Machine Learning Introduction
Part II: Multilayer Perceptron – Nonlinear Binary Classification

Matthieu Gurrisi

February 2026

In this document, we extend Part I to a *multilayer perceptron* (MLP) in order to solve binary classification problems that are *not linearly separable*. We follow the same philosophy as before: build the model from scratch, understand the math, and connect each formula to an implementation.

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

1 Motivation: Beyond Linear Decision Boundaries

In Part I, logistic regression (a single neuron) produces a linear decision boundary. However, many datasets cannot be separated by a single line. A classic example is the *concentric circles* dataset: one class forms an inner circle and the other an outer ring. No straight line can separate them.

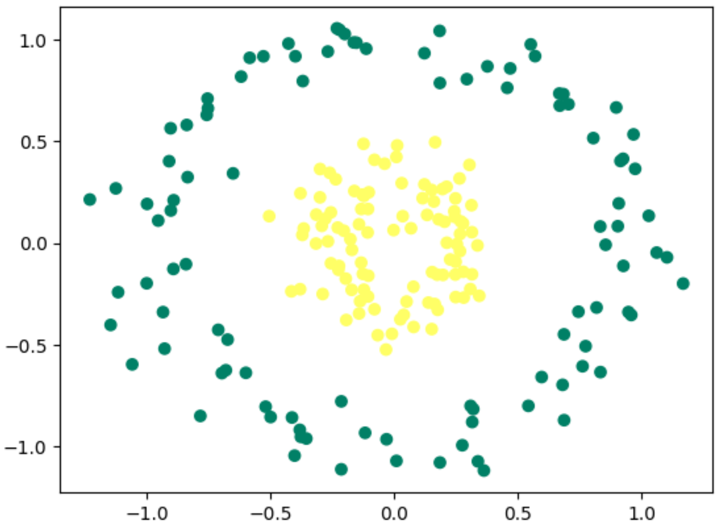

Figure 1: Example of a non-linearly separable dataset (concentric circles).

The idea of an MLP is to compose simple nonlinear transformations in order to create a flexible decision boundary in the original input space.

2 Model: A Two-Layer Neural Network

We consider an input $x \in \mathbb{R}^2$ and a network with:

- a **hidden layer** of n_1 neurons,
- an **output layer** of one neuron for binary classification.

2.1 Architecture

We denote the parameters by:

$$W^{[1]} \in \mathbb{R}^{n_1 \times 2}, \quad b^{[1]} \in \mathbb{R}^{n_1 \times 1}, \qquad W^{[2]} \in \mathbb{R}^{1 \times n_1}, \quad b^{[2]} \in \mathbb{R}^{1 \times 1}.$$

A schematic view is shown in Figure 2.

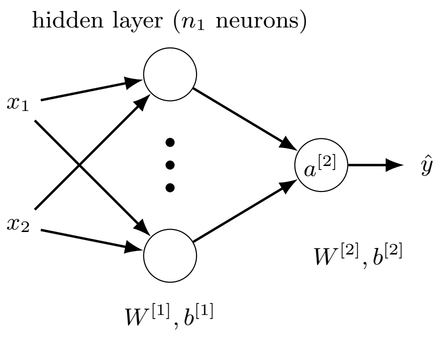

Figure 2: Two-layer MLP for binary classification (one hidden layer).

2.2 Forward Propagation

For a single example x, the forward pass is:

$$z^{[1]} = W^{[1]}x + b^{[1]}, \qquad a^{[1]} = \sigma\left(z^{[1]}\right),$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}, \qquad a^{[2]} = \sigma\left(z^{[2]}\right).$$

We interpret $a^{[2]}$ as the predicted probability:

$$P(Y = 1 \mid x) \approx a^{[2]}*$$

*Once again, we will see in the next section that this is not quite true.

Finally, a hard prediction is obtained by thresholding:

$$\hat{y} = \mathbf{1}_{\{a^{[2]} \geq 1/2\}}.$$

3 Vectorized Notation

We now consider m training samples stacked into a matrix

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \end{bmatrix} \in \mathbb{R}^{2 \times m}, \qquad Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} \in \mathbb{R}^{1 \times m}.$$

With this convention, forward propagation becomes:

$$Z^{[1]} = W^{[1]}X + b^{[1]}, \qquad A^{[1]} = \sigma\left(Z^{[1]}\right),$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}, \qquad A^{[2]} = \sigma\left(Z^{[2]}\right).$$

Here $b^{[1]}$ and $b^{[2]}$ are broadcasted across the m columns.

