into the expression:

$$\frac{\partial \hat{y}_j}{\partial z_j} = \frac{\hat{y}_j \cdot S(S - \hat{y}_j \cdot S)}{S^2} = \hat{y}_j \cdot (1 - \hat{y}_j)$$

$$\boxed{\frac{\partial \hat{y}_j}{\partial z_j} = \hat{y}_j(1 - \hat{y}_j)}$$

Case 2: $i \neq j$

We now differentiate $\hat{y}_j$ with respect to z_i, where $i \neq j$. In this case:

- The numerator e^{z_j} is independent of z_i, so its derivative is 0.
- The denominator S still depends on z_i, since z_i appears in the sum.

We compute:

$$\frac{\partial \hat{y}_j}{\partial z_i} = \frac{\partial}{\partial z_i}\left(\frac{e^{z_j}}{S}\right) = -\frac{e^{z_j} \cdot \frac{\partial S}{\partial z_i}}{S^2}$$

Now, $\frac{\partial S}{\partial z_i} = e^{z_i}$. So:

$$\frac{\partial \hat{y}_j}{\partial z_i} = -\frac{e^{z_j} \cdot e^{z_i}}{S^2}$$

Again, using $\hat{y}_j = \frac{e^{z_j}}{S}$ and $\hat{y}_i = \frac{e^{z_i}}{S}$, we obtain:

$$\frac{\partial \hat{y}_j}{\partial z_i} = -\hat{y}_j \cdot \hat{y}_i$$

$$\boxed{\frac{\partial \hat{y}_j}{\partial z_i} = -\hat{y}_j \hat{y}_i \quad \text{for } i \neq j}$$

Unified expression

$$\frac{\partial \hat{y}_j}{\partial z_i} = \begin{cases} \hat{y}_j(1-\hat{y}_j) & \text{if } i = j \\ -\hat{y}_j \hat{y}_i & \text{if } i \neq j \end{cases}$$

This can also be written more compactly using the Kronecker delta δ_{ij}:

$$\boxed{\frac{\partial \hat{y}_j}{\partial z_i} = \hat{y}_j(\delta_{ij} - \hat{y}_i)}$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise.

Step 3 — Combine the two parts

Plug everything into the chain rule:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \sum_{j=1}^{C}\left(-\frac{y_j}{\hat{y}_j} \cdot \hat{y}_j(\delta_{ij} - \hat{y}_i)\right) = -\sum_{j=1}^{C} y_j(\delta_{ij} - \hat{y}_i)$$

We can split the sum:

$$= -\left(y_i(1-\hat{y}_i) + \sum_{j\neq i} y_j(-\hat{y}_i)\right) = -y_i + y_i\hat{y}_i + \hat{y}_i \sum_{j \neq i} y_j$$

But since y is one-hot encoded:

$$\sum_{j\neq i} y_j = 1 - y_i$$

Then:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i(1-y_i) - y_i(1-\hat{y}_i) = \hat{y}_i - y_i$$

Final Result

$$\boxed{\frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i - y_i}$$

This is the key identity that makes softmax + crossentropy so efficient to train. During backpropagation, we don't need to manually differentiate through the softmax and the log. It all collapses into a simple subtraction.

Conclusion: this result is the mathematical foundation behind modern classification models. It's efficient, elegant, and widely used.

3.4. Training the model

We now train the model on the data:

```
history = model.fit(
    X_train_flat, y_train_cat,
    epochs=10,
    batch_size=32,
    validation_split=0.1
)
```

Here's what happens:

- The training set is split into 90% for training and 10% for validation.
- The model goes through the data 10 times (**epochs**).
- At each step, it updates the weights to reduce the loss — learning from its own errors.

And just like that — the model learns to recognize digits. From raw pixels. From scratch.

3.5. Visualizing the learning process

Let's plot the training and validation curves:

```
import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='Training loss')
plt.plot(history.history['val_loss'], label='Validation loss')
plt.title("Loss over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()
```

And for accuracy:

```
plt.plot(history.history['accuracy'], label='Training acc')
plt.plot(history.history['val_accuracy'], label='Validation acc')
plt.title("Accuracy over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.show()
```

ICI PHOTO DES COURBES

If the curves decrease steadily (for loss) and rise (for accuracy), your model is learning. If not, it's either stuck — or overfitting. We'll talk about that later.

3.6. Evaluate on the test set

Let's now measure how well the model performs on unseen data:

```
test_loss, test_acc = model.evaluate(X_test_flat, y_test_cat)
print(f"Test accuracy: {test_acc:.4f}")
```

If everything went well, you should get a test accuracy around **96%** — not bad for a first attempt.

You just built your first digit classifier — from raw images to reliable predictions.

3.7 — Beyond Accuracy: Error Analysis and Confusion Matrix

Until now, we've focused on accuracy — a single number, often celebrated, sometimes misleading.

But real understanding begins when we ask: *Where did the model go wrong?*

This is not just a debugging step — it's a moment of learning. What we're about to do is called **error analysis**.

Let's unpack the model's behavior class by class.

Confusion Matrix — Making Errors Visible

A **confusion matrix** is a square grid where:

- Rows represent the **true labels**.
- Columns represent the **predicted labels**.
- Each cell (i, j) counts how many samples of class i were predicted as class j.

The diagonal shows the number of correct predictions. Off-diagonal values are mistakes — and that's where the insights live.

Here's how to compute and display it:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np

# Predict on the test set
y_pred_probs = model.predict(X_test_flat)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test_cat, axis=1)

# Compute confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Display
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap="Blues", values_format='d')
```

This matrix gives you a powerful visual sense of what digits are getting confused — and why.

- Do 4s get misclassified as 9s?
- Do 3s and 5s look similar to the network?
- Are errors symmetric?

Now we move from accuracy to accountability.

But don't get too comfortable. This model has a fundamental flaw — it doesn't know that images have shape. It treats each pixel as independent. It sees no edges, no curves, no local structure.

And that's where convolutional neural networks come in.

Next: From pixels to perception — let's teach the model how to see.