Deep Learning From Scratch in NumPy

A Chapter-Scoped Reimplementation of the Goodfellow–Bengio–Courville Foundations

Mohammed Rafeeq Faraaz Shaik

November 2025

Abstract

This report describes an independent deep learning project that reconstructs core parts of the Goodfellow–Bengio–Courville (2016) deep learning textbook in code using only NumPy and standard Python. The implementation is organised as chapter-scoped mini packages that mirror the book's conceptual structure and culminates in a fully functional MNIST digit classification application. The work emphasises mathematical transparency, numerically stable implementations and clean software architecture rather than reliance on automatic differentiation frameworks. I document the design decisions, algorithms, testing strategy and empirical behaviour of the system and I position the project as evidence of readiness for mathematically grounded research level work in machine learning.

1 Motivation and Project Goals

The primary aim of this project was to internalise the mathematical and algorithmic foundations of deep learning by rebuilding them from first principles. Instead of learning only through high level frameworks, I reconstructed the main ingredients of feedforward neural networks as presented in *Deep Learning* by Goodfellow, Bengio and Courville [1] and bound them into an executable code base.

The project had four concrete goals:

G1 Conceptual fidelity: track the structure of the textbook at chapter and section level and ensure that every major mathematical object has a direct and readable code counterpart.

G2 NumPy only: avoid automatic differentiation, high level optimisers and neural network libraries. All gradients, losses and updates are coded explicitly.

G3 Composable APIs: expose each chapter as a small but coherent Python package so that later chapters compose earlier ones without bypassing them.

G4 End to end application: demonstrate that these pieces are sufficient to train a simple multi layer perceptron (MLP) on MNIST with competitive test accuracy.

This report summarises the resulting system and highlights the aspects that are most relevant to a research oriented master's programme: mathematical understanding, numerical awareness and disciplined software design.

2 High Level Architecture

2.1 Consolidated implementation

For convenience the full implementation can be consolidated into a single Python file `COMPLETE_CODE_CONSOLI` which contains:

- Chapter 2: linear algebra utilities (shapes, norms, SVD, pseudoinverse, PCA).
- Chapter 3: discrete probability, expectation, variance, covariance and information measures.
- Chapter 4: numerical computing and stable reductions (floating point utilities, logsumexp, log softmax, softplus, gradient checking).
- Chapter 5: basic machine learning plumbing (train/validation/test splits, batching, generalisation diagnostics).
- Chapter 6: feedforward neural network building blocks (linear layers, ReLU, cross entropy from logits, backpropagation identities).
- Chapter 7: weight decay regularisation.
- Chapter 8: optimisation (SGD, momentum, learning rate schedules).
- MNIST application: data loading, model definition, training loop and inference entry points.

The same codebase can also be structured as a multi module repository with a `chapters/` directory and an `app_mnist/` application as described in the accompanying `README.md`. The logical architecture is identical in both layouts.

2.2 Data and gradient flow

Conceptually the system factorises into four layers:

1. **Mathematical primitives** (Chapters 2-4): linear algebra, probability and numerical stability.
2. **Neural network primitives** (Chapters 5-7): loss functions, activations, regularisation and accuracy metrics.
3. **Optimisation layer** (Chapter 8): parameter updates and learning rate schedules.
4. **Application layer** (MNIST): data loading, model wiring, training loop and checkpointing.

Data flows from the MNIST loader into mini batches, then through the MLP forward pass to produce logits. These logits are converted to cross entropy loss with numerically stable log softmax. Backward passes compute gradients by hand for each layer and the optimiser updates parameters in place, optionally including weight decay terms.

3 Mathematical Foundations and Implementations

This section summarises how core textbook concepts are realised in NumPy, and how they are later reused in the MNIST application.

3.1 Chapter 2 – Linear algebra

The linear algebra layer provides safe, well typed wrappers around NumPy that enforce shapes, detect invalid operations and expose key matrix properties.