4 Loss Function (Log-Loss)

We use the same log-loss as in Part I:

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)}\log(a^{[2](i)}) + (1-y^{(i)})\log(1-a^{[2](i)})\Big),$$

where $a^{[2](i)}$ denotes the output activation for the i-th example.

5 Backpropagation

5.1 Backpropagation: the Chain Rule in Action

In Part I, computing gradients was relatively direct because the loss depended on a single affine score $z = w^\top x + b$. In a multilayer network, the situation is different: the output of each layer becomes the input of the next one. As a result, the loss $\mathcal{L}$ depends on the parameters of early layers only *indirectly*, through a chain of intermediate computations:

$$(W^{[1]}, b^{[1]}) \longrightarrow Z^{[1]} \longrightarrow A^{[1]} \longrightarrow Z^{[2]} \longrightarrow A^{[2]} \longrightarrow \mathcal{L}.$$

Training is still done with gradient descent: for each layer ℓ, we update

$$W^{[\ell]} \leftarrow W^{[\ell]} - \alpha\,\frac{\partial \mathcal{L}}{\partial W^{[\ell]}}, \qquad b^{[\ell]} \leftarrow b^{[\ell]} - \alpha\,\frac{\partial \mathcal{L}}{\partial b^{[\ell]}}.$$

So the practical task is clear: compute the gradients with respect to *all* parameters.

The difficulty is that these gradients are no longer "local". For example, the loss depends on $W^{[1]}$ only through its effect on $Z^{[1]}$, then $A^{[1]}$, then $Z^{[2]}$, etc. This is exactly the situation where the *chain rule* is needed. For instance, at the output layer one naturally obtains a product of local derivatives such as

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial A^{[2]}}\cdot\frac{\partial A^{[2]}}{\partial Z^{[2]}}\cdot\frac{\partial Z^{[2]}}{\partial W^{[2]}},$$

and for the first layer the chain is even longer because variations in $W^{[1]}$ must travel through all subsequent quantities before affecting $\mathcal{L}$.

Backpropagation is simply a systematic and efficient way to apply the chain rule along this computation graph. The key idea is to start from the end (the output layer), compute how the loss reacts to changes in the output, and then *propagate* this information *backward* through the network.

To make these repeated chain-rule patterns readable (and reusable), we introduce the intermediate quantities

$$dZ^{[\ell]} := \frac{\partial \mathcal{L}}{\partial Z^{[\ell]}} \qquad (\ell = 1, 2),$$

which measure how sensitive the loss is to the *pre-activation* $Z^{[\ell]}$ at each layer.

Why is this natural? Because once $dZ^{[\ell]}$ is known, the gradients of the parameters at that layer follow immediately from the simple derivatives of the affine transformation $Z^{[\ell]} = W^{[\ell]}A^{[\ell-1]} + b^{[\ell]}$ (exactly the same spirit as Part I). In other words, $dZ^{[\ell]}$ is the compact object that *carries* all the dependence of $\mathcal{L}$ on the layer ℓ computations.

This explains the structure of the next subsections: we first derive $dZ^{[2]}$ at the output (closest to the loss), then use it to obtain $dZ^{[1]}$ by propagating backward. With $dZ^{[2]}$ and $dZ^{[1]}$ in hand, all parameter gradients can be written in a clean, vectorized form and implemented efficiently.

5.2 Output Layer

A key simplification (same as Part I) is:

$$dZ^{[2]} = A^{[2]} - Y \qquad \in \mathbb{R}^{1\times m}.$$

Then:

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} (A^{[1]})^\top \qquad \in \mathbb{R}^{1\times n_1},$$

$$db^{[2]} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[2](i)} \qquad \in \mathbb{R}^{1\times 1}.$$

5.3 Hidden Layer

We propagate the gradient back to layer 1:

$$dZ^{[1]} = (W^{[2]})^\top dZ^{[2]} \odot A^{[1]} \odot (1 - A^{[1]}) \qquad \in \mathbb{R}^{n_1\times m},$$

where $\odot$ denotes elementwise multiplication.

