Deep Learning Course

Lesson 2 — Feedforward Neural Networks (ANNs)

Andrea Giardina
contact@andreagiardina.com
https://www.linkedin.com/in/agiardina

October 17, 2025

Lesson Goals

- Build a full **NumPy-only** MNIST pipeline for **forward inference** (no training loops).
- Understand images as tensors: shapes, dtypes, normalization, standardization.
- Implement a **784→32→10 MLP** forward pass and evaluation.
- Use a **pre-trained mini-network** to test inference.

Out of Scope Today

- **No** backpropagation, **no** gradient descent, **no** optimizers.
- **No** CNNs, data augmentation, or regular training loops.
- Gradients and parameter updates are covered in the next lesson.

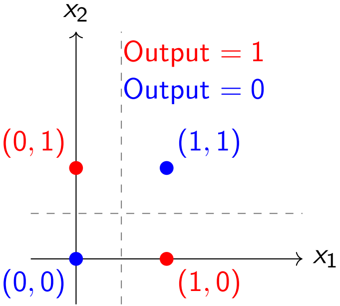
Linear Separability and the Perceptron

- A single perceptron learns **linear** decision boundaries.
- Works when classes are linearly separable.
- Fails when no hyperplane can separate the classes.

The XOR Problem

- XOR truth table: outputs 1 when inputs differ, 0 otherwise.
- The positive and negative points are **not** linearly separable.
- No single line/hyperplane separates XOR classes in $\mathbb{R}^2$.

XOR Problem Visualization



XOR (exclusive OR): The output is 1 if *exactly one* of x_1 or x_2 is 1.

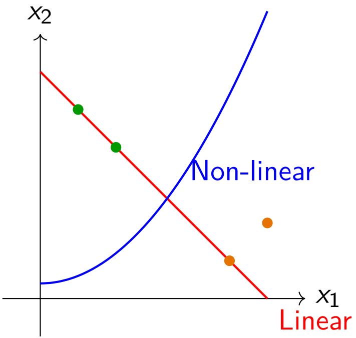
$$y = x_1 \oplus x_2$$

$\rightarrow$ *Not linearly separable with a single perceptron.*

Understanding Non-Linearity

- A **linear model** can only draw straight lines (or hyperplanes) to separate data.
- Some problems (like XOR) **cannot be separated** by a straight line.
- Adding **non-linear activation functions** (e.g. ReLU, sigmoid, tanh) allows the network to:
  - combine multiple linear regions,
  - create curved decision boundaries,
  - and learn complex relationships.

Without non-linearity, a neural network behaves like a single linear model.



Why Linear + Linear = Linear

Composing linear functions does not increase expressiveness.

$$\text{Let } f(\mathbf{x}) = W_2(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

then

$$f(\mathbf{x}) = (W_2 W_1)\mathbf{x} + (W_2\mathbf{b}_1 + \mathbf{b}_2)$$

Observation:

$$f(\mathbf{x}) = A\mathbf{x} + \mathbf{c} \text{ where } A = W_2 W_1 \text{ and } \mathbf{c} = W_2\mathbf{b}_1 + \mathbf{b}_2.$$

The result is still linear

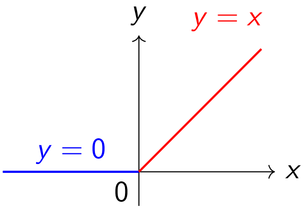
Therefore, stacking linear layers without a non-linear activation is equivalent to a single linear transformation.

Non-Linear Activation: ReLU

Rectified Linear Unit (ReLU):

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

- Introduces **non-linearity** while keeping computation simple.
- Outputs 0 for negative inputs, and passes positive inputs unchanged.
- Allows neural networks to learn **non-linear decision boundaries**.