3.1.1 Shapes and validation

The file defines helper functions such as:

`infer_shape(x)`	$\to$ tuple of dimensions,
`ensure_vector(x)`	$\to$ assert x is 1D,
`ensure_matrix(X)`	$\to$ assert X is 2D.

Guard functions such as `check_same_shape(A, B)` and `check_broadcastable(a_shape, b_shape)` catch inconsistent tensor operations early, which is particularly important once gradients are involved.

3.1.2 Matrix and vector products

The standard products are wrapped as:

$$\begin{aligned} \texttt{mm}(A, B) &= AB, \\ \texttt{mv}(A, v) &= Av, \\ \texttt{vv}(a, b) &= a^\top b, \end{aligned}$$

with explicit dimension checks instead of silently relying on NumPy broadcasting. Affine maps are expressed as

$$\texttt{affine\_map}(X, W, b) = XW + b, \tag{1}$$

which later matches the linear layer semantics in the MLP.

3.1.3 Norms, SVD and condition numbers

Vector and matrix norms implement the usual definitions

$$\|x\|_1 = \sum_i |x_i|, \tag{2}$$

$$\|x\|_2 = \sqrt{\sum_i x_i^2}, \tag{3}$$

$$\|A\|_F = \sqrt{\sum_{i,j} A_{ij}^2}, \tag{4}$$

and the spectral norm is computed via the largest singular value. The SVD wrapper `svd_thin` returns U, S and $V^\top$, and the condition number $\kappa_2(A) = \sigma_{\max}/\sigma_{\min}$ is used to reason about ill conditioned systems.

A pseudoinverse A^+ is implemented via SVD with a small singular value cutoff, and a minimum norm least squares solution is provided as

$$x^\star = A^+ b. \tag{5}$$

3.1.4 Trace, determinant and PCA

Trace, determinant and log determinant are implemented with explicit checks for square matrices and informative error paths when determinants are zero or negative. Principal components analysis is implemented via centred data and thin SVD, exposing utilities to compute explained variance ratios and to project onto the top k components.

These tools are not only useful in isolation but also underpin numerical checks in later chapters, for example in gradient checking and condition number analysis.

3.2 Chapter 3 – Probability and information theory

The probability layer focuses on finite discrete distributions and joint tables.

3.2.1 Discrete distributions and expectation

The helper `_check_probs(p)` enforces non negativity and unit mass. From there, the code implements:

- normalisation from counts to probability mass functions,
- marginals and conditionals from joint tables,
- discrete expectations $\mathbb{E}[X]$ and variances $\mathrm{Var}(X)$,
- covariance between two discrete variables given a joint table.

Empirical mean, variance and covariance are computed from samples using unbiased estimators, which is consistent with statistical practice in later evaluation code.

3.2.2 Entropy, cross entropy and KL divergence

For a discrete distribution p the Shannon entropy is implemented as

$$H(p) = -\sum_i p_i \log p_i, \tag{6}$$

with safe logarithms that clip very small probabilities. Given two distributions p and q, cross entropy

$$H(p, q) = -\sum_i p_i \log q_i \tag{7}$$

and Kullback–Leibler divergence

$$D_{\mathrm{KL}}(p \,\|\, q) = \sum_i p_i \log \frac{p_i}{q_i} \tag{8}$$

are implemented with the same numerically robust primitives.

Although the MNIST model uses a specialised cross entropy from logits defined in the feedforward chapter, these probability functions are valuable for diagnostics and sanity checks.

3.3 Chapter 4 – Numerical computing and gradient checking

3.3.1 Floating point utilities

The module exposes machine epsilon for a given dtype, unit in the last place (`ulp`) and the safe range for $\exp(x)$ before underflow or overflow. A Neumaier compensated summation (`kahan_sum`) is implemented to reduce catastrophic cancellation in long sums.