Then:

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^\top \qquad \in \mathbb{R}^{n_1\times 2},$$

$$db^{[1]} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[1](i)} \qquad \in \mathbb{R}^{n_1\times 1}.$$

6 Training Procedure

At each iteration t, we perform:

- a forward pass to compute $A^{[2]}$,
- compute the loss $\mathcal{L}$,
- a backward pass to compute gradients,
- update parameters with learning rate α:

$$\begin{cases} W^{[1]} \leftarrow W^{[1]} - \alpha\, dW^{[1]}, \\ b^{[1]} \leftarrow b^{[1]} - \alpha\, db^{[1]}, \\ W^{[2]} \leftarrow W^{[2]} - \alpha\, dW^{[2]}, \\ b^{[2]} \leftarrow b^{[2]} - \alpha\, db^{[2]}. \end{cases}$$

Remark. We observe that, unlike in Part I, the training loss does not clearly plateau and continues to decrease throughout the run. A natural reaction would be to keep training for more iterations. However, the training accuracy reaches a near-stable value much earlier and improves only marginally afterward.

This highlights an important point: minimizing the *training* loss indefinitely is not necessarily desirable. Past a certain stage, additional iterations mainly help the network fit the training set more tightly (including noise), which can reduce performance on unseen data, a phenomenon known as *overfitting*. In practice, one would monitor the performance on a validation/test set and stop training once that performance stops improving (early stopping). We will address this generalization issue more explicitly in Part III.

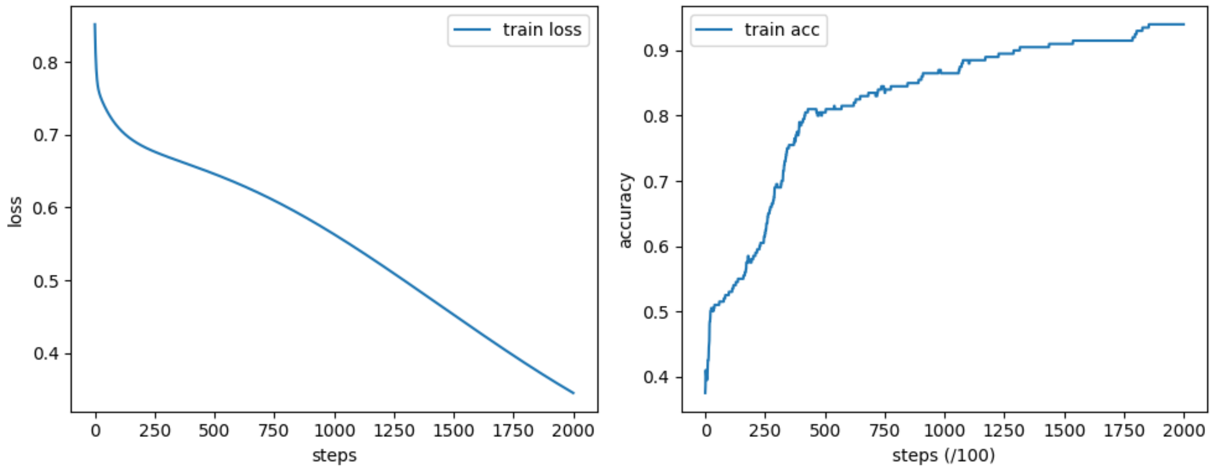

Figure 3: Training curves (loss and accuracy) over iterations.

7 Train/Test Split and Generalization

To evaluate the model, we generate a separate test set and compute the accuracy using the learned parameters.