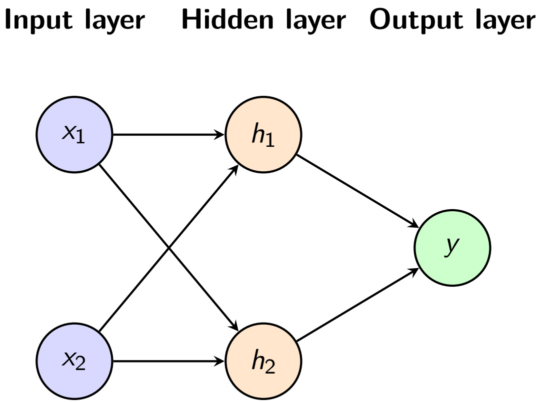


Without ReLU (or any non-linearity), multiple layers would collapse into a single linear transformation.

Two-Layer Network Solves XOR

- Hidden layer splits input space into regions.
- Step1: $h = f(W_1 x + b_1)$
- Step2: $y = W_2 h + b_2$.
- With suitable W_1, b_1, W_2, b_2 and f (e.g., ReLU/sigmoid), XOR is representable.

Neural Network with 2–2–1 Architecture



A simple feedforward neural network with two inputs, two hidden units, and one output.

Back to school: Matrix Multiplication 1/2

Definition:

$$C = A \times B$$

If

$$A \in \mathbb{R}^{m \times n}, \quad B \in \mathbb{R}^{n \times p}$$

then

$$C \in \mathbb{R}^{m \times p}, \quad c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Example: $A_{2\times3} \times B_{3\times2} = C_{2\times2}$

- Each element c_{ij} is the **dot product** of row i of A and column j of B.
- Order matters: $A \times B \neq B \times A$ (in general).
- In neural networks: $\mathbf{y} = W\mathbf{x}$ combines inputs linearly.

Back to school: Matrix Multiplication 2/2

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}}_{A_{2\times 3}} \times \underbrace{\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}}_{B_{3\times 2}} = \underbrace{\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}}_{C_{2\times 2}}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

Row 1 of A · Column 1 of B $\rightarrow$ c_{11}

Quiz time: XOR Network, what are the shapes?

We use a simple neural network to solve the XOR problem:

$$\mathbf{x} \in \mathbb{R}^2 \quad \xrightarrow{W_1, b_1} \quad \text{Hidden layer (2 units)} \quad \xrightarrow{W_2, b_2} \quad \text{Output (1 unit)}$$

Question: Given this 2–2–1 architecture, what are the shapes of each matrix and bias?

Input: $\mathbf{x}$ (?) $\quad W_1$ (?), $\quad b_1$ (?), $\quad W_2$ (?), $\quad b_2$ (?)

XOR Network: Parameter Shapes (Answer)

For a 2–2–1 network:

$$\mathbf{x} \in \mathbb{R}^2 \quad \xrightarrow{W_1, b_1} \quad \text{Hidden layer } \mathbf{h} \in \mathbb{R}^2 \quad \xrightarrow{W_2, b_2} \quad \text{Output } y \in \mathbb{R}^1$$

$$W_1 \in \mathbb{R}^{2\times 2} \qquad \text{(2 hidden units, 2 inputs)}$$

$$b_1 \in \mathbb{R}^2 \qquad \text{(1 bias per hidden unit)}$$

$$W_2 \in \mathbb{R}^{2\times 1} \qquad \text{(2 hidden units, 1 output)}$$

$$b_2 \in \mathbb{R}^1 \qquad \text{(1 bias for the output unit)}$$

Quiz time 1/3

Question

x is a single sample, and W has shape equals to (output-size,input-size). Is x in the formula, $y = Wx$ a column vector or a row vector?

Quiz time 1/3

Question

x is a single sample, and W has shape equals to (output-size,input-size). Is x in the formula, $y = Wx$ a column vector or a row vector?

Answer

A column vector: it must have shape equals to (input-size, 1), otherwise the matrix multiplication wouldn't work. y has shape (output-size,1)

Quiz time 2/3

Question

Is x in the formula, $y = xW^T$, where W^T is the transpose of the matrix W, a column vector or a row vector?

Quiz time 2/3

Question

Is x in the formula, $y = xW^T$, where W^T is the transpose of the matrix W, a column vector or a row vector?

