Implementing a Neural Network from Scratch with NumPy: Training, Optimization, and Experimentation

November 25, 2025

Students:

Chinmay Prasad Dongarkar s250155@dtu.dk
Ng Yuhang Dilon s252047@dtu.dk
Arham Aziz Noman s250173@dtu.dk

Christos Belitselis s250292@dtu.dk
Konstantinos Lykostratis 2520269@dtu.dk

Supervised by:
Viswanathan Sankar (viswa@dtu.dk)
Technical University of Denmark (DTU)
Course: 02456 Deep Learning

Abstract

This project implements a fully connected feedforward neural network (FFNN) from first principles using only NumPy, without relying on deep learning frameworks like TensorFlow or PyTorch. The implementation includes manual forward and backward propagation, multiple optimization algorithms (SGD, Momentum, RMSProp, Adam), and L2 regularization. Models will be trained and evaluated on Fashion-MNIST and CIFAR-10 datasets. All experiments will be systematically tracked using Weights & Biases (WandB) to enable reproducible research and comparative analysis. Through hyperparameter sweeps and ablation studies, this project explores the impact of activation functions, weight initialization strategies, and network architectures on model performance. The project bridges theoretical understanding with practical implementation, while also providing deep insights into the mathematical foundations of modern deep learning.

1 Introduction and Motivation

Deep learning frameworks such as TensorFlow and PyTorch have democratized access to neural networks through high-level APIs and automatic differentiation. While these tools accelerate development, they often obscure the fundamental mathematical principles underlying neural network training. Understanding these foundations is critical for debugging complex systems, designing novel architectures and diagnosing numerical instabilities.

Building a neural network from scratch using NumPy offers several educational and practical advantages:

- **Deep Understanding:** Manual implementation of backpropagation reveals how gradients flow through networks and how the chain rule operates in practice.
- **Transparency:** Complete control over every computational step enables experimentation with custom loss functions, activation functions and optimization strategies.
- **Debugging Skills:** Understanding low-level mechanics improves the ability to identify and resolve training issues in production systems.
- **Algorithm Insight:** Implementing optimizers like Adam from scratch demonstrates the mathematics of adaptive learning rates and momentum.

This project combines foundational understanding with modern best practices by integrating Weights & Biases for experiment tracking, hyperparameter optimization, and visualization. The result is a complete deep learning pipeline that maintains transparency while following industry standards for reproducible machine learning research.

The implementation will be evaluated on two benchmark datasets: Fashion-MNIST (grayscale images) and CIFAR-10 (color images). These dataset will provide sufficient complexity to demonstrate overfitting, regularization effects and analyze optimizer behavior while remaining computationally feasible for CPU-based NumPy training.

2 Project Objectives

The primary objectives of this project are summarized as follows:

1. **Design and Implementation:** Develop a fully modular feedforward neural network (FFNN) architecture using `NumPy`, supporting configurable network depth, layer width, activation and loss functions. Implement forward and backward propagation from first principles through analytical gradient derivations.
2. **Optimization and Regularization:** Implement and compare various optimization algorithms, including Stochastic Gradient Descent (SGD), Momentum, RMSProp, and Adamwhile analyzing the impact of activation functions, weight initialization strategies (e.g., Xavier, He), and regularization methods (such as L2 regularization and dropout) on training stability and convergence.
3. **Experimentation and Evaluation:** Train and evaluate the network on benchmark datasets such as Fashion-MNIST and CIFAR-10. Compute key performance metrics including accuracy, precision, recall, and F1-score, and visualize learning dynamics through plots of loss curves, gradient norms, and activation distributions.

4. **Experiment Tracking and Hyperparameter Tuning:** Integrate Weights & Biases (WandB) for comprehensive experiment logging, tracking hyperparameters, metrics, and gradient statistics. Perform systematic hyperparameter optimization using random or Bayesian search to ensure reproducibility and robust performance comparison across models.

Through these objectives, the project will develop both theoretical understanding and practical skills in building, training, and analyzing neural networks without framework abstractions.

3 Methodology and System Design

3.1 System Architecture

The implementation follows a modular, object-oriented design with the following core components. The overall network orchestrates layer construction, forward and backward propagation, and parameter storage.

For a network with L layers, the forward pass computes:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}, \tag{1}$$

$$\mathbf{a}^{[l]} = f^{[l]}(\mathbf{z}^{[l]}), \tag{2}$$

where $\mathbf{a}^{[0]} = \mathbf{x}$ is the input, $\mathbf{W}^{[l]}$ and $\mathbf{b}^{[l]}$ are learnable parameters, and $f^{[l]}(\cdot)$ denotes the activation function of layer l.

- **Layer Module:** Each layer encapsulates its weight matrix, bias vector, activation function, and layer-specific gradient computations. This modular approach allows flexible depth and width configurations.

