

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.

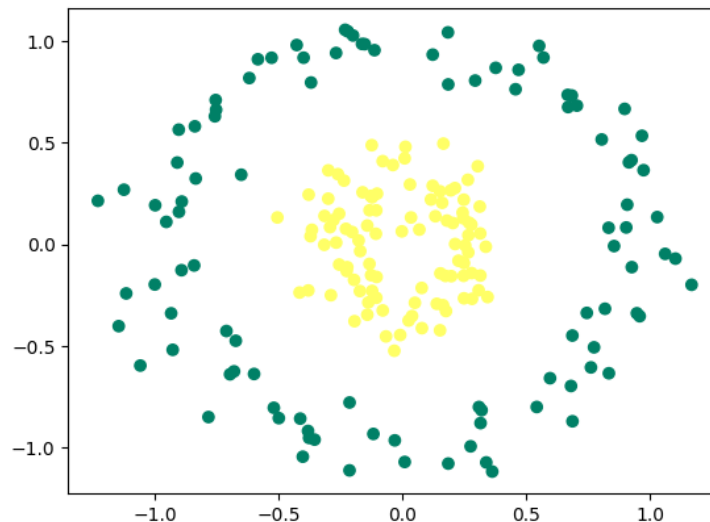


Figure 1: Example of a non-linearly separable dataset (concentric circles).

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:

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

A schematic view is shown in Figure 2.

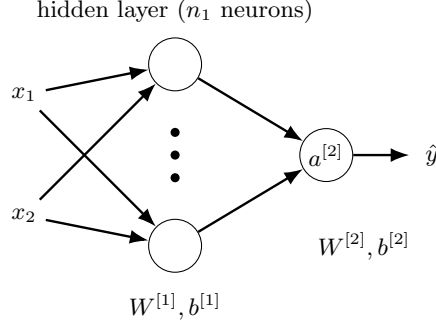


Figure 2: Two-layer MLP for binary classification (one hidden layer).

2.2 Forward Propagation

For a single example x , the forward pass is:

$$\begin{aligned} z^{[1]} &= W^{[1]}x + b^{[1]}, & a^{[1]} &= \sigma(z^{[1]}), \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]}, & a^{[2]} &= \sigma(z^{[2]}). \end{aligned}$$

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

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

With this convention, forward propagation becomes:

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]}, & A^{[1]} &= \sigma(Z^{[1]}), \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]}, & A^{[2]} &= \sigma(Z^{[2]}). \end{aligned}$$

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:

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

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

5 Backpropagation

5.1 The Idea

In Part I, computing gradients was relatively direct because the loss depended on a single linear 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 whole chain of intermediate computations:

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

Backpropagation is simply the systematic way to compute all partial derivatives $\frac{\partial \mathcal{L}}{\partial W^{[\ell]}}$ and $\frac{\partial \mathcal{L}}{\partial b^{[\ell]}}$ efficiently in such a chain. It is not a different learning rule: it is an efficient application of the *chain rule*.

The key idea is to start from the end (the output layer), compute how the loss changes when the output changes, and then propagate this information backward through the network. For each layer ℓ , we define

$$dZ^{[\ell]} := \frac{\partial \mathcal{L}}{\partial Z^{[\ell]}},$$

which measures how sensitive the loss is to the pre-activation $Z^{[\ell]}$. Once $dZ^{[\ell]}$ is known, the gradients of the parameters at that layer follow immediately (just as in Part I).

The name *backpropagation* comes from this mechanism: we *propagate* derivatives *backward* from the output layer to the earlier layers, using the chain rule at each step. This gives all gradients in one pass, with a computational cost comparable to the forward pass.

5.2 Why the Chain Rule Introduces $dZ^{[2]}$ and $dZ^{[1]}$

Gradient descent tells us that training consists in updating every parameter in the direction of the negative gradient:

$$W^{[\ell]} \leftarrow W^{[\ell]} - \alpha \frac{\partial \mathcal{L}}{\partial W^{[\ell]}}, \quad b^{[\ell]} \leftarrow b^{[\ell]} - \alpha \frac{\partial \mathcal{L}}{\partial b^{[\ell]}} \quad (\ell = 1, 2).$$

So the practical problem is clear: we must compute the partial derivatives of the loss with respect to the parameters of *each* layer.

