Machine Learning Introduction

Part III: MNIST Classification with PyTorch – Softmax, ReLU and Validation

Matthieu Gurrisi

February 2026

In Part I and Part II, we built and trained neural networks *from scratch* in NumPy in order to fully understand forward propagation, gradient descent and backpropagation. In this part, we move to a realistic multi-class task: recognizing handwritten digits from MNIST. Since we now master the core mechanics, we will use **PyTorch** to implement the same ideas more efficiently, and focus on preprocessing and training methodology (activations, mini-batches, validation, early stopping).

The associated notebook/code is available here: Part III code repository.

1 MNIST and the Goal of Part III

MNIST is a classic benchmark dataset for image classification. Each sample is a grayscale image of size 28×28 representing a digit from 0 to 9. The goal is to learn a function that maps an image to one of the ten classes.

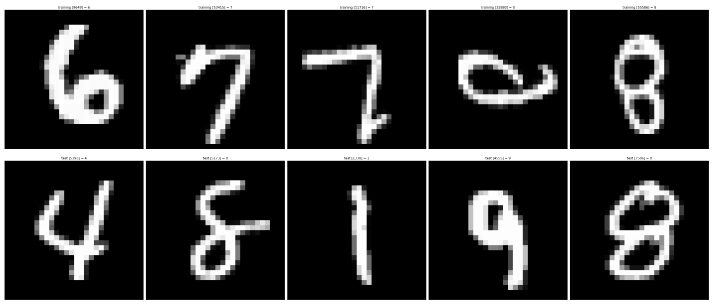

Figure 1: Examples from the MNIST dataset. Each image is 28×28 and belongs to one of 10 classes.

In Part II we performed binary classification. Here, we extend the problem to *multi-class classification*. The output is no longer a single probability: we want a probability distribution over the 10 digits.

2 Preprocessing: Flattening and Normalization

In Part II, inputs were already vectors in $\mathbb{R}^2$. Here, inputs are images. A simple multilayer perceptron (MLP) expects vector inputs, so we apply two standard preprocessing steps:

2.1 Flattening (Vectorization)

Each image is a matrix in $\mathbb{R}^{28\times 28}$. We reshape it into a vector in $\mathbb{R}^{784}$:

$$x \in \mathbb{R}^{28\times 28} \quad \longrightarrow \quad \mathrm{vec}(x) \in \mathbb{R}^{784}.$$

This operation does not change the information content; it only changes the representation so it can be fed to a fully-connected network.

2.2 Normalization

Raw pixel values typically lie in $\{0, 1, \ldots, 255\}$. If left unscaled, these large values can lead to unstable gradients and slow training. A common normalization is:

$$x \leftarrow \frac{x}{255},$$

which brings pixel values to $[0, 1]$.

Remark. For image datasets, this simple normalization is often sufficient for a first model. More advanced pipelines may also include mean/variance normalization, data augmentation, or whitening.

3 Why PyTorch?

In Part I and Part II, we implemented everything manually: forward pass, gradients and parameter updates. PyTorch provides:

- tensor operations optimized for CPU/GPU,
- automatic differentiation (`autograd`) to compute gradients,
- efficient data handling with `Dataset` and `DataLoader`,
- well-tested loss functions and optimizers.

This does not change the underlying mathematics: PyTorch simply automates and accelerates what we already derived by hand.

4 Model: MLP for Multi-Class Classification

We reuse the architecture from Part II (one hidden layer), but we adapt the output layer to handle 10 classes. The model produces *logits* (unnormalized scores) in $\mathbb{R}^{10}$, then a *softmax* converts them into probabilities.

4.1 Network Diagram

Figure 2 summarizes the pipeline:

- input image (28×28),
- flatten to a vector of size 784,
- hidden layer (sigmoid or ReLU),
- output layer of size 10,
- softmax probabilities.

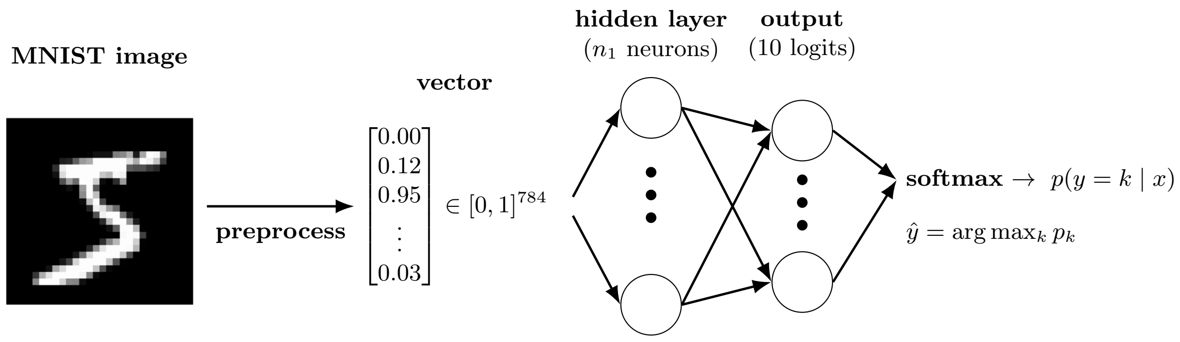
MNIST image
preprocess
vector
$\begin{bmatrix}0.00\\0.12\\0.95\\\vdots\\0.03\end{bmatrix} \in [0,1]^{784}$
hidden layer
(n_1 neurons)
output
(10 logits)
softmax $\to$ $p(y = k \mid x)$
$\hat{y} = \arg\max_k p_k$

