PRACTICAL ASSIGNMENT: NEURAL NETWORKS AND DEEP LEARNING

Santiago Tobio
Department of Artificial Intelligence Engineering
Universidad de San Andres
Buenos Aires, Argentina
stobio@udesa.edu.ar

November 19, 2025

ABSTRACT

This work presents a from-scratch implementation of a complete training system for multilayer neural networks applied to handwritten character classification on the EMNIST Bymerge dataset (809,555 images, 47 classes with 15:1 imbalance). All fundamental components were manually implemented (forward pass, backpropagation, SGD and Adam optimizers, early stopping) using CuPy for GPU acceleration. A systematic exploration of 12 configurations was conducted, evaluating model capacity (112k to 445k parameters), optimizers (SGD vs Adam), regularization (L2, early stopping), learning rate scheduling, and architectures (deep vs wide). Validation with PyTorch confirmed the correctness of the custom implementation (difference <0.3% in test accuracy). The best model (M3b) incorporates batch normalization, dropout 0.3, and SiLU activation, achieving 88.83% test accuracy (+1.38% over baseline) and demonstrating exceptional robustness to Gaussian noise (87.60% accuracy with $\sigma = 0.1$ vs 80.04% for baseline). Key findings reveal that (i) model capacity is the most critical factor, (ii) Adam outperforms SGD by 6.11 percentage points, (iii) deep architectures slightly outperform wide ones with equivalent capacity, and (iv) batch normalization is crucial for both accuracy and robustness to perturbations.

1 Introduction

This practical assignment addresses the development and implementation of multilayer perceptron (MLP) models from scratch in Python. The main objective is to understand the theoretical and practical foundations behind neural networks and evaluate the performance of different architectures and optimization techniques within this model family.

We evaluate model performance on the EMNIST Bymerge dataset. This dataset is an extension of the classic MNIST, which includes images of handwritten digits and uppercase and lowercase letters, providing an additional challenge for image classification.

The models receive as input a 784-dimensional vector (28x28 flattened pixels) and produce as output a probability for each of the 47 possible classes (digits 0-9, uppercase letters A-Z, and lowercase letters a-z, excluding some letters to avoid confusion). Classification is performed by determining the class with the highest probability in the model output.

To accelerate the training process and leverage GPU matrix computation, we used the CuPy library, which offers a NumPy-like interface but with support for GPU calculations.

Five experimental configurations were defined by varying architecture, optimizer, and regularization techniques. Model **M0** establishes the baseline with 2 hidden layers [128, 64], ReLU activation, softmax at the output, cross-entropy loss, and optimization via SGD with batch size 512, learning rate 0.001, and no momentum. Models **M1** systematically explore the hyperparameter space through 8 variants (M1a-M1h) that evaluate different optimizers (SGD with momentum, Adam), regularization techniques (L2, early stopping), learning rate scheduling (linear and exponential), and alternative architectures (deeper and wider). Model **M2** reimplements the optimal M1 configuration

in PyTorch, allowing validation of the custom implementation. Finally, models **M3** explore modern architectures with PyTorch, incorporating alternative activation functions (GELU, SiLU), batch normalization, and dropout.

Subsequently, we present results on validation and test sets, and evaluate model robustness against Gaussian noise perturbations on test images.

Note: We recommend consulting the Jupyter Notebook `Tobio_Santiago_Notebook_TP3.ipynb` as technical support for this report, which includes complete implementations, interactive visualizations, and detailed results from all experiments.

2 Data Exploration and Preprocessing

The EMNIST Bymerge dataset contains 809,555 images of 28×28 pixels distributed across 47 classes. Figure 1 shows a representative example of each class, evidencing the complexity of the classification problem due to visual similarity between certain characters (e.g., 'O' and '0', 'l' and '1').

