*Project 1 Briefing:*

Core Concepts and Foundations

*Author: Iván Blanco (Noax Trading). Date 23/04/2025*

**Project objective:**  
To introduce students to the foundational concepts of deep learning using PyTorch, starting from the basics of tensors and progressing toward simple neural networks. The project uses a real-world dataset throughout to build practical understanding and serves as a scaffold for more complex architectures in later modules.

**Development Constraints:**

* You must not use any external data sources or external APIs in any part of the notebooks.
* All examples, exercises, and model training must be based exclusively on the provided CSV file: Concrete\_Data.csv.

<https://archive.ics.uci.edu/dataset/165/concrete+compressive+strength>

* The use of internet connections, dynamic data fetching, or third-party API based datasets is strictly prohibited.
* Book (only for reference and NOTATION): <https://udlbook.github.io/udlbook/>

# Project Structure

This project consists of four progressively structured Jupyter notebooks:

|  |  |  |
| --- | --- | --- |
| Notebook | Title | Core Topic |
| 1 | Deep Learning Context and PyTorch Basics | Conceptual intro and PyTorch setup |
| 2 | Linear Regression and Gradient Descent | First model and training by hand |
| 3 | Non-Linearity and Activation Functions | Activations and limitations of linear models |
| 4 | From a Neuron to a Network (MLP) | Building a multi-layer neural network |

## Notebook 1 – Deep Learning Context and PyTorch Basics

**Learning Objectives:**

* Understand what deep learning is, its historical development, and modern applications.
* Learn how to create and manipulate tensors in PyTorch.
* Load and explore a real-world dataset (**Concrete Compressive Strength**).
* Understand how real-world data is represented as tensors (1D, 2D, 3D).
* Visualize data distributions and establish an intuitive grasp of input-output structures.

**Developer Instructions:**

Introduction

* Introduce key applications of deep learning using brief real-world examples (only Markdown no code).

Background Mathematics:

* Include a section of Background Mathematics (use this section notebook as example: <https://github.com/udlbook/udlbook/blob/main/Notebooks/Chap01/1_1_BackgroundMathematics.ipynb>)

Tensors/PyTorch + DataSet:

* Present different types of tensors with well-commented examples (scalar, vector, matrix, 3D tensor).
* Include a dedicated section demonstrating basic tensor operations in PyTorch, such as addition, element-wise multiplication, dot product, matrix multiplication, and broadcasting. Use small, interpretable tensors (e.g., vectors of size 2 or 3, 2×2 matrices) to keep examples intuitive. Accompany each operation with a short explanation and printed output.
* Use Markdown, when need it, with LaTeX to explain operations symbolically.
* Also, load the Concrete\_Data.csv dataset and show how it maps to tensors.
* Include at least one basic visualization of both inputs and target variables.
* Conclude the notebook with a short conceptual reflection on what features might be most relevant for predicting compressive strength.

## Notebook 2 – Linear Regression and Gradient Descent

**Learning Objectives:**

* Understand that a neural network with no hidden layers and no activation is a linear model.
* Implement linear regression from scratch using PyTorch.
* Define and compute the Mean Squared Error (MSE) loss also present other loss functions useful on regression problems.
* Manually perform gradient descent to train the model.
* Connect this to the need for deeper networks with activation functions.

**Developer Instructions:**

* Use one or two features from the dataset for simplicity.
* Introduce the concept of training and test sets, explaining their purpose in model evaluation and generalization.
* Split the dataset into training and test sets (e.g., 80% train, 20% test) using PyTorch or NumPy indexing.
* Introduce and explain the concept of **gradient descent**, including:
  + The purpose of gradient-based optimization
  + The update rules for weights and bias
  + The role of the learning rate
* Manually initialize weights and bias as PyTorch tensors with requires\_grad=True.
* Define a predict(X) function that computes y=X⋅W+b
* Implement the Mean Squared Error (MSE) as a separate function.
* Train the model on the training set using a gradient descent loop: forward pass, loss computation .backward( ), and parameter update with torch.no\_grad( ).
* After training, compute and compare the MSE on both the training and test sets to illustrate model generalization and the potential for underfitting or overfitting.
* Plot the training loss over time using matplotlib.
* End with a short theory section discussing the limitations of linear models and introducing the motivation for using activation functions and hidden layers in the next notebook.

## Notebook 3 – Non-Linearity and Activation Functions

**Learning Objectives:**

* Understand the limitation of linear models: they cannot capture non-linear patterns.
* Learn how activation functions introduce non-linearity into neural networks.
* Implement simple neural networks with and without activation functions.
* Visualize and compare model performance on linearly vs. non-linearly separable data.
* Develop intuition for the role of functions like ReLU, Sigmoid, and Tanh.

