# Deep Learning Course Project

From-Scratch Foundations, Optimization, CNNs, and Modern Architectures

Zinetov Alikhan Yernur Bidollin

IT-2301

**Abstract**

This project studies deep learning systems across four sections: (1) multilayer perceptrons implemented from scratch in NumPy, (2) optimization and training, including derived backpropagation equations, numerical gradient checking, and from-scratch optimizers (SGD, RMSprop, Adam), (3) convolutional neural networks implemented in NumPy with pooling strategy comparison and receptive field analysis, and (4) modern architectures, where we compare plain networks to residual networks and demonstrate transfer learning with a pretrained ResNet18 in PyTorch. We provide training curves, confusion matrices, learning-rate schedule plots, gradient magnitude/flow analysis, and qualitative filter/feature-map visualizations.

## Contents

**1 Introduction** **5**
1.1 Problem Statement . . . . . 5
1.2 Motivation . . . . . 5
1.3 Project Objectives . . . . . 5

**2 Background** **5**
2.1 Neural Network Fundamentals . . . . . 5
2.2 Backpropagation . . . . . 5
2.3 Optimization Algorithms . . . . . 6
2.4 Convolutional Neural Networks and Residual Learning . . . . . 6

**3 Methodology** **6**
3.1 Datasets . . . . . 6
3.2 Model Architectures (High-Level) . . . . . 6
3.3 Training Procedure . . . . . 6
3.4 Evaluation Metrics . . . . . 6

**4 Implementation Details** **7**
4.1 Section 1: Foundations of Deep Learning (NumPy) . . . . . 7
4.1.1 MLP Forward/Backward . . . . . 7
4.1.2 Universal Approximation (Sin Regression) . . . . . 7
4.2 Section 2: Optimization and Training (NumPy) . . . . . 7
4.2.1 Backpropagation for 2-Layer Network . . . . . 7
4.2.2 Gradient Checking . . . . . 7
4.2.3 Optimizer Comparison (SGD vs RMSprop vs Adam) . . . . . 8
4.2.4 Learning-Rate Schedules . . . . . 8
4.2.5 Gradient Magnitude Plots . . . . . 8
4.3 Section 3: Convolutional Neural Networks (NumPy) . . . . . 8
4.3.1 CNN From Scratch . . . . . 8
4.3.2 Pooling Strategy Comparison . . . . . 8
4.3.3 Receptive Field Study . . . . . 8
4.3.4 Filter and Feature Map Visualization . . . . . 8
4.3.5 Confusion Matrix . . . . . 9
4.4 Section 4: Modern Architectures and Transfer Learning (PyTorch) . . . . . 9
4.4.1 Plain vs Residual Network . . . . . 9
4.4.2 Gradient Flow / Magnitude Analysis . . . . . 9
4.4.3 Transfer Learning (Pretrained ResNet18) . . . . . 9
4.4.4 Data Augmentation Impact . . . . . 9

**5 Experiments and Results** **9**
5.1 Summary of Key Numerical Results . . . . . 9

**6 Discussion** **10**
6.1 Analysis . . . . . 10
6.2 Limitations . . . . . 11
6.3 Future Work . . . . . 11

**7 Supplementary Materials** **11**

**8 Individual Contributions** **11**

**9 Conclusion** **11**

**10 References** **12**

**A Project Structure (Brief)** **12**

# 1 Introduction

## 1.1 Problem Statement

The assignment requires implementing and analyzing fundamental deep learning components with a focus on correct mathematical reasoning and reproducible experiments. We must cover four core areas: foundational MLPs, optimization/backpropagation, CNNs, and modern architectures with transfer learning.

## 1.2 Motivation

Building networks from scratch clarifies how gradients propagate, why numerical stability matters (e.g., softmax), and how architectural choices (pooling vs. residual connections) affect convergence and generalization. This understanding is critical for both debugging and principled model design.

## 1.3 Project Objectives

- Implement MLPs (classification/regression) and multiple activation functions with forward/backward passes.
- Derive and implement backpropagation for a 2-layer network; verify via numerical gradient checking.
- Implement SGD, RMSprop, and Adam updates from scratch and compare convergence.
- Implement CNN layers (convolution + pooling + fully connected) using NumPy; achieve strong MNIST accuracy.
- Compare max pooling vs. average pooling; analyze receptive fields.
- Compare plain vs. residual architectures; analyze gradient flow and training stability.
- Apply transfer learning using a pretrained network and compare to from-scratch training.
- Provide required visualizations (curves, confusion matrices, LR schedules, gradient magnitude plots, and filter/feature maps).