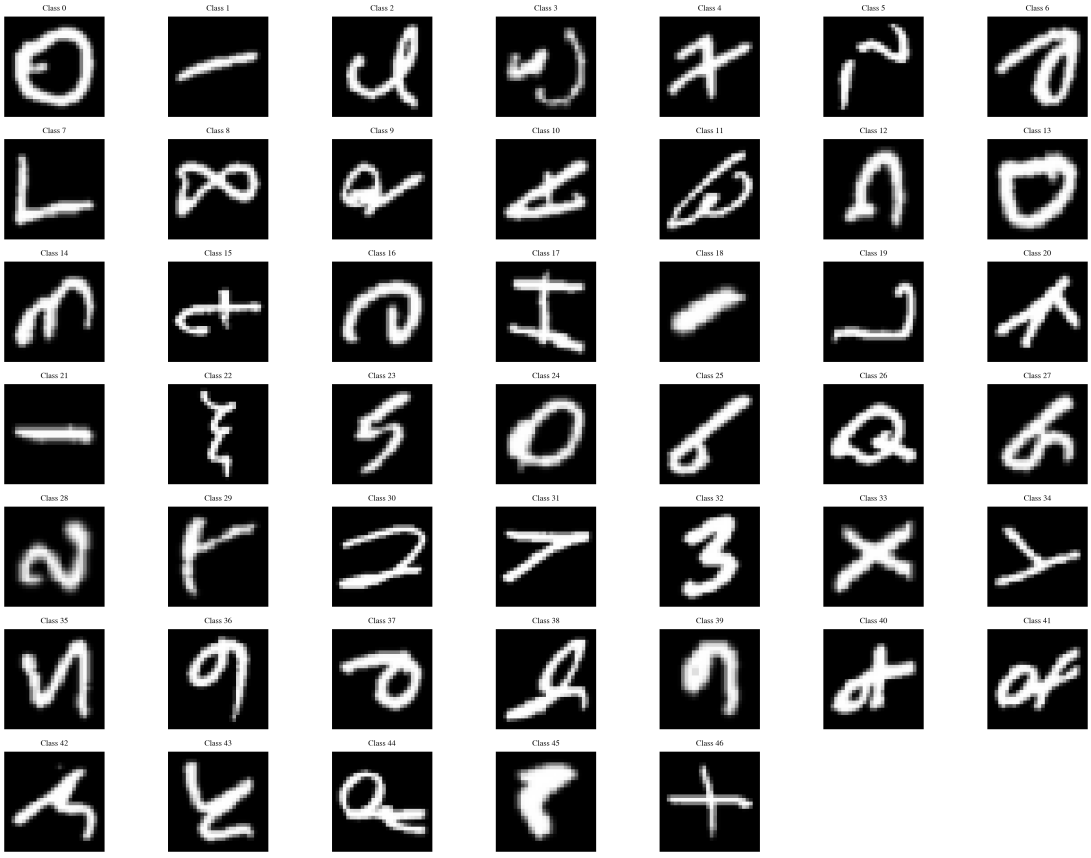

Figure 1: Sample showing one image from each of the 47 classes in the EMNIST Bymerge dataset.

2.1 Class Distribution

The dataset exhibits significant class imbalance, with a mean of 17,225 samples per class and a standard deviation of 13,696. The most represented class (class 1) contains 44,604 samples, while the least represented (class 40) contains only 2,861 samples, resulting in a 15.59:1 ratio. Figure 2 illustrates this heterogeneous distribution.

2.2 Stratified Partitioning

Given the observed imbalance, we implemented a stratified split that preserves class proportions in each subset. The resulting division is presented in Table 1.

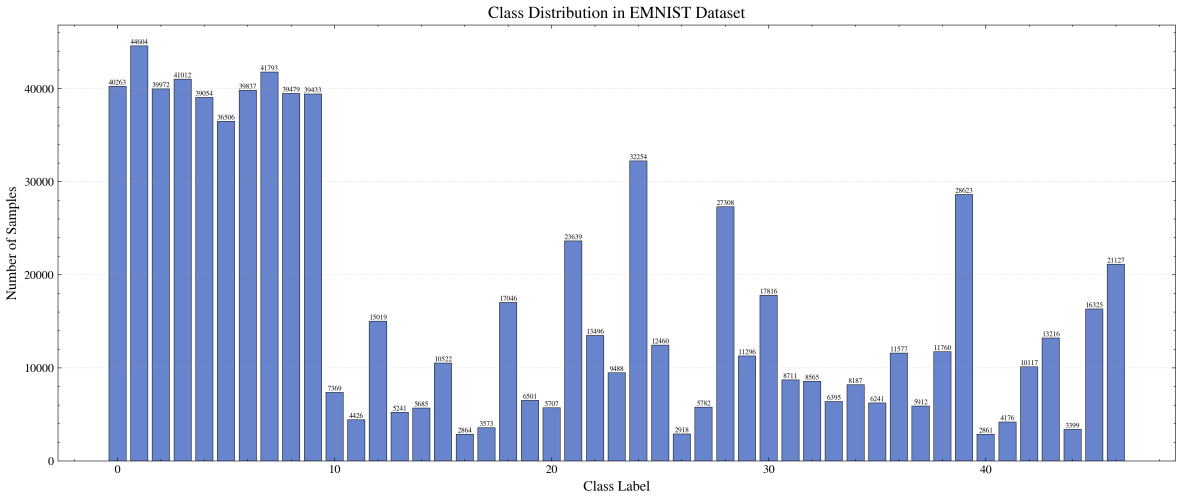

Figure 2: Distribution of samples per class in the EMNIST Bymerge dataset, showing the imbalance present in the data.