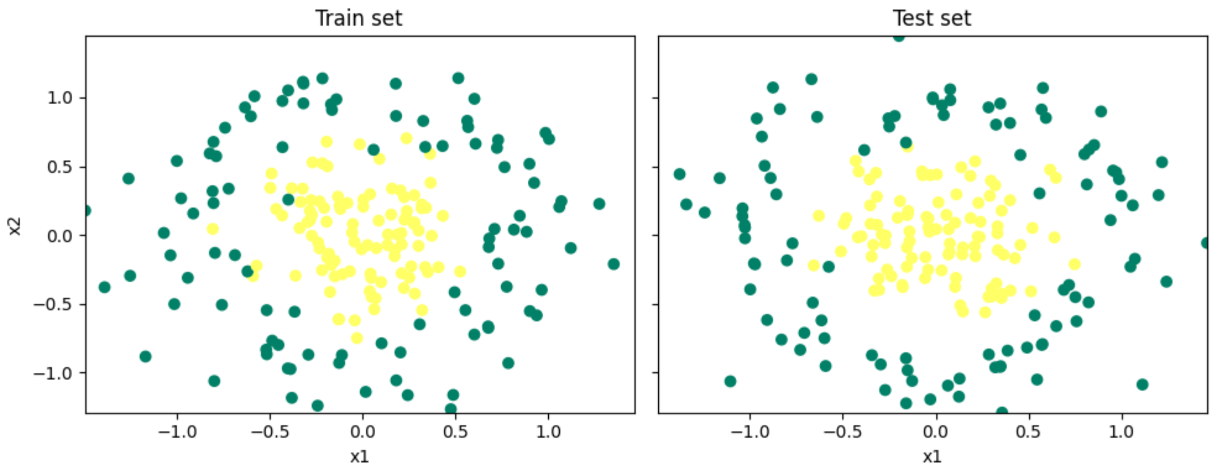

Figure 4: Test set generated independently from the training set.

We achieve an accuracy on the test set of up to 96%.

8 Decision Regions and Visualization

Unlike logistic regression, the MLP can create nonlinear decision boundaries. A practical way to visualize the learned classifier is:

- build a grid of points in the (x_1, x_2) plane,
- compute $A^{[2]}$ on the grid via forward propagation,
- display the resulting probabilities as a heatmap,

- plot the contour $A^{[2]} = 0.5$ as the decision boundary.

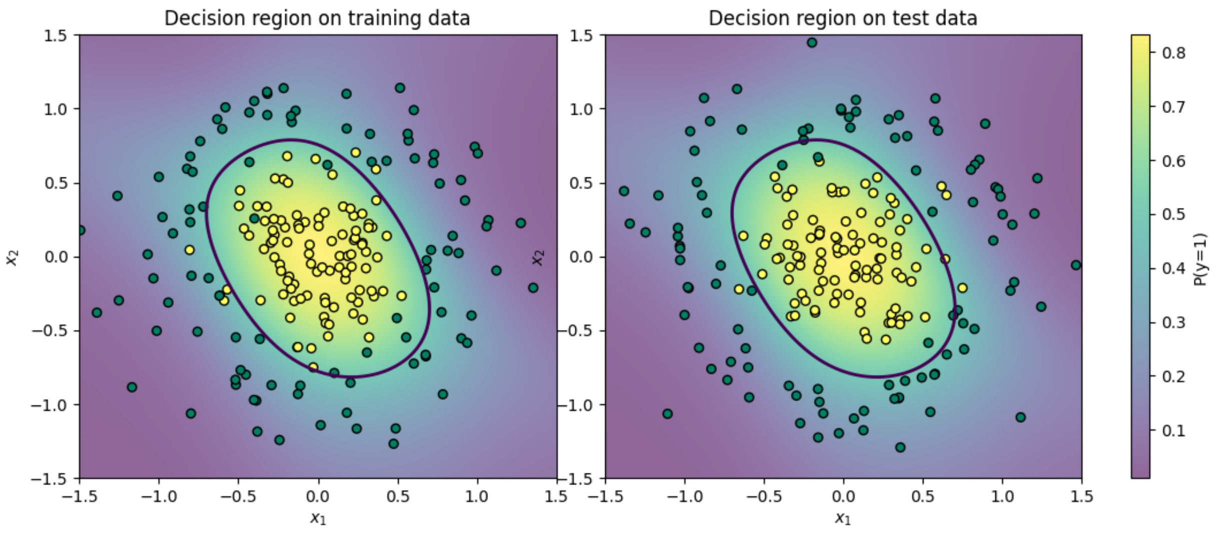

Figure 5: Decision regions learned by the MLP (probability heatmap and decision boundary).

9 Remark on Scaling

All formulas above remain valid if we increase:

- the number of input features (replace 2 by d),
- the number of neurons in the hidden layer (n_1),
- the number of layers (deeper networks).

What changes is mainly the shape of the matrices, but the core idea stays the same: *forward propagate to compute activations, backpropagate to compute gradients, update parameters.*

10 Conclusion and Link to the Implementation

This Part II shows how adding a hidden layer allows us to move beyond linear decision boundaries. The MLP remains conceptually simple, but becomes much more expressive thanks to the nonlinear activation.

The complete implementation (forward pass, backpropagation, training loop, and visualizations) is available here: Part II notebook/code.

In Part III, we will push these ideas further by studying multi-class classification by doing handwritten digit recognition. We will also see how to start choosing the correct hyperparameters for our model and some problems we may encounter during training.