- **Activation Functions:** The activation module supports modular implementations of ReLU and Softmax, each with their analytical derivatives.

  ReLU:

  $$\text{ReLU}(z) = \max(0, z), \quad \text{ReLU}'(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

  Softmax (output layer):

  $$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

- **Loss Functions:** The network supports both Cross-Entropy and Mean Squared Error (MSE) losses, each optionally combined with L2 regularization for improved generalization.

  L2 Regularization:

  $$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2m} \sum_{l=1}^{L} \|\mathbf{W}^{[l]}\|_F^2 \tag{3}$$

  where λ is the regularization coefficient, m is the number of samples, and $\|\cdot\|_F$ denotes the Frobenius norm.

- **Optimizer Module:** Implements multiple optimization algorithms including SGD, RMSProp, and Adam with configurable learning rates and decay parameters.

  Stochastic Gradient Descent (SGD):

  $$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) \tag{4}$$

  where η is the learning rate and $\nabla_\theta \mathcal{L}$ denotes the gradient of the loss with respect to parameters θ.

- **Trainer:** Manages the training loop, mini-batch processing, validation evaluation, and performance metric computation. It coordinates the overall learning process by integrating forward passes, gradient backpropagation, and optimizer updates.

Figure 1 illustrates the complete training pipeline from data loading through experiment logging.

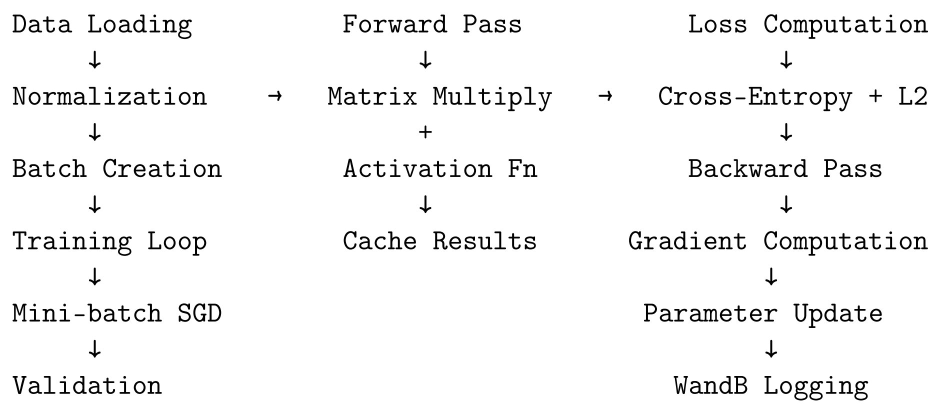

Figure 1: Training pipeline architecture

3.2 Implementation Workflow

Data Preprocessing: Images are normalized to [0,1] by dividing by 255, flattened into vectors (784 for Fashion-MNIST, 3072 for CIFAR-10), and labels are one-hot encoded. Data is split into training (80%), validation (10%), and test (10%) sets.

Forward Propagation: Each layer performs a linear transformation followed by a non-linear activation. Intermediate activations are cached for use during backpropagation.

Loss Computation: Cross-entropy loss is computed for classification tasks, with an optional L2 regularization term added to penalize large weights.

Backward Propagation: Gradients are computed layer-by-layer using the chain rule, starting from the output layer and propagating backwards through the network.

Parameter Updates: The chosen optimizer applies updates to weights and biases based on computed gradients and optimizer-specific state (e.g., momentum buffers, adaptive learning rates).

Training Loop: For each epoch, the training data is processed in mini-batches. After each epoch, validation metrics are computed and logged to WandB.

Evaluation: Final model performance is assessed on the test set using accuracy, per-class precision/recall/F1-score, and confusion matrices.

3.3 Weights & Biases Integration

To ensure experiment reproducibility, transparency, and efficient model tracking, the project integrates the Weights & Biases (WandB) platform. Integration facilitates comprehensive documentation and analysis of all training runs.

The main functionalities of the WandB integration include:

- **Hyperparameter Tracking:** Records all relevant experimental settings such as learning rate, batch size, number of layers, hidden units, optimizer type, and regularization coefficients, ensuring that every configuration is systematically logged.
- **Metric Logging:** Monitors and stores real-time metrics during training and validation, including loss values, accuracy scores, and other performance indicators across epochs.
- **Visualization and Analysis:** Automatically generates visualizations of learning curves, confusion matrices, gradient norms, and weight distributions, providing valuable insights into the network's learning dynamics.
- **Model Checkpointing and Artifact Storage:** Saves model states, parameters, and related experiment artifacts to facilitate reproducibility, rollback, and result comparison between different runs.
- **Hyperparameter Optimization:** Supports automated hyperparameter sweeps using both random and Bayesian optimization strategies to identify the best-performing model configurations.
- **Comparative Experiment Evaluation:** Enables side-by-side comparison of multiple training runs through interactive dashboards, assisting in systematic analysis of architectural and optimization choices.