Set	Samples	Proportion	Min/class	Max/class
Training	566,667	70%	2,002	31,222
Validation	80,933	10%	286	4,460
Test	161,955	20%	573	8,922

Table 1: Stratified partition of the dataset into training, validation, and test sets.

Stratification ensures that ratios between sets remain constant: Train/Val = 7.00:1, Train/Test = 3.50:1, and Val/Test = 0.50:1. More importantly, the proportions of each class are preserved identically across all three sets, as verified by comparing the percentage distribution of the first 5 classes (all with differences less than 0.01% between sets).

2.3 Preprocessing

Original images (uint8 in range [0, 255]) were normalized by dividing by 255 to obtain float32 values in the range [0, 1]. Each 28×28 image was flattened into a 784-dimensional vector, and labels were encoded in one-hot format (47-dimensional vectors). This preprocessing was applied consistently across all experiments to ensure comparability.

3 Architecture and Theoretical Foundations

A multilayer perceptron (MLP) is a composition of vector functions where each layer applies an affine transformation followed by a nonlinear activation. For layer ℓ, the output is computed as $z^{(\ell)} = W^{(\ell)}a^{(\ell-1)} + b^{(\ell)}$ and $a^{(\ell)} = \sigma(z^{(\ell)})$, where $W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ is the weight matrix, $b^{(\ell)} \in \mathbb{R}^{n_\ell}$ the bias vector, and σ the activation function.

Hidden layers employ the ReLU function, $\text{ReLU}(x) = \max(0, x)$, which introduces nonlinearity enabling approximation of complex functions while its simple gradient facilitates training and mitigates vanishing gradients. The output layer uses softmax, $\text{softmax}(z_i) = e^{z_i} / \sum_{j=1}^{47} e^{z_j}$, which transforms logits into a probability distribution over the 47 classes.

For the multiclass classification problem, we use the categorical cross-entropy loss function, $\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{47} y_{ic}\log(\hat{y}_{ic})$, which measures the discrepancy between the predicted distribution $\hat{y}$ and the true distribution y in one-hot format. Since y_i is one-hot, this simplifies to $\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\log(\hat{y}_{i,c_i})$ where c_i is the true class, heavily penalizing predictions with low probability on the correct class.

3.1 Optimization Algorithms

Training is performed via stochastic gradient descent (SGD) with mini-batches, where gradients are computed through backpropagation. The basic update is $\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t)$, where θ represents all model parameters, η is the learning rate, and $\nabla_\theta \mathcal{L}$ the loss gradient.

To accelerate convergence, some experiments incorporate **momentum**, which maintains accumulated velocity by combining current gradient with previous ones: $v_{t+1} = \beta v_t + \nabla_\theta \mathcal{L}(\theta_t)$ and $\theta_{t+1} = \theta_t - \eta v_{t+1}$, with $\beta = 0.9$ typically. This allows traversing shallow local minima and converging more rapidly.

The **Adam** optimizer adapts the learning rate for each parameter individually, combining momentum with gradient magnitude normalization. It maintains estimates of the first moment m_t and second moment v_t of the gradient, bias-corrected, to update parameters as $\theta_{t+1} = \theta_t - \eta \hat{m}_{t+1} / (\sqrt{\hat{v}_{t+1}} + \epsilon)$ with $\beta_1 = 0.9$, $\beta_2 = 0.999$ (see Appendix B for complete equations).

In some experiments, **learning rate scheduling** is applied so the model explores broadly in early epochs and refines subsequently. We implemented linear decay ($\eta_t = \max(\eta_{\text{min}}, \eta_0 - \alpha \cdot t)$) and exponential decay ($\eta_t = \eta_0 \cdot \gamma^t$ with $\gamma \approx 0.95$).

3.2 Regularization Techniques

To prevent overfitting, various techniques were employed. **L2 regularization** adds a penalty term to the loss, $\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2} \sum_{\ell=1}^{L} ||W^{(\ell)}||_F^2$, which discourages high-magnitude weights by forcing the model to distribute information across multiple weights. **Early stopping** monitors validation loss and halts training if it does not improve for p consecutive epochs, preventing excessive specialization on the training set. PyTorch models additionally explored **dropout** (random deactivation of neurons during training) and **batch normalization** (normalization of activations per mini-batch).