# 2 Background

## 2.1 Neural Network Fundamentals

A neural network layer performs an affine transformation followed by a non-linearity:

$$z = Wx + b, \qquad a = \phi(z).$$

Stacking layers increases representational power. For classification, softmax maps logits to class probabilities.

## 2.2 Backpropagation

Backpropagation computes gradients efficiently using the chain rule through the computational graph. The classic formulation was popularized in early work on learning representations [?].

### 2.3 Optimization Algorithms

Training updates parameters to reduce loss. SGD performs a direct step along the negative gradient, while adaptive methods such as RMSprop and Adam normalize or re-scale updates using gradient statistics [?]. Practical training is often improved by learning-rate schedules.

### 2.4 Convolutional Neural Networks and Residual Learning

CNNs exploit locality and weight sharing, making them effective for images. Residual connections improve gradient propagation in deeper networks and enable stable training [?]. A comprehensive overview is available in standard references [?].

## 3 Methodology

### 3.1 Datasets

Our experiments use:

- **MNIST**: 70,000 grayscale digit images (28×28).
- **Oxford-IIIT Pet (binary setting)**: used in Section 4 for cat-vs-dog style classification via torchvision download.

### 3.2 Model Architectures (High-Level)

- **Section 1 (NumPy MLP)**: fully-connected network for MNIST classification + toy XOR + universal approximation for $\sin(x)$ regression.
- **Section 2 (NumPy 2-layer net)**: input → hidden → output, with from-scratch optimization algorithms and learning-rate schedules.
- **Section 3 (NumPy CNN)**: convolution + pooling + fully-connected classifier for MNIST; includes filter and feature-map visualizations.
- **Section 4 (PyTorch)**: plain vs. residual network comparison; transfer learning using pre-trained ResNet18 (frozen backbone vs. fine-tune).

### 3.3 Training Procedure

Sections 1–3 use mini-batch training loops with explicit forward/backward computations. Section 4 uses a manual training loop with PyTorch autograd and a custom update step, while keeping the gradient descent logic explicit (no high-level trainer).

### 3.4 Evaluation Metrics

We report:

- Accuracy (classification)
- Cross-entropy loss (classification)
- MSE (regression)

- Confusion matrices
- Learning-rate schedule plots
- Gradient magnitude / gradient flow plots

## 4 Implementation Details

### 4.1 Section 1: Foundations of Deep Learning (NumPy)

#### 4.1.1 MLP Forward/Backward

We implemented multi-layer forward propagation for hidden layers with configurable activation functions (sigmoid/tanh/ReLU). The backward pass applies the chain rule layer-by-layer to compute $dW$ and $db$.

#### 4.1.2 Universal Approximation (Sin Regression)

We trained MLPs of increasing capacity to approximate $\sin(x)$ over $[-2\pi, 2\pi]$. Final test MSE for three sizes was:

- small: 0.0666
- medium: 0.0533
- big: 0.0101

### 4.2 Section 2: Optimization and Training (NumPy)

#### 4.2.1 Backpropagation for 2-Layer Network

We derived gradients for a 2-layer network with softmax + cross-entropy. For softmax probabilities $\hat{y}$ and labels $y$, the gradient for logits is:

$$\frac{\partial L}{\partial z} = \frac{\hat{y} - y}{N}. \tag{1}$$

This simplifies the output-layer gradient and improves numerical stability.

#### 4.2.2 Gradient Checking

We implemented numerical gradients via centered finite differences:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta + h) - L(\theta - h)}{2h}.$$

After fixing dtype issues (using float64 during checking), the relative errors became very small:

- $W1$: $3.50 \times 10^{-9}$
- $b1$: $9.90 \times 10^{-9}$
- $W2$: $2.80 \times 10^{-9}$
- $b2$: $1.71 \times 10^{-9}$

### 4.2.3 Optimizer Comparison (SGD vs RMSprop vs Adam)

With a constant schedule, we observed fast convergence for RMSprop and Adam compared to SGD. Example final test accuracy after 12 epochs (constant schedule) showed RMSprop/Adam near $\approx 0.97$ while SGD improved more slowly.

### 4.2.4 Learning-Rate Schedules

We implemented and visualized:

- constant LR
- step decay (every 5 epochs $\times 0.5$)
- exponential decay ($\gamma = 0.97$)

Additionally, we logged LR per epoch to produce schedule plots.

### 4.2.5 Gradient Magnitude Plots

We compute gradient norms (e.g., $\|dW1\|_2, \|dW2\|_2$ per step/epoch) to visualize training dynamics and compare stability across optimizers/schedules.

## 4.3 Section 3: Convolutional Neural Networks (NumPy)

### 4.3.1 CNN From Scratch

We implemented convolution, pooling, and fully connected layers. The model trains on MNIST and produces learning curves.