Figure 2: MNIST classification pipeline: an image is preprocessed (normalized and flattened) into a vector in $[0, 1]^{784}$, then fed to a one-hidden-layer MLP producing 10 logits, which are converted into class probabilities by softmax.

5 Epochs, Mini-Batches and Why We Use Them

In Part I and Part II, we often used (full) batch gradient descent: each update used the entire training set. With MNIST, this becomes inefficient because the dataset is large.

5.1 Definitions

Let m be the number of training samples.

- A **mini-batch** is a small subset of the training set, of size B (e.g., $B = 32$).
- One **iteration** (or **step**) is one parameter update using one mini-batch.
- One **epoch** is one full pass over the training set.

Therefore, the number of steps per epoch is approximately:

$$\text{steps per epoch} \approx \left\lceil \frac{m}{B} \right\rceil .$$

5.2 Why mini-batches?

Mini-batches make training faster and more stable:

- computing gradients on smaller chunks is computationally efficient,
- noisy gradient estimates can help escape poor regions in the loss landscape,
- training scales naturally to large datasets and hardware accelerators.

6 Softmax: Extending Sigmoid to Multi-Class Outputs

In binary classification, the sigmoid maps one scalar score to one probability. For multi-class classification, we need probabilities that sum to 1 across $K = 10$ classes.

Given logits $z \in \mathbb{R}^{10}$, softmax is defined by:

$$\text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=1}^{10} e^{z_j}}, \qquad k = 1, \ldots, 10.$$

This produces a valid probability distribution:

$$\text{softmax}(z)_k \in (0, 1), \qquad \sum_{k=1}^{10} \text{softmax}(z)_k = 1.$$

Important practical note (PyTorch). In PyTorch, `CrossEntropyLoss` expects raw logits as input and internally applies the log-softmax. Therefore, we do *not* manually apply softmax inside the model during training.

7 ReLU vs Sigmoid in the Hidden Layer

We compare two models that only differ by the hidden activation:

$$\text{Sigmoid: } \sigma(z) = \frac{1}{1+e^{-z}}, \qquad \text{ReLU: } \mathrm{ReLU}(z) = \max(0, z).$$

7.1 Why ReLU is often preferred

ReLU is widely used because:

- it is simple and cheap to compute,
- it avoids saturation in the positive regime (unlike sigmoid),
- it often leads to faster training and better gradient flow in deeper networks.

In contrast, sigmoid can saturate for large $|z|$, producing very small gradients and slowing down learning. It can also lead to numerical issues if probabilities become extremely close to 0 or 1 when combined with log-loss. (Using PyTorch's stable losses mitigates this, but the optimization behavior still differs.)

8 Empirical Comparison: Sigmoid vs ReLU

We trained two MLPs with the same architecture and hyperparameters, changing only the hidden activation. The results clearly favor ReLU: training converges faster and reaches a higher accuracy in fewer epochs.