Answer

A row vector: it must have shape equals to (1,input-size), because W^T has shape (input-size,output-size). y has shape (1, output-size)

Quiz time 3/3

Question

If X is a batch of N samples with shape (N, input-size), what is the right formula to use?
$y = WX$ or $y = XW^T$?

Quiz time 3/3

Question

If X is a batch of N samples with shape (N, input-size), what is the right formula to use? $y = WX$ or $y = XW^T$?

Answer

$y = XW^T$ because X has shape (N, input-size) and W^T has shape (input-size,output-size). y has shape (N,output-size), so, it has one "row" for each input in the batch.

Lab Time: the xor network

Find the weights of the W_2 matrix.

```
import numpy as np

def relu(z): return np.maximum(z, 0.0)

X = np.array([[0, 0],
              [1, 0],
              [0, 1],
              [1, 1]])

y = np.array([0, 1, 1, 0]).reshape(-1,4)
W1 = np.ones((2, 2))
W2 = ???
b1 = np.array([0,-1])
b2 = 0
H = relu(X @ W1.T + b1)
y_hat = H @ W2
assert(y_hat.all() == y.all())
```

Lab Time: the xor network

Find the weights of the W_2 matrix.

```
import numpy as np

def relu(z): return np.maximum(z, 0.0)

X = np.array([[0, 0],
              [1, 0],
              [0, 1],
              [1, 1]])

y = np.array([0, 1, 1, 0]).reshape(-1,4)
W1 = np.ones((2, 2))
W2 = np.array([1,-2])
b1 = np.array([0,-1])
b2 = 0
H = relu(X @ W1.T + b1)
y_hat = H @ W2


print(H)
print(X @ W1.T)
```

Lab Time: the MLP class

Create a MLP class with only the *constructor* and the *forward* method. The constructor takes $\mathbf{W_1}, \mathbf{b_1}, \mathbf{W_2}, \mathbf{b_2}$ and the **activation function** as input. The forward method takes $\mathbf{X}$ as input and return the $\widehat{\mathbf{y}}$ vector. Test the class with inputs and weights from the xor network example.

My trivial implementation

```python
#....
class MLP:
    def __init__(self,W1,b1,W2,b2,activaction):
        self.W1 = W1
        self.W2 = W2
        self.b1 = b1
        self.b2 = b2
        self.activation = activaction

    def forward(self,X):
        H = self.activation(X @ self.W1.T + self.b1)
        y_hat = H @ self.W2.T + self.b2
        return y_hat

mlp = MLP(W1,b1,W2,0,relu)
print(mlp.forward(X))
```

A more interesting problem: the MNIST Dataset

MNIST (Modified National Institute of Standards and Technology) is one of the most iconic datasets in machine learning and computer vision. It contains **70,000 grayscale images** of handwritten digits (0–9), each of size **28×28 pixels**, divided into **60,000 training** and **10,000 test** samples.

Historical background:

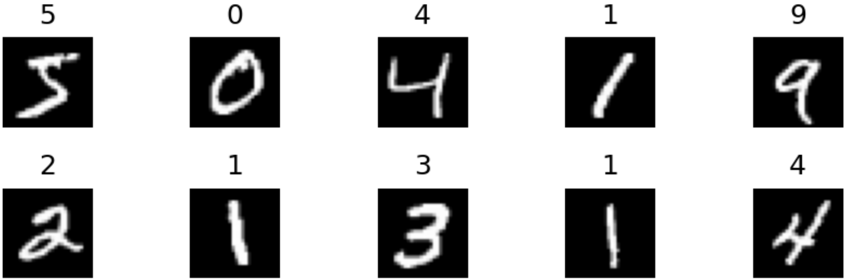
- Created in **1998** by **Yann LeCun**, **Corinna Cortes**, and **Christopher J.C. Burges**.
- Derived from the original **NIST handwritten digit database** from the late 1980s.
- The original dataset used digits written by two groups: U.S. Census Bureau employees and American high school students.
- MNIST combined and normalized samples from both sources, hence the "**Modified**" in its name.