Unlike Part I, the loss \mathcal{L} does not depend directly on the first-layer parameters $(W^{[1]}, b^{[1]})$. Instead, their influence passes through the intermediate variables created during the forward pass:

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

This dependency chain is precisely why the *chain rule* becomes unavoidable. For instance, differentiating with respect to a parameter of the output layer naturally produces 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 differentiating with respect to a parameter of the first layer makes the chain even longer, because changes in $W^{[1]}$ first modify $Z^{[1]}$, then $A^{[1]}$, and only *then* affect the output and the loss.

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

$$dZ^{[2]} := \frac{\partial \mathcal{L}}{\partial Z^{[2]}} \quad \text{and} \quad dZ^{[1]} := \frac{\partial \mathcal{L}}{\partial Z^{[1]}}.$$

This is exactly why the next subsections focus on deriving $dZ^{[2]}$ first (starting from the loss at the output), and then propagating it backward to obtain $dZ^{[1]}$. With these two results in hand, all parameter gradients can be computed efficiently and the gradient descent updates become straightforward to implement.

5.3 Output Layer

A key simplification (same as Part I) is:

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

Then:

$$\begin{aligned} dW^{[2]} &= \frac{1}{m} dZ^{[2]} (A^{[1]})^\top \quad \in \mathbb{R}^{1 \times n_1}, \\ db^{[2]} &= \frac{1}{m} \sum_{i=1}^m dZ^{[2](i)} \quad \in \mathbb{R}^{1 \times 1}. \end{aligned}$$

5.4 Hidden Layer

We propagate the gradient back to layer 1:

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

where \odot denotes elementwise multiplication.

Then:

$$\begin{aligned} dW^{[1]} &= \frac{1}{m} dZ^{[1]} X^\top \quad \in \mathbb{R}^{n_1 \times 2}, \\ db^{[1]} &= \frac{1}{m} \sum_{i=1}^m dZ^{[1](i)} \quad \in \mathbb{R}^{n_1 \times 1}. \end{aligned}$$

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 :

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

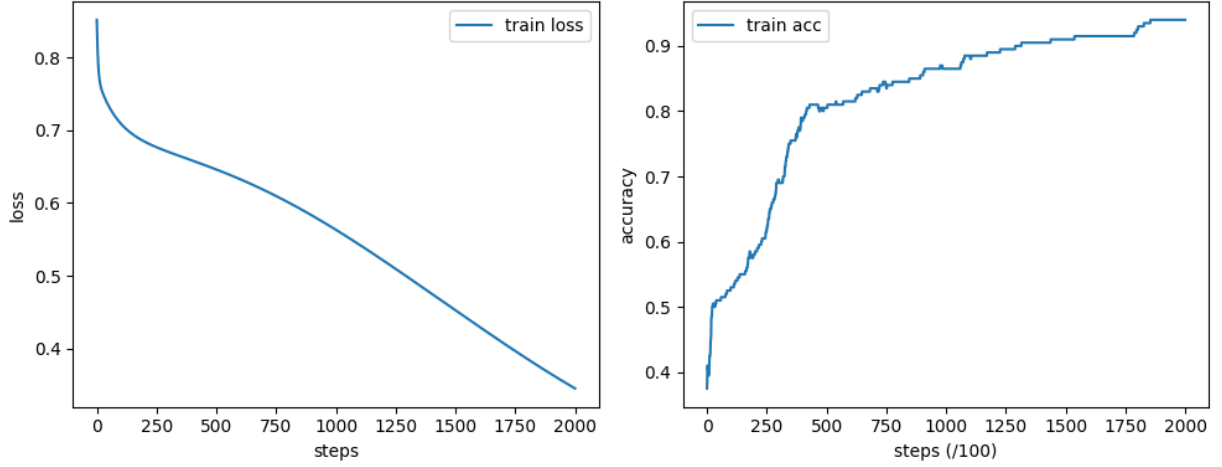


Figure 3: Training curves (loss and accuracy) over iterations.

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.