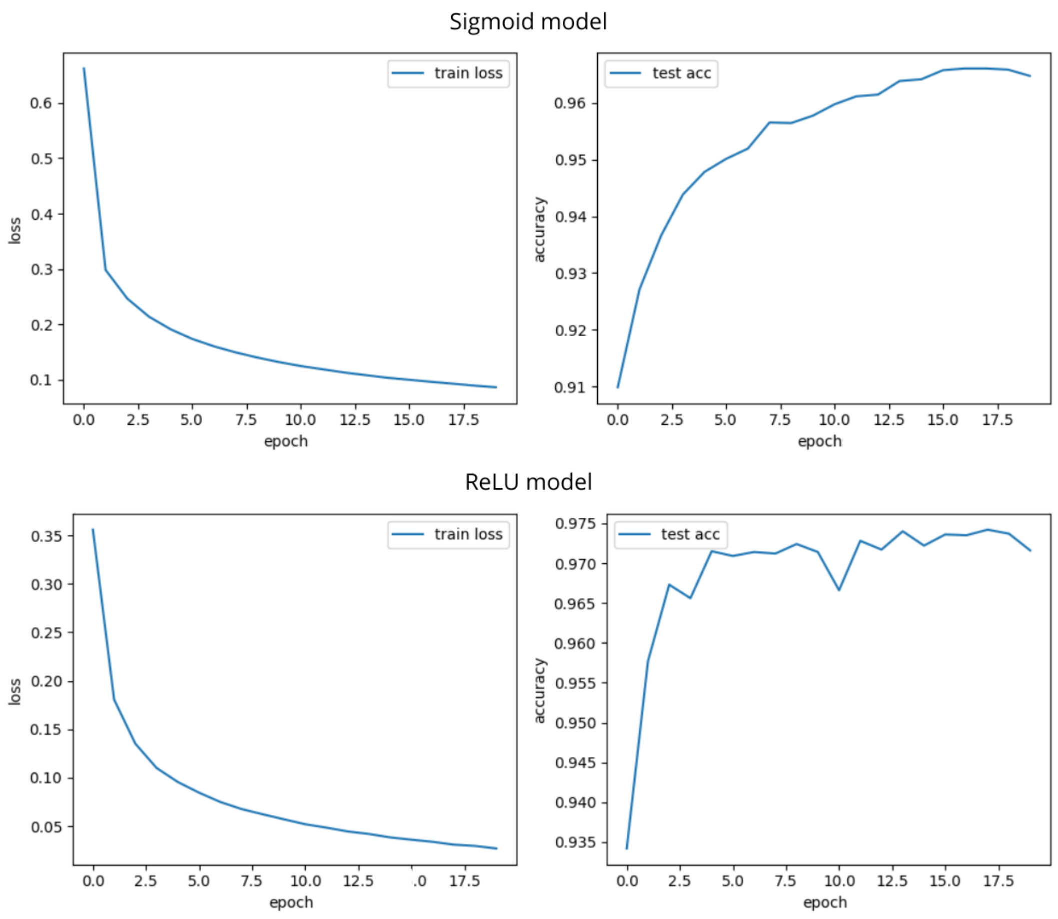

Figure 3: Comparison of the two models: sigmoid vs ReLU in the hidden layer.

Observation. ReLU reaches strong performance quickly (often within the first few epochs), which is convenient, but also raises the question of *overfitting*: continuing to train may improve training loss while harming generalization.

9 Validation Set and Early Stopping

To reduce overfitting, we introduce a **validation set**. The training set is split into:

$$\text{train set} \cup \text{validation set}.$$

During training, we monitor validation performance and stop when it stops improving.

9.1 Early Stopping

Early stopping works as follows:

- after each epoch, compute the validation loss,
- keep track of the best validation loss achieved,
- stop training if validation loss does not improve for a fixed number of epochs (patience),
- restore the best model parameters.

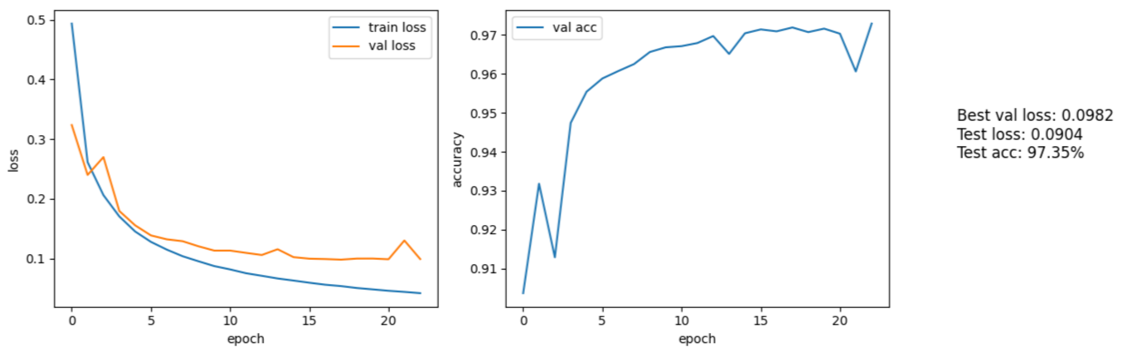

Figure 4: Early stopping: training is stopped when validation performance stops improving.

9.2 Why Results Can Slightly Drop

With early stopping, the final test accuracy may be slightly lower than the best score obtained without validation. This is mainly due to two factors:

- **Less training data.** The validation split reduces the number of samples used to fit the model.
- **Dependence on the split.** The chosen stopping point depends on the particular validation set: some splits are easier/harder than others.

9.3 Cross-Validation (Concept)

A common way to reduce dependence on a single validation split is **cross-validation**. The idea is to perform multiple train/validation splits (folds), train multiple times and average the results. This yields a more robust estimate of model performance and reduces variance due to a particular split. We do not implement cross-validation in the notebook (it is more costly for neural networks), but it remains an important concept in practical machine learning.

10 Conclusion: From a Single Neuron to MNIST

Across these three parts, we progressively built intuition and practical tools:

- **Part I:** logistic regression as a single neuron; log-loss from maximum likelihood; gradient descent.
- **Part II:** multilayer perceptron; nonlinear decision boundaries; backpropagation and vectorization.
- **Part III:** multi-class classification on MNIST; softmax and cross-entropy; ReLU vs sigmoid; training methodology with mini-batches, validation and early stopping.

At this stage, we can implement and train neural networks both from scratch (for understanding) and with modern libraries (for efficiency), while making informed choices about activations, loss functions and training strategies.