**Developer Instructions:**

* Begin with a recap of linear models and what they can and cannot represent (e.g., only linear relationships).
* Introduce a small synthetic or simplified real example (1D or 2D) where a non-linear relationship is visually evident and cannot be captured by a linear model (e.g., polynomial curve or XOR).
* Explain why stacking linear layers without activation functions does not solve this issue (remains linear).
* Introduce the concept of activation functions and explain their role in adding non-linearity:
  + Provide theoretical definitions and LaTeX-rendered formulas for ReLU, Sigmoid, and Tanh.
  + Include simple plots showing the shape and behavior of each activation function.
  + Use this notebook as a reference: <https://github.com/udlbook/udlbook/blob/main/Notebooks/Chap03/3_4_Activation_Functions.ipynb>
* Define and implement a simple neural network with a single hidden layer:
  + Use manual layer composition: Linear → Activation → Linear (not Sequential or Module)
  + The model architecture should be shallow, typically:
    - y^=W2⋅f(W1⋅X+b1)+b2\hat{y} = W\_2 \cdot f(W\_1 \cdot X + b\_1) + b\_2y^​=W2​⋅f(W1​⋅X+b1​)+b2​
  + This is not a general-purpose MLP yet, but a minimal working example to demonstrate the effect of non-linearity.
* Compare two models on the same dataset:
  + One without activation (Linear → Linear)
  + One with activation (Linear → ReLU → Linear)
* Train both models and plot:
  + Training loss over epochs
  + Prediction curves or scatter plots (depending on the problem)
* Emphasize the difference in performance and explain why non-linearity helps.
* Conclude with a theoretical summary explaining:
  + The limits of purely linear stacks
  + Why deep networks require activation functions
  + How this prepares us to build Multi-Layer Perceptrons (MLPs) in Notebook 4

## Notebook 4 – From a Neuron to a Network (MLP)

**Learning Objectives:**

* Understand how a neural network is built from stacked linear transformations and activation functions.
* Learn how to implement a Multi-Layer Perceptron (MLP) using nn.Module or nn.Sequential.
* Train a model using torch.optim and nn.MSELoss.
* Compare the performance of a deep model with previous linear-only models.
* Reflect on overfitting, underfitting, and the role of hidden layers.

**Developer Instructions:**

* Start with a conceptual intro explaining why stacking linear layers is not enough.
* Present and define the concept of hyperparameters, including:
  + What they are (values not learned during training)
  + Common examples: learning rate, number of layers, number of neurons, activation functions, number of epochs, batch size
  + Why tuning hyperparameters affects model performance and generalization
* Define an MLP model using PyTorch, either with nn.Sequential and a custom nn.Module (Explain both briefly).
* Use multiple input features from the dataset (ideally all 8) as model input.
* Include at least one hidden layer with an activation function (e.g., ReLU).
* Train the model using torch.optim (e.g., SGD or Adam) and nn.MSELoss.
* Plot the training loss over epochs using matplotlib, and comment on convergence behavior.
* Provide a short Markdown section explaining model capacity, overfitting, and underfitting.
* Optionally, encourage experimentation by allowing students to modify hyperparameters such as number of layers, hidden units, or activation functions to see how it affects performance.

# Dataset Requirements

Use the Concrete Compressive Strength dataset from the UCI Machine Learning Repository.  
Filename: Concrete\_Data.csv

<https://archive.ics.uci.edu/dataset/165/concrete+compressive+strength>

The dataset should be used consistently across all four notebooks.

# Technical and Style Guidelines

- Alternate consistently between Markdown (for theory) and code (for implementation).  
- Markdown cells must include LaTeX-style equations where needed.  
- Each notebook should follow a clear pedagogical progression: introduction, concept, code, reflection.  
- Code must be clean, commented, and executable top-to-bottom.  
- Use visualizations where helpful (e.g., histograms, line plots for loss, etc.).

# Deliverables

- Four complete .ipynb notebooks (or Google Colab), one for each part of the project.  
- Clear and structured use of Markdown and code.  
- LaTeX used in Markdown to explain mathematical ideas when needed.  
- Visualizations included where helpful for understanding.  
- Notebooks should be clean, readable, and runnable from top to bottom without error.

**What’s Coming in the Next Project**

**Project 2: Neural Networks**

* Introduce the **Perceptron** as the most basic unit in a neural network.
* Explore **Feedforward Neural Networks (FNNs)** and understand how data flows through them.
* Introduce the concept of **Backpropagation** and how weights are updated during training.

**Project 3: Training Neural Networks**

* Cover practical **optimization techniques**, including basic gradient descent, stochastic gradient descent, and mini-batch variants.
* Compare different **loss functions**, such as Mean Squared Error and Cross-Entropy.
* Introduce **regularization techniques** to combat overfitting, including **Dropout** and **L2 regularization**.

**Project 4: Advanced Deep Neural Networks**

* Learn how to build and train **deep networks** with more layers and more complex architecture.
* Introduce classic deep architectures such as **LeNet**, **AlexNet**, and **VGG**.
* Address challenges like the **vanishing and exploding gradient problems**, and introduce basic mitigation strategies (e.g., better initialization, batch normalization).

**Project 5: Convolutional Neural Networks (CNNs) Overview**

* Present the fundamental concepts of **Convolutional Neural Networks (CNNs)**: convolutional layers, pooling layers, and fully connected layers.
* Discuss **real-world applications** of CNNs, particularly in image classification.
* Revisit foundational CNN architectures such as **LeNet**, **AlexNet**, and **VGG** from a structural point of view.

**Project 6: Advanced CNN Architectures**

* Introduce **Transfer Learning** and how pre-trained models can be adapted to new tasks.
* Explain **Data Augmentation** as a technique to improve model generalization.
* Explore advanced architectures such as **ResNet** and **Inception Networks**, focusing on their structural innovations and training efficiency.