7 Train/Test Split and Generalization

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

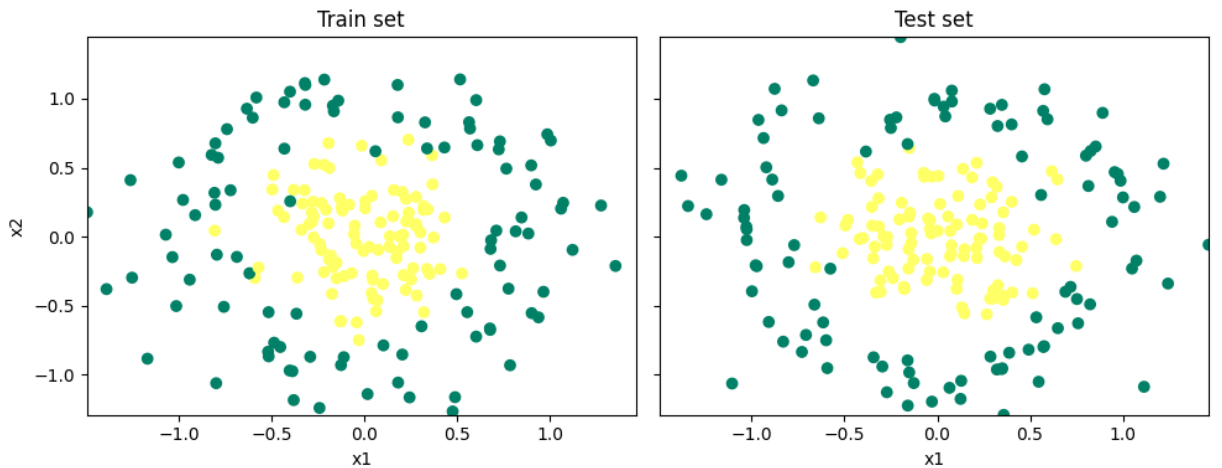


Figure 4: Test set generated independently from the training set.

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.

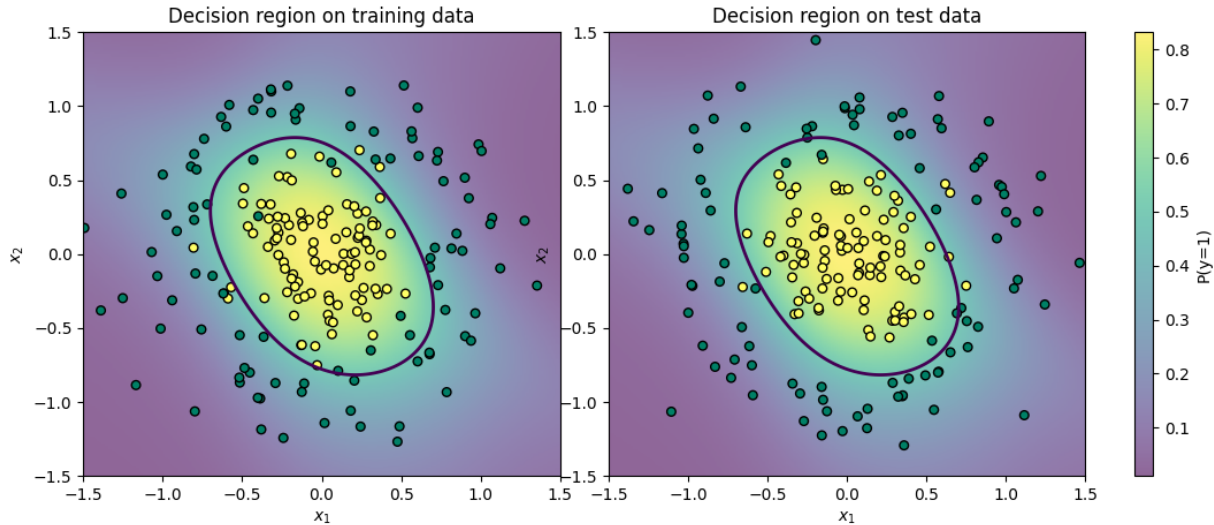


Figure 5: Decision regions learned by the MLP (probability heatmap and decision boundary).

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.