3.3.2 Stable logsumexp, log softmax and softplus

The core numerically stable reduction is

$$\mathrm{logsumexp}(x) = \log \sum_i e^{x_i} = m + \log \sum_i e^{x_i - m}, \quad m = \max_i x_i, \tag{9}$$

implemented so that even rows that are all $-\infty$ return $-\infty$ rather than `NaN`. Log softmax is then

$$\log \mathrm{softmax}(x)_i = x_i - \mathrm{logsumexp}(x), \tag{10}$$

which is the central building block for the cross entropy loss from logits.

Softplus is implemented in the numerically stable form

$$\text{softplus}(x) = \log\left(1 + e^{-|x|}\right) + \max(x, 0), \tag{11}$$

which avoids overflow when x is large and positive.

3.3.3 Gradient checking

To validate hand derived gradients, a central difference numerical gradient is provided:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + \epsilon e_i) - f(x - \epsilon e_i)}{2\epsilon}, \tag{12}$$

where e_i is the standard basis vector. The function `check_grad` compares an analytical gradient to the numerical one and reports the maximum absolute difference. This tool is used on tiny problems in float64 to verify the correctness of the linear layer and cross entropy backpropagation.

3.4 Chapter 5 – Machine learning basics

3.4.1 Splits, batching and seeding

The module implements:

- **Train/validation/test splits**: given arrays (X, y) and ratios, indices are shuffled using a seeded NumPy generator then split deterministically.
- **Mini batch iterator**: `batch_iterator` yields batches of fixed size over (X, y) with optional shuffling and the ability to drop an incomplete last batch.
- **Seeding**: `set_all_seeds(seed)` sets both Python's `random` and NumPy's RNG to ensure reproducible runs.

These utilities are used directly by the MNIST training loop.

3.4.2 Capacity diagnostics and metrics

Generalisation gap utilities compare training and validation losses or accuracies and flag regimes that look like underfitting or overfitting. An accuracy helper computes top one accuracy from logits and either integer labels or one hot targets.

Simple CSV logging helpers allow experiments to be tracked over epochs without adding external dependencies.

3.5 Chapter 6 – Feedforward networks

Chapter 6 is where the mathematical and numerical groundwork is assembled into an actual neural network.

3.5.1 Linear layer

Given inputs $X \in \mathbb{R}^{B \times D_{\text{in}}}$, weights $W \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$ and bias $b \in \mathbb{R}^{D_{\text{out}}}$, the forward pass computes

$$Z = XW + b. \tag{13}$$

The backward pass for an upstream gradient dZ returns

$$dX = dZW^\top, \tag{14}$$

$$dW = X^\top dZ, \tag{15}$$

$$db = \sum_{i=1}^{B} dZ_i. \tag{16}$$

These formulas are implemented directly in NumPy with caches that store the necessary tensors.

3.5.2 ReLU activation

The ReLU activation is defined elementwise as

$$\mathrm{ReLU}(z) = \max(0, z), \tag{17}$$

with a boolean mask recorded during the forward pass so that the backward pass can set gradients to zero where the activation was inactive.

3.5.3 Stable cross entropy from logits

Given logits $L \in \mathbb{R}^{B \times C}$ and targets y in either index or one hot form, the implementation computes:

1. $\log p = \log \mathrm{softmax}(L)$ using the stable logsumexp routine from Chapter 4.

2. The mean cross entropy

$$\mathcal{L} = -\frac{1}{B} \sum_{i=1}^{B} \log p_{i,y_i}. \tag{18}$$

Importantly, the loss is invariant to adding a constant per row to the logits, which is tested numerically.

3.5.4 Backpropagation identity

The gradient of the mean cross entropy with respect to logits has the closed form

$$\frac{\partial \mathcal{L}}{\partial L_{ij}} = \frac{1}{B} \left(\mathrm{softmax}(L)_{ij} - \mathbf{1}\{j = y_i\} \right). \tag{19}$$

This identity is implemented directly and becomes the starting point of the backward pass through the MLP. It has been checked against finite differences on small problems.