Why it matters:

- Served as the **benchmark dataset** for early image recognition research.
- Used to test **Convolutional Neural Networks (CNNs)**, including LeCun's famous **LeNet-5**.
- Though now considered a "**solved problem**", MNIST remains a standard starting point for learning and testing new models.

Examples from the MNIST Dataset

Each image in MNIST is a **28×28 grayscale** handwritten digit. The examples below illustrate the natural variation in handwriting across individuals.



- Digits range from **0** to **9**.
- Pixel values range from **0 (white)** to **255 (black)**.

Recreating MNIST .npy Files 1/2

The original MNIST dataset is distributed as four compressed binary files in **IDX** format. We can convert them into easy-to-use **NumPy arrays** by following three main steps:

1. Download the Original Files

- Hosted on: `https://ossci-datasets.s3.amazonaws.com/mnist/`
- Files:
  - `train-images-idx3-ubyte.gz` — 60,000 images
  - `train-labels-idx1-ubyte.gz` — 60,000 labels
  - `t10k-images-idx3-ubyte.gz` — 10,000 images
  - `t10k-labels-idx1-ubyte.gz` — 10,000 labels

2. Decompress and Parse

- Use `gzip` to read binary files.
- Skip headers: `offset=16` (images), `offset=8` (labels).
- Reshape images to `(-1, 28×28)`.

Recreating MNIST .npy Files 2/2

3. Save as .npy

- Save the NumPy arrays:
  - `np.save("train_images.npy", np_train_images)`
  - `np.save("train_labels.npy", np_train_labels)`
- Load them later with `np.load(...)` for instant access.

Result: Four reusable files — `train/test_images.npy` and `train/test_labels.npy`.

MNIST dataset

- In a **color image**, each pixel is composed of **three channels**: Red(R),Green(G) and Blue(B).Each channel stores an intensity value between **0 and 255**:

$$\text{Pixel} = [R,\ G,\ B] = [255,\ 128,\ 0]$$

- When all three components have the same intensity, i.e. R=G=B the resulting color is a gray level, ranging from black (0,0,0) to white (255,255,255).

MNIST dataset

- In a **color image**, each pixel is composed of **three channels**: Red(R),Green(G) and Blue(B).Each channel stores an intensity value between **0 and 255**:

$$\text{Pixel} = [R,\ G,\ B] = [255,\ 128,\ 0]$$

- When all three components have the same intensity, i.e. R=G=B the resulting color is a gray level, ranging from black (0,0,0) to white (255,255,255).
- A **grayscale image** like MNIST uses only one channel - a single value per pixel.
- In the MNIST image dataset each pixel is represented by one byte indicating the gray-level intensity, ranging from 0 (black) to 255 (white).
- To read a full image you have to read 784 bytes (784=28x28)
- Each byte in the MNIST label dataset is an integer label from 0 to 9, representing the digit shown in the corresponding image.

Using a Trained Neural Network for Inference

Idea

Once the network has been trained, its weights $\mathbf{W}$ and biases $\mathbf{b}$ are fixed. Inference means applying the same transformations learned during training to **new unseen data**.

Using a Trained Neural Network for Inference

Idea

Once the network has been trained, its weights $\mathbf{W}$ and biases $\mathbf{b}$ are fixed. Inference means applying the same transformations learned during training to **new unseen data**.

Goal

Predict the output $\hat{y}$ given a new input $\mathbf{x}_{new}$ using the learned model:

$$\hat{y} = f(\mathbf{x}_{new}; \mathbf{W}, \mathbf{b})$$

Using a Trained Neural Network for Inference

Idea

Once the network has been trained, its weights $\mathbf{W}$ and biases $\mathbf{b}$ are fixed. Inference means applying the same transformations learned during training to **new unseen data**.

Goal

Predict the output $\hat{y}$ given a new input $\mathbf{x}_{new}$ using the learned model:

$$\hat{y} = f(\mathbf{x}_{new}; \mathbf{W}, \mathbf{b})$$

Important

No learning occurs during inference — the model only performs a **forward pass**.

Steps to Perform Inference

1. **Load the trained model:** Retrieve the weights $\mathbf{W}$ and biases $\mathbf{b}$ saved after training.