### 4.3.2 Pooling Strategy Comparison

We compared MaxPool vs AvgPool under the same training settings. Example test accuracy after 5 epochs:

- MaxPool: 0.9400
- AvgPool: 0.9117

Max pooling consistently performed better in our runs.

### 4.3.3 Receptive Field Study

For our CNN, the theoretical receptive field after the second pooling layer was:

$$\text{RF} = 10 \times 10, \quad \text{effective stride} = 4.$$

This gives an interpretable view of how much of the input contributes to deeper activations.

### 4.3.4 Filter and Feature Map Visualization

We visualized learned $3 \times 3$ conv filters and feature-map responses across layers (Conv1, Pool1, Conv2, Pool2), showing edge-like detectors and progressively more abstract responses.

#### 4.3.5 Confusion Matrix

We computed a confusion matrix for the MNIST CNN. The matrix is strongly diagonal, indicating correct classification dominates, with minor confusions between visually similar digits.

### 4.4 Section 4: Modern Architectures and Transfer Learning (PyTorch)

#### 4.4.1 Plain vs Residual Network

We trained a PlainNet and a SmallResNet for 3 epochs. Observed validation accuracy:

- PlainNet: val_acc $\approx 0.684$
- SmallResNet: val_acc $\approx 0.692$

Residual connections slightly improved validation accuracy and provided healthier gradient flow.

#### 4.4.2 Gradient Flow / Magnitude Analysis

We logged gradient norms of early vs late layers during training. Compared to PlainNet, the residual network maintained stronger gradients across depth, indicating improved signal propagation.

#### 4.4.3 Transfer Learning (Pretrained ResNet18)

We compared three settings:

- Training from scratch (SmallResNet): val_acc $\approx 0.692$
- Pretrained ResNet18 (frozen backbone): val_acc $\approx 0.810$
- Pretrained ResNet18 (fine-tune layer4 + fc): val_acc $\approx 0.902$

Fine-tuning delivered the best performance within a short training budget.

#### 4.4.4 Data Augmentation Impact

We compared training with and without augmentation. In a 3-epoch experiment:

- no augmentation: val_acc $\approx 0.690$
- with augmentation: val_acc $\approx 0.668$

With very few epochs, augmentation can initially slow learning; longer training typically reveals generalization benefits.

## 5 Experiments and Results

### 5.1 Summary of Key Numerical Results

Table 1 aggregates representative results from our runs.

Table 1: Representative results from our implementation

| Section | Experiment | Result |
|---|---|---|
| 1 | Sin approximation (big) | test MSE = 0.0101 |
| 2 | Gradient check | rel err $\sim 10^{-9}$ |
| 3 | Pooling comparison | max=0.9400, avg=0.9117 |
| 4 | Transfer learning (fine-tune) | val_acc = 0.9020 |