3.6 Chapter 7 – Regularisation

The regularisation module implements L2 weight decay.

Given model parameters w (excluding biases) and a coefficient $\lambda \geq 0$, the L2 penalty is

$$\Omega(w) = \frac{\lambda}{2} \sum_{j} w_j^2, \tag{20}$$

and its gradient is simply λw. The code distinguishes weights from biases by both name and shape and by default does not penalise one dimensional parameters, following the standard practice that regularising biases often leads to unnecessary underfitting.

At training time the loss becomes

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \Omega(w), \tag{21}$$

and the gradient contribution λw is added into the backpropagated gradients before the optimiser step.

3.7 Chapter 8 – Optimisation

3.7.1 Vanilla stochastic gradient descent

The simplest optimiser applies the update

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}, \tag{22}$$

where $\eta > 0$ is the learning rate. The implementation works over either dictionaries or lists of parameter arrays and checks that gradient shapes match parameter shapes. Entries with gradient `None` are skipped, which is convenient for freezing parameters.

3.7.2 Momentum and Nesterov momentum

Polyak momentum maintains a velocity v for each parameter and updates it as

$$v \leftarrow \beta v + g, \tag{23}$$

$$\theta \leftarrow \theta - \eta v, \tag{24}$$

where g is the current gradient and $\beta \in [0, 1)$ is the momentum coefficient. The code also supports Nesterov style momentum where the step uses $\beta v + g$ instead of v.

Velocity states mirror the structure of the parameters and are updated in place. This optimiser is used in the MNIST experiments, with momentum improving convergence speed and final accuracy compared to vanilla SGD.

3.7.3 Cosine learning rate schedule

For completeness a cosine annealing schedule with optional linear warmup is implemented:

$$\eta_t = \begin{cases} \eta_{\max} \dfrac{t}{T_{\text{warmup}}}, & t < T_{\text{warmup}}, \\ \eta_{\min} + \dfrac{1}{2}(\eta_{\max} - \eta_{\min}) \left(1 + \cos\left(\pi \dfrac{t - T_{\text{warmup}}}{T_{\text{total}} - T_{\text{warmup}}}\right)\right), & \text{otherwise}, \end{cases} \tag{25}$$

where t is the current step, T_{total} is the total number of steps and $\eta_{\min}$ is the final learning rate. The MNIST training loop can be configured to use either a fixed learning rate or this schedule.

4 MNIST Application

4.1 Data loading and preprocessing

MNIST is loaded directly from the original binary `idx` format using Python's `gzip` and `struct` modules. If the files are missing the loader attempts to download them from a public mirror.

Images are reshaped to vectors of length 784 and normalised to $[0, 1]$ by dividing by 255. A standard validation split is carved out of the training set. For environments without network access a synthetic digits dataset is provided, where each class corresponds to a different block pattern plus noise.

A separate preprocessing function converts arbitrary PNG or JPEG files into the MNIST format for inference: images are converted to grayscale, resized to 28×28, flattened and scaled to $[0, 1]$.

4.2 Model architecture

The MNIST classifier is a shallow MLP with one hidden layer:

$$h = \mathrm{ReLU}(XW_1 + b_1), \tag{26}$$

$$\ell = hW_2 + b_2, \tag{27}$$

where $X \in \mathbb{R}^{B\times 784}$, $h \in \mathbb{R}^{B\times H}$ and $\ell \in \mathbb{R}^{B\times 10}$. The default hidden size H is 128 for quick experiments, with 512 used for higher accuracy runs. Weights are initialised with Glorot uniform initialisation and biases are initialised to zero.