4 Training and Evaluation Methodology

Due to GPU memory limitations (NVIDIA RTX 3050 Ti with 4 GB VRAM), we implemented a hybrid strategy where the dataset remains in system RAM and only mini-batches of 512 samples are transferred on-demand to GPU for executing forward and backward passes. This allows leveraging GPU acceleration without out-of-memory errors (details in Appendix A). The batch size of 512 was determined empirically as the optimal point balancing training speed, convergence stability, and memory usage.

4.1 Unified Experimental Framework

To ensure fair and reproducible comparisons, all experiments run under a unified framework implemented in the `experiments/` module. The `Dataset` class performs the stratified split once at the beginning, and all subsequent experiments receive references to the same train/val/test arrays, eliminating random variations in partitioning. Each experiment is defined by an `ExperimentConfig` object that fully specifies the architecture, hyperparameters, and regularization techniques, enabling reproducibility without code modification. The system automatically constructs all components (model, optimizer, trainer, evaluator) and follows a standardized training flow implemented in `Trainer`: iteration over mini-batches, forward pass, loss computation, backpropagation, parameter updates, and validation evaluation at the end of each epoch.

4.2 Evaluation Metrics

Model evaluation is performed via `evaluate_model` which computes a fixed set of metrics consistently. **Accuracy** measures the proportion of correct predictions, though it can be misleading with imbalanced classes. **Macro F1-Score** averages the F1 of each class without weighting by frequency, treating the class with 2,861 samples and the one with 44,604 equally, which is particularly relevant given the dataset imbalance. **Cross-entropy** evaluates not only correctness but also confidence in predictions, with low values indicating high probability on the correct class. The **confusion matrix** of 47×47 records predictions per class, allowing identification of systematic confusions between visually similar characters. All metrics are computed via centralized implementation in `metrics.py`, ensuring that differences genuinely reflect performance and not variations in calculation.

5 Experimental Results

This section presents the results obtained in each experimental configuration, starting with the baseline model M0 and subsequently comparing variants M1, M2, and M3.

5.1 Model M0: Baseline

Model M0 establishes the baseline performance against which subsequent configurations are compared. Its architecture consists of two hidden layers of 128 and 64 neurons respectively, with ReLU activation. Training was performed via SGD without momentum ($\beta = 0$) with constant learning rate $\eta = 0.001$, batch size of 512, and without applying L2 regularization or early stopping.

5.1.1 Training Curves

Figure 3 shows the evolution of loss and accuracy during the 50 training epochs. We observe that training loss decreases consistently throughout training, while validation loss begins to increase approximately after epoch 30, a characteristic signal of overfitting. Training accuracy continues improving until reaching values close to 90%, while validation accuracy stabilizes around 87% and shows slight oscillations in the final epochs.

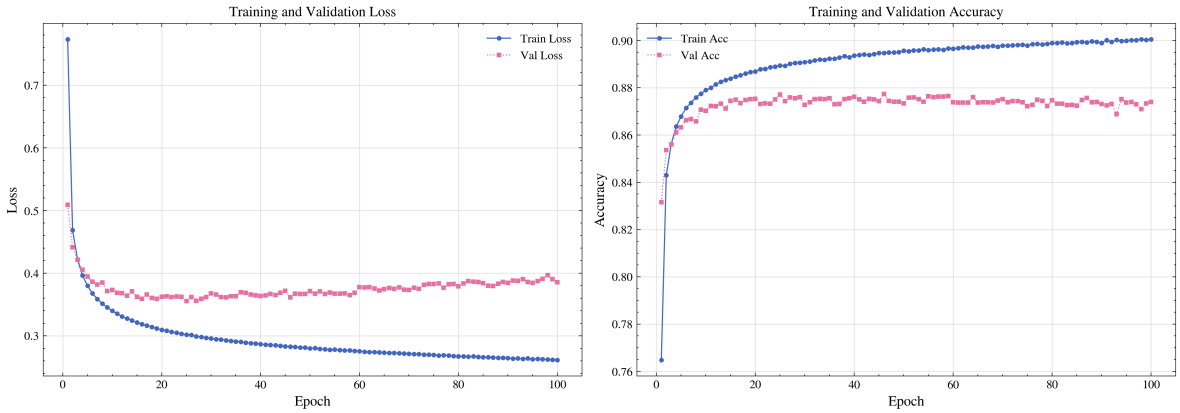

Figure 3: Training curves for model M0. Left: evolution of loss on train and validation. Right: evolution of accuracy. Overfitting is observed from epoch 30 onwards, evidenced by the increase in validation loss while training loss continues decreasing.