This integration ensures all experiments are reproducible and enables systematic comparison across different configurations.

4 Experimental Design

4.1 Datasets

Fashion-MNIST: 70,000 grayscale images (60,000 train, 10,000 test) of 10 fashion categories, 28×28 pixels, flattened to 784 features.

CIFAR-10: 60,000 RGB images (50,000 train, 10,000 test) of 10 object categories, $32 \times 32 \times 3$ pixels, flattened to 3072 features.

4.2 Planned Experiments

Table 1 summarizes the seven planned experiments with their focus areas, variables tested, and expected findings.

Table 1: Experimental Plan

Exp	Focus	Variables Tested	Expected Finding
1	Baseline	Simple 1-hidden-layer network	Establish reference performance
2	Activations	ReLU, Sigmoid	ReLU outperform
3	Optimizers	SGD, Momentum, RMSProp, Adam	Adam converges fastest; SGD requires tuning
4	Regularization	L2 coefficients: 0, 0.001, 0.01, 0.1	Optimal λ reduces overfitting
5	Architecture	Depths 1-4, widths 64-512	Deeper networks learn complex features
6	Initialization	Random, He	He initialization best for ReLU
7	Learning Rate	Sweep 10^{-5} to 10^{-1} (Bayesian)	Identify optimal LR per optimizer

4.3 Evaluation Metrics

For each experiment, the following metrics will be computed:

- **Accuracy:** Overall classification accuracy on test set
- **Precision, Recall, F1-Score:** Per-class and macro-averaged metrics
- **Confusion Matrix:** Visualization of misclassification patterns
- **Loss Curves:** Training and validation loss progression over epochs
- **Gradient Statistics:** Mean, variance, and maximum gradient magnitudes
- **Training Time:** Computational efficiency comparison

4.4 Hyperparameter Search

A WandB sweep will explore the following hyperparameter space using Bayesian optimization:

- Learning Rate: $[10^{-5}, 10^{-1}]$ (log scale)
- Batch Size: {32, 64, 128, 256}
- Hidden Units: {64, 128, 256, 512}
- Number of Layers: {1, 2, 3, 4}
- Activation: {ReLU, SoftMax}
- Optimizer: {SGD, Adam, RMSProp}
- L2 Lambda: $[0, 0.1, 0.01, 0.001]$
- Weight Initialization: {Xavier, He}

The objective will be to maximize validation accuracy within 30 epochs.

5 Expected Outcomes

5.1 Technical Deliverables

1. **Complete NumPy Implementation:** A modular, well-documented neural network library supporting arbitrary architectures, multiple activations, loss functions, and optimizers.
2. **Trained Models:** Saved model weights and configurations achieving:
   - Fashion-MNIST: $> 85\%$ test accuracy
   - CIFAR-10: $> 50\%$ test accuracy (without convolutional layers)
3. **Experiment Logs:** 20+ experiments tracked in WandB with comprehensive metrics, hyperparameters, and visualizations.
4. **Analysis Report:** Jupyter notebooks and written report documenting findings, comparative analysis, and insights into neural network behavior.
5. **Codebase:** Clean, tested Python code with documentation, unit tests for gradient checking, and usage examples.

5.2 Learning Outcomes

This project will develop the following competencies:

- Deep mathematical understanding of backpropagation and gradient-based optimization
- Proficiency in numerical computing and matrix operations with NumPy
- Experience with systematic experimentation, ablation studies, and hyperparameter tuning
- Skills in reproducible machine learning research using modern tracking tools
- Ability to diagnose and resolve training issues (vanishing gradients, overfitting, numerical instability)
- Insight into design decisions underlying modern deep learning frameworks

6 Tools and Project Timeline

6.1 Tools and Libraries

The project is developed using Python 3.8 or higher with essential libraries like NumPy (v1.21+) for data handling. For data visualization, it utilizes Matplotlib and Seaborn. Weights Biases is used for experiment tracking to monitor model performance and results. The main datasets include Fashion-MNIST and CIFAR-10, accessed through Keras or direct download. Development and testing are carried out in Jupyter Notebook or JupyterLab, with version control via Git and testing through pytest. The project runs on a local CPU with at least 8 GB RAM or can be executed using Google Colab for cloud-based computing.

7 References

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. https://www.deeplearningbook.org/

2. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv:1412.6980*. https://arxiv.org/abs/1412.6980

3. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. *ICCV 2015*.

4. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *AISTATS 2010*.

5. Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv:1609.04747*. https://arxiv.org/abs/1609.04747

6. Weights & Biases Documentation. (2024). https://docs.wandb.ai/

7. Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *arXiv:1708.07747*.

8. Krizhevsky, A., & Hinton, G. (2009). *Learning multiple layers of features from tiny images*. Technical Report, University of Toronto.