Steps to Perform Inference

1. **Load the trained model:** Retrieve the weights $\mathbf{W}$ and biases $\mathbf{b}$ saved after training.
2. **Prepare the input:** Apply the **same preprocessing** as in training (normalization, scaling, encoding).

Steps to Perform Inference

1. **Load the trained model:** Retrieve the weights $\mathbf{W}$ and biases $\mathbf{b}$ saved after training.
2. **Prepare the input:** Apply the **same preprocessing** as in training (normalization, scaling, encoding).
3. **Forward pass:** Compute activations layer by layer without updating weights.

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \quad \mathbf{a}^{(l)} = g(\mathbf{z}^{(l)})$$

Steps to Perform Inference

1. **Load the trained model:** Retrieve the weights $\mathbf{W}$ and biases $\mathbf{b}$ saved after training.
2. **Prepare the input:** Apply the **same preprocessing** as in training (normalization, scaling, encoding).
3. **Forward pass:** Compute activations layer by layer without updating weights.

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \quad \mathbf{a}^{(l)} = g(\mathbf{z}^{(l)})$$

4. **Output interpretation:**
   - Regression $\rightarrow \hat{y}$ is a numeric value
   - Classification $\rightarrow$ choose $\arg\max(\hat{y})$

Inference Pipeline Overview

New data $\mathbf{x}_{new}$ → Preprocessing → Trained network → Prediction $\hat{y}$

The network applies the same learned transformations to the new input without modifying its parameters.

Pretrained Network for MNIST Digits

Task

We trained a simple feedforward neural network to recognize handwritten digits (MNIST dataset).

Pretrained Network for MNIST Digits

Task

We trained a simple feedforward neural network to recognize handwritten digits (MNIST dataset).

- **Input layer:** $28 \times 28 = 784$ neurons (pixel intensities)
- **Hidden layer:** 32 neurons with nonlinear activation
- **Output layer:** 10 neurons (digits 0–9)

Pretrained Network for MNIST Digits

Task

We trained a simple feedforward neural network to recognize handwritten digits (MNIST dataset).

- **Input layer:** $28 \times 28 = 784$ neurons (pixel intensities)
- **Hidden layer:** 32 neurons with nonlinear activation
- **Output layer:** 10 neurons (digits 0–9)

Model summary

$$\mathbf{x} \in \mathbb{R}^{784} \xrightarrow{\mathbf{W}_1,\, \mathbf{b}_1} \mathbf{h} \in \mathbb{R}^{32} \xrightarrow{\mathbf{W}_2,\, \mathbf{b}_2} \hat{\mathbf{y}} \in \mathbb{R}^{10}$$

Pretrained Network for MNIST Digits

Task

We trained a simple feedforward neural network to recognize handwritten digits (MNIST dataset).

- **Input layer:** $28 \times 28 = 784$ neurons (pixel intensities)
- **Hidden layer:** 32 neurons with nonlinear activation
- **Output layer:** 10 neurons (digits 0–9)

Model summary

$$\mathbf{x} \in \mathbb{R}^{784} \quad \xrightarrow{\mathbf{W}_1,\, \mathbf{b}_1} \quad \mathbf{h} \in \mathbb{R}^{32} \quad \xrightarrow{\mathbf{W}_2,\, \mathbf{b}_2} \quad \hat{\mathbf{y}} \in \mathbb{R}^{10}$$

Training complete

Weights $\mathbf{W}_1, \mathbf{W}_2$ and biases $\mathbf{b}_1, \mathbf{b}_2$ are now **frozen** and ready for inference.

What the Network Has Learned

Hidden layer representation

Each hidden neuron combines pixel patterns to detect simple features, like edges, curves, or parts of digits.

What the Network Has Learned

Hidden layer representation

Each hidden neuron combines pixel patterns to detect simple features, like edges, curves, or parts of digits.

Output layer interpretation

The 10 output neurons represent the probability of each class:

$$\hat{y}_i = P(\text{digit} = i \mid \mathbf{x})$$

and the predicted digit is

$$\text{Predicted class} = \arg\max_i(\hat{y}_i)$$

What the Network Has Learned

Hidden layer representation

Each hidden neuron combines pixel patterns to detect simple features, like edges, curves, or parts of digits.