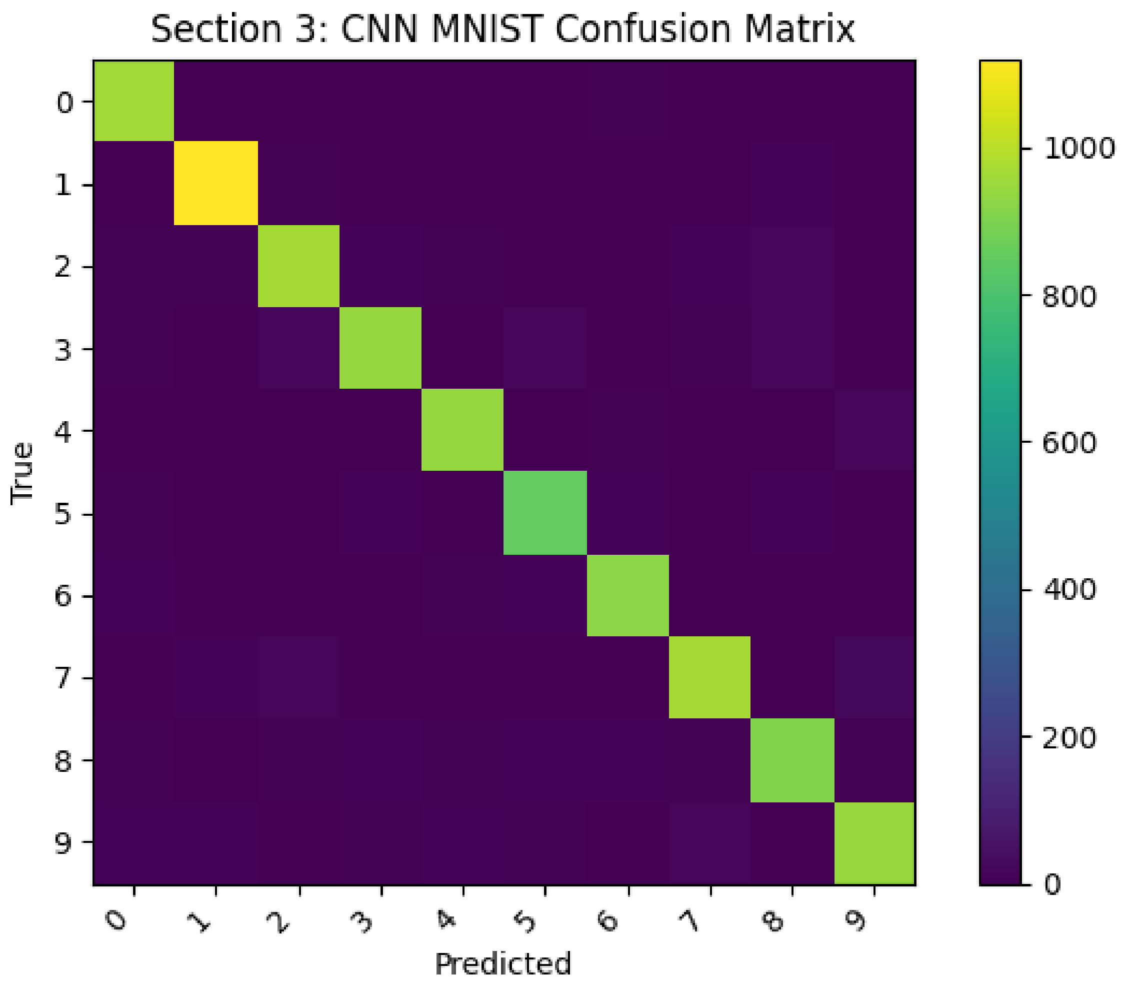

Figure 1: Section 3: CNN confusion matrix on MNIST (example path).

**Example figure block (edit filenames):**

## 6 Discussion

### 6.1 Analysis

Across sections, our experiments highlight consistent patterns:

- Larger models approximate complex functions better (sin regression).
- Numerical stability and dtype control matter for gradient checking.
- Adaptive optimizers converge faster than vanilla SGD in the same epoch budget.
- Max pooling provides stronger performance than average pooling in our CNN setup.
- Skip connections improve gradient flow and typically increase validation accuracy.
- Transfer learning dramatically improves accuracy with limited training time.

### 6.2 Limitations

- Short training runs in Section 4 due to time/compute constraints.
- Some comparisons would benefit from multi-run averaging (variance estimation).
- Checkpoint export and supplementary packaging must be added explicitly.

### 6.3 Future Work

- Train longer and compare robustness under distribution shift.
- Add systematic hyperparameter sweeps.
- Extend CNN capacity to reach higher MNIST accuracy if required.

## 7 Supplementary Materials

The assignment requests additional deliverables beyond the PDF:

- **Trained checkpoints:** saved parameters for NumPy models (`.npz`) and PyTorch models (`.pt`).
- **Additional visualizations:** exported plots (PNG) including confusion matrices, LR schedule plots, gradient magnitude plots, and feature-map visualizations.

These files should be placed in a folder such as `supplementary/` in the submission zip.

## 8 Individual Contributions

- **Zinetov Alikhan:** Section 1 (MLP foundations, activation studies, sin approximation) and Section 4 (plain vs ResNet, transfer learning, gradient flow, augmentation analysis).
- **Yernur Bidollin:** Section 2 (backpropagation, gradient checking, optimizers, LR schedules, gradient magnitude plots) and Section 3 (CNN from scratch, pooling comparison, filter/feature visualizations, receptive field, confusion matrix).

## 9 Conclusion

We implemented core deep learning methods from scratch and validated correctness via gradient checking. Experiments show the impact of optimization strategies, pooling choices, and skip connections, while transfer learning provides a strong improvement under limited training budgets.

## 10 References

- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *CVPR*, 770-778.
- Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

## A Project Structure (Brief)

- `src/section1_mlp_numpy.py` — MLP (MNIST, XOR, sin regression)
- `src/section2_optim_numpy.py` — 2-layer net, backprop, grad check, optimizers, LR schedules
- `src/section3_cnn_numpy.py` — CNN layers, pooling comparison, receptive field + visualizations
- `src/section4_train_torch.py` — plain vs resnet, transfer learning, augmentation effect
- `tests/test_section1_mlp.py` — pytest unit tests
- `configs/` — configuration files (Section 2)