The divergence between loss curves after epoch 30 evidences clear overfitting: training loss continues decreasing while validation loss increases, indicating that the model begins memorizing the training set at the expense of generalization. The final gap between train accuracy (90%) and validation accuracy (87%) of approximately 3 percentage points confirms this behavior. This overfitting is expected given the absence of regularization techniques (L2, dropout, early stopping) in M0. The model would have benefited from stopping training around epoch 30-35, when validation loss reaches its minimum.

5.1.2 Confusion Matrix

Figure 4 presents the confusion matrix of model M0 on the test set. The main diagonal concentrates most predictions, indicating that the model generalizes reasonably well. However, bands of systematic confusion are observed in certain regions of the matrix, corresponding to visually similar characters.

Some notable error patterns include confusions between digits and letters that share visual form (e.g., ’0’ vs ’O’, ’1’ vs ’l’ or ’I’), as well as between uppercase and lowercase letters of similar shape. These errors are inherent to the nature of the problem and reflect genuine ambiguities in the data.

5.1.3 Analysis

Model M0 establishes a solid baseline, demonstrating that a simple MLP architecture with two hidden layers is capable of learning effective representations of the EMNIST Bymerge dataset, validating the viability of the implementation and the GPU memory management strategy.

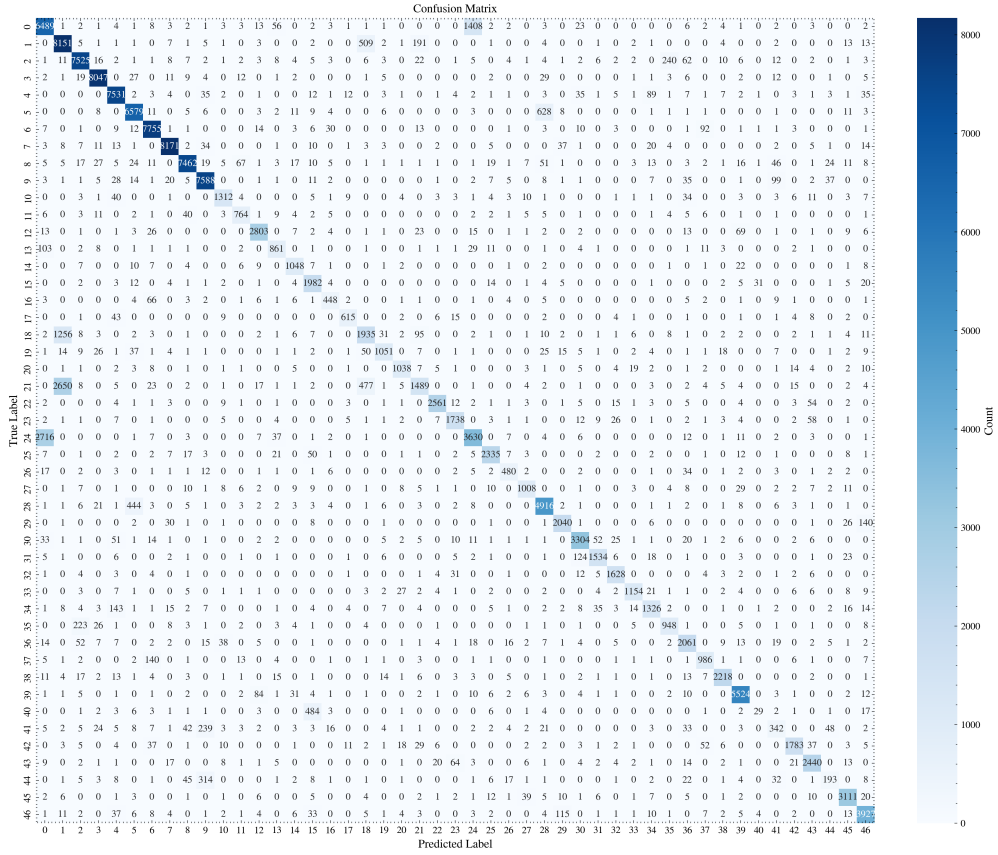

Figure 4: Confusion matrix of model M0 on the test set. Regions off the diagonal reveal systematic confusions between similar characters.

However, the overfitting observed from epoch 30 onwards and the systematic confusion patterns between visually similar characters suggest clear opportunities for improvement. The M1 experiments explore regularization techniques (L2, early stopping), advanced optimizers (Adam), learning rate scheduling, and architectures with greater capacity to address these limitations. Quantitative results on the test set are reported in the final comparison (Section 4.5).