4.3 Training loop

The training pipeline is:

1. Seed RNGs using `set_all_seeds`.
2. Load data and split into train, validation and test subsets.
3. Initialise MLP parameters.
4. For each epoch:
   - 4.1. Iterate over mini batches using `batch_iterator`.
   - 4.2. For each batch, compute logits and cross entropy loss.
   - 4.3. Backpropagate gradients through the MLP.
   - 4.4. Add L2 gradients if weight decay is enabled.
   - 4.5. Update parameters via SGD or momentum.
5. After each epoch, evaluate loss and accuracy on validation and test sets.
6. Save the best model parameters according to validation accuracy into a `.npz` checkpoint.

Inference runs reuse the same forward pass and load the saved parameters. Predicted digit and confidence are reported, together with the full probability vector.

4.4 Hyperparameters and empirical behaviour

Table 1 summarises a typical configuration used for MNIST.

Hyperparameter	Value
Hidden units	512
Batch size	256
Optimiser	Momentum SGD
Learning rate	0.2
Momentum coefficient	0.9
Weight decay λ	10^{-4}
Epochs	20
Activation	ReLU
Initialisation	Glorot uniform

Table 1: Example hyperparameters for MNIST experiments.

With configurations of this form the model reaches test accuracies in the high ninety percent range, which is consistent with known baselines for shallow MLPs on MNIST. Training and validation losses decrease monotonically after a brief warmup, and generalisation gaps remain modest when weight decay is enabled.

5 Testing and Verification

A key objective of the project was to treat the code as a small numerical laboratory rather than a one off script. The following checks are in place:

- **Unit tests for numerical invariants:** softmax rows sum to one, cross entropy is invariant under per row logits shifts, and extreme values of inputs do not produce `NaN` or `inf`.
- **Gradient checks:** the gradients of the linear layer and the cross entropy from logits match central difference approximations within a tolerance when evaluated on small random problems in float64.
- **Regularisation behaviour:** L2 penalties are applied only to weights and not biases and the contribution to the loss and gradients is verified in isolation.
- **End to end smoke test:** a short run on a small subset of MNIST is used to confirm that cross entropy decreases and accuracy increases from random chance.

These tests are designed to approximate the level of rigour expected when building research code that others will later extend.

6 Skills Demonstrated and Limitations

6.1 Skills and preparation for further study

This project demonstrates several capabilities that are directly relevant to a research oriented master's degree in computer science and machine learning:

- The ability to translate textbook mathematics into working code, maintaining a clear mapping between equations and implementations.
- Awareness of numerical issues in floating point computation and the discipline to use stable formulations.
- Comfort with linear algebra, probability and optimisation at the level required to reason about model behaviour without relying on black box libraries.
- Software engineering discipline in designing small composable modules with explicit APIs and regression tests.

These skills form a solid platform for more advanced work on convolutional architectures, sequence models, probabilistic models or numerical methods for partial differential equations.

6.2 Limitations and future work

The project is intentionally minimalistic and therefore has limitations:

- The only neural architecture implemented is a fully connected MLP; no convolutional, recurrent or attention based models are included yet.
- The optimisation algorithms are restricted to SGD with momentum; adaptive methods such as Adam are not currently implemented.
- Experiments are limited to MNIST and synthetic variants; more challenging datasets would exercise the numerical and optimisation layers more fully.

- The system does not include automatic differentiation, so extension to deeper networks requires explicit gradient derivation.

Future extensions that build directly on this foundation include:

- Adding convolutional layers and pooling operations using the same chapter scoped design.
- Implementing simple sequence models for text or time series to connect with recurrent and attention based architectures.
- Bridging this framework with numerical solvers for differential equations as part of hybrid machine learning and scientific computing research.

7 Conclusion

This project reconstructs the core pipeline of supervised learning with neural networks using only NumPy and standard Python. By mirroring the structure of a canonical deep learning textbook and insisting on explicit gradients, numerically stable reductions and disciplined code organisation, it serves both as a learning vehicle and as a small but complete framework.

For admissions committees it provides concrete evidence that I can move comfortably between mathematical definitions, numerical considerations and executable code. It also shows that I am willing to engage deeply with foundational material to prepare for more advanced research in machine learning and related areas.

References

[1] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016.