Output layer interpretation

The 10 output neurons represent the probability of each class:

$$\hat{y}_i = P(\text{digit} = i \mid \mathbf{x})$$

and the predicted digit is

$$\text{Predicted class} = \arg\max_i(\hat{y}_i)$$

Example

For an image of “3”:

$$\hat{\mathbf{y}} = [0.01, 0.02, 0.05, \mathbf{0.90}, 0.01, 0.00, \dots]$$

The network predicts **digit 3**.

Normalization Applied Before Training

Goal

Ensure that all input features have comparable ranges, so that the neural network trains stably and converges faster.

Normalization Applied Before Training

Goal

Ensure that all input features have comparable ranges, so that the neural network trains stably and converges faster.

Procedure

1. The input values has been scaled, so the range is not between $0 - 255$ but between $0 - 1$.
2. Compute the **mean** and **standard deviation** of each feature using only the training data.
3. If a standard deviation is extremely small ($\text{std} < 10^{-6}$), replace it with 10^{-6} to avoid division by zero.
4. Normalize all datasets (training, validation, test) using: $X' = \frac{X - \text{mean}}{\text{std}}$

Normalization Applied Before Training

Goal

Ensure that all input features have comparable ranges, so that the neural network trains stably and converges faster.

Procedure

1. The input values has been scaled, so the range is not between $0 - 255$ but between $0 - 1$.
2. Compute the **mean** and **standard deviation** of each feature using only the training data.
3. If a standard deviation is extremely small ($\text{std} < 10^{-6}$), replace it with 10^{-6} to avoid division by zero.
4. Normalize all datasets (training, validation, test) using: $X' = \frac{X - \text{mean}}{\text{std}}$

Key point

The same mean and std values computed on the **training set** must be reused during inference - never recomputed on new data.

Lab Time

1. Download **W1.npy, b1,npy, W2.npy, b2.npy, mean.npy** and **std.npy** from `https://github.com/agiardina/rawai` (lesson2 folder) and load them with `np.load`
2. Let X the **test_images.npy** matrix
3. Scale X so that all values are between 0 and 1
4. Normalize X with the formula $\frac{X - \text{mean}}{\text{std}}$
5. Use the previous created MLP class and the loaded weights to predict the labels of X ($\hat{Y}$)
6. Let Y the **test_labels.npy** vector and calculate the model accurancy, where

$$\text{Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}(\hat{y}_i = y_i)$$

My trivial implementation

```
import numpy as np

def relu(x): return np.maximum(0, x)

class MLP:
    def __init__(self, W1, b1, W2, b2, activation):
        self.W1 = W1.astype(np.float32)
        self.b1 = b1.astype(np.float32).reshape(-1)
        self.W2 = W2.astype(np.float32)
        self.b2 = b2.astype(np.float32).reshape(-1)
        self.activation = activation

    def forward(self, X):
        H = self.activation(X @ self.W1.T + self.b1)
        y_hat = H @ self.W2.T + self.b2
        return y_hat

W1 = np.load("W1.npy")
b1 = np.load("b1.npy")
```

My trivial implementation

```
W2 = np.load("W2.npy")
b2 = np.load("b2.npy")
mean = np.load("mean.npy").astype(np.float32)
std  = np.load("std.npy").astype(np.float32)


X_test = np.load("test_images.npy").astype(np.float32)
y_test = np.load("test_labels.npy")

X_test /= 255.0
X_test = (X_test - mean) / std


model = MLP(W1, b1, W2, b2, activation=relu)
logits = model.forward(X_test)

y_pred = np.argmax(logits, axis=1)
num_correct = int((y_pred == y_test).sum())
total = y_test.shape[0]
accuracy = num_correct / total

print(f"Correct predictions: {num_correct}/{total}  (accuracy = {accuracy:.4%})")
```

Next Lesson

- From loss to gradients via chain rule.
- Parameter updates with gradient descent.

Thanks!