5.2 Model M1: Hyperparameter Search

To improve baseline model M0 performance, we conducted a systematic hyperparameter exploration organizing 8 variants (M1a-M1h) into 4 experimental groups. Each group evaluates a specific dimension of the hyperparameter space while keeping other factors constant, allowing isolation of each technique's effect.

5.2.1 Experimental Design

Experimental groups were defined as follows:

Group 1: Optimizers. We compare SGD with momentum ($\beta = 0.9$) against Adam, both with early stopping (patience=5). This group evaluates whether adaptive optimizers offer advantages over traditional momentum methods in this problem.

Group 2: Regularization. We evaluate the effect of L2 regularization ($\lambda = 0.01$) combined with different optimizers. M1c uses Adam + L2, while M1d combines SGD with momentum + L2 + early stopping. This group determines whether explicit regularization is necessary given the use of early stopping.

Group 3: Learning Rate Scheduling. We explore two learning rate decay strategies: linear (M1e) and exponential (M1f), both with Adam as the base optimizer. This group evaluates whether dynamic learning rate adjustment improves convergence beyond Adam's automatic adaptation.

Group 4: Architecture. We contrast two approaches to increasing model capacity: a deeper network with 3 layers [400, 240, 120] totaling 444,847 parameters (M1g), versus a wider network with 2 layers [370, 370] totaling 445,157 parameters (M1h). Both architectures have approximately the same number of parameters (445k), allowing comparison of depth vs width with equivalent capacity.

5.2.2 Results

Table 2 presents complete results of the 8 M1 variants evaluated on training and validation sets. The best model according to validation accuracy is M1g (88.51%), followed by M1h (88.23%). Notably, M1g also achieves the lowest validation loss (0.3322), indicating better calibrated predictions.

Model	Group	Architecture	Train Acc	Val Acc	Val Loss	Parameters
M1a	Opt.	[128, 64]	83.15	81.41	0.5815	111,791
M1b	Opt.	[128, 64]	89.29	87.52	0.3640	111,791
M1c	Reg.	[128, 64]	89.96	87.89	0.3555	111,791
M1d	Reg.	[128, 64]	84.49	84.30	0.4933	111,791
M1e	Sched.	[128, 64]	89.53	87.64	0.3630	111,791
M1f	Sched.	[128, 64]	89.35	87.67	0.3583	111,791
M1g	Arch.	[400, 240, 120]	**90.66**	**88.51**	**0.3322**	444,847
M1h	Arch.	[370, 370]	90.68	88.23	0.3374	445,157

Table 2: Results of M1 variants on training and validation sets. The best model by validation is M1g with 88.51% accuracy and loss of 0.3322.

5.2.3 Analysis by Experimental Group

Optimizers (M1a vs M1b): Adam (M1b: 87.52% val acc) significantly outperformed SGD with momentum (M1a: 81.41% val acc), demonstrating the advantage of per-parameter adaptive learning rates. The 6.11 percentage point difference validates that Adam is essential for efficient convergence in this problem. Additionally, Adam achieves considerably higher training accuracy (89.29% vs 83.15%), indicating that SGD with momentum struggles to optimize even the training set.

Regularization (M1c vs M1d): L2 regularization combined with Adam (M1c) marginally improved performance (87.89% val acc vs 87.52% for M1b without L2), with slightly lower validation loss (0.3555 vs 0.3640), suggesting better calibrated predictions. Conversely, L2 with SGD+momentum (M1d) degraded performance (84.30% val acc), confirming a negative interaction between momentum and weight decay that requires careful hyperparameter tuning.

Learning Rate Scheduling (M1e vs M1f): Both scheduling strategies (linear: 87.64% val acc, exponential: 87.67% val acc) produced results virtually identical to M1b (87.52%), indicating that Adam's automatic adaptation already provides sufficient dynamic learning rate adjustment. Additional scheduling did not provide significant improvements and adds unnecessary complexity.

Architecture (M1g vs M1h): Increasing capacity from 112k to 445k parameters resulted in the most substantial improvement, raising val accuracy from 87.52% (M1b) to 88.51% (M1g) and 88.23% (M1h). Surprisingly, the deeper architecture (M1g: [400, 240, 120]) slightly outperformed the wider one (M1h: [370, 370]) with 88.51% vs 88.23% val accuracy, suggesting that greater depth may be beneficial when total capacity is equivalent (445k parameters). M1g also achieves the lowest validation loss (0.3322), indicating better calibrated predictions. Both models show similar train accuracy (90.7%), but M1g generalizes marginally better.

5.2.4 Optimal Configuration

Based on validation accuracy (88.51%) and lowest validation loss (0.3322), we select M1g as the optimal configuration. This configuration combines: deep architecture [400, 240, 120] with 444,847 parameters, Adam optimizer ($\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$), batch size 512, early stopping with patience 5, ReLU activation, and no L2 regularization or dropout. The model achieves 90.66% train accuracy and 88.51% val accuracy, with a gap of approximately 2.15 percentage points indicating healthy generalization without severe overfitting. Results on the test set are reported in the final comparison (Section 4.5).

5.3 Model M2: PyTorch Validation

To validate the correctness of the custom implementation, we reimplemented the optimal M1g configuration in PyTorch. Model M2 exactly replicates the architecture [400, 240, 120], Adam optimizer with the same hyperparameters, batch size 512, and early stopping with patience 5. Table 3 presents the comparison between both implementations.

Model	Framework	Train Acc	Val Acc	Val Loss
M1g	Custom (CuPy)	90.66	88.51	0.3322
M2	PyTorch	90.99	88.23	0.3494
Δ (%)		+0.33	−0.28	+5.18

Table 3: Comparison M1g (custom implementation) vs M2 (PyTorch). Test metrics are reported in Section 4.5.

The difference in val accuracy between M1g (custom) and M2 (PyTorch) is only 0.28%, validating the correctness of the from-scratch custom implementation. M2 achieves slightly higher train accuracy (90.99% vs 90.66%), but M1g generalizes marginally better on validation (88.51% vs 88.23%). Validation loss is 5.18% lower in M1g (0.3322 vs 0.3494), indicating better calibrated predictions in the custom implementation. These minimal differences confirm that both implementations are functionally equivalent and that the manually implemented components (forward pass, backpropagation, Adam optimizer) correctly replicate PyTorch's behavior.

5.4 Model M3: Modern Architectures with PyTorch

Models M3 explore modern architectures incorporating advanced regularization techniques and alternative activation functions. We evaluated two variants: M3a with deeper architecture [256, 128, 64, 64], GELU activation, dropout 0.2, and weight decay 0.01; and M3b with wider architecture [400, 400], SiLU (Swish) activation, batch normalization, dropout 0.3, and weight decay 0.01. Both models use Adam with learning rate 0.001 and early stopping with patience 7.

Model	Architecture	Activation	Train Acc	Val Acc	Val Loss
M3a	[256, 128, 64, 64]	GELU	86.94	88.51	0.3188
M3b	[400, 400]	SiLU	**87.54**	**88.59**	**0.3128**

Table 4: Results of M3 models with modern architectures. M3b is selected as best model by lowest validation loss.

M3b slightly outperformed M3a in val accuracy (88.59% vs 88.51%) and achieved the lowest validation loss among all evaluated models (0.3128), indicating exceptionally well-calibrated predictions. Notably, M3b presents considerably lower train accuracy (87.54%) than M1g/M2 (90.7%), evidencing the strong regularization effect of batch normalization and dropout 0.3. This train-val gap of only 1.05 percentage points (87.54% vs 88.59%) is the lowest among all models, suggesting optimal generalization without overfitting. The SiLU (Swish) activation combined with batch normalization proved particularly effective, outperforming GELU in M3a. M3a, despite its greater depth, shows a larger train-val gap (86.94% vs 88.51%, 1.57 pp), indicating that the combination of batch norm + wide architecture is superior to depth + GELU in this problem.

5.5 Final Comparison on Test Set

Table 5 presents the comparison of the four final models evaluated on the test set. This is the only evaluation on test performed during the project, ensuring that no model was tuned based on its performance on this data.

Model	Description	Accuracy	F1-Macro	Cross-Entropy	Improvement vs M0
M0	Baseline	87.45	83.92	0.3862	—
M1g	Best Custom	88.58	85.73	0.3368	+1.13%
M2	PyTorch Validation	88.36	85.56	0.3466	+0.91%
M3b	Best PyTorch Modern	**88.83**	**85.81**	**0.3122**	**+1.38%**

Table 5: Final comparison of the four selected models on the test set.

The best model on test is M3b with 88.83% accuracy and 85.81% macro F1, representing a 1.38 percentage point improvement over baseline M0. M3b also achieves the lowest cross-entropy (0.3122), confirming better calibrated predictions thanks to batch normalization and dropout.

M1g (custom implementation) achieves 88.58% test accuracy, outperforming M2 (PyTorch, 88.36%) by 0.22 percentage points. This small difference (less than 0.3%) completely validates the correctness of the from-scratch custom implementation, demonstrating that manually implemented components faithfully replicate PyTorch's behavior.

The progression M0 (87.45%) $\rightarrow$ M1g (88.58%) $\rightarrow$ M3b (88.83%) evidences incremental improvements: +1.13% from increased capacity and Adam optimizer, and +0.25% additional from modern regularization techniques (batch norm, dropout 0.3, SiLU). The gap between accuracy (88.83%) and macro F1 (85.81%) of 3.02 pp in M3b is slightly smaller than in M0 (87.45% vs 83.92%, 3.53 pp), indicating better balance in performance across imbalanced classes.

5.6 Robustness Analysis Under Perturbations

To evaluate model robustness against noisy data, we perturbed test set images by adding Gaussian noise with standard deviations $\sigma \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$. Table 6 shows the accuracy degradation for each model.

Model	$\sigma = 0.0$	$\sigma = 0.1$	$\sigma = 0.2$	$\sigma = 0.3$	$\sigma = 0.4$	$\sigma = 0.5$
M0	87.45	80.04	52.21	25.98	12.13	6.49
M1g	88.58	77.44	39.95	17.44	8.98	5.49
M2	88.36	76.12	32.55	9.13	3.77	2.33
M3b	**88.83**	**87.60**	**70.34**	**37.77**	**15.83**	**7.49**

Table 6: Accuracy (%) under different levels of Gaussian noise on the test set.

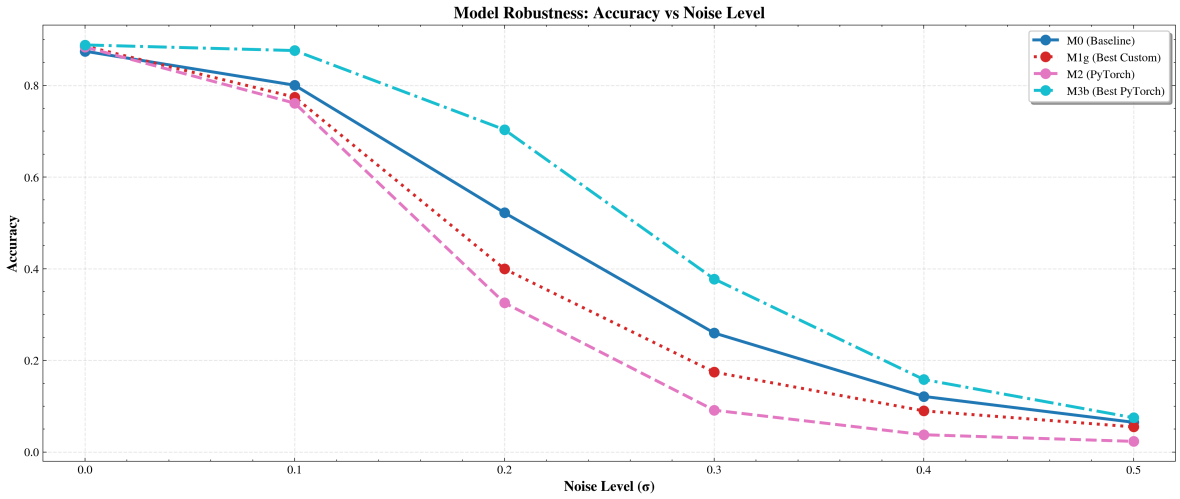

Figure 5: Accuracy degradation as a function of Gaussian noise level. M3b maintains superior performance at all noise levels thanks to batch normalization.

Robustness analysis reveals dramatic differences between models. M3b is consistently the most robust model at all noise levels, maintaining 87.60% accuracy with $\sigma = 0.1$ (only 1.39% degradation) and 70.34% with $\sigma = 0.2$ (20.81% degradation). In contrast, M2 shows the greatest fragility: with $\sigma = 0.2$ it falls to 32.55% (63.16% degradation) and with $\sigma = 0.3$ collapses to 9.13% (89.67% degradation).

The exceptional robustness of M3b is attributed to batch normalization, which normalizes activations even when inputs are perturbed. With high noise ($\sigma \geq 0.3$), M3b maintains average accuracy of 20.36%, widely surpassing M0 (14.87%), M1g (10.64%), and M2 (5.08%). M3b's percentage degradation is systematically lower: 1.39% ($\sigma = 0.1$) vs 8.48% (M0), 13.85% (M2), and 12.58% (M1g).

Surprisingly, M1g and M2, despite sharing equivalent architecture, show very different robustness. M1g consistently outperforms M2 at all high noise levels ($\sigma \geq 0.2$), suggesting that subtle differences in initialization or optimization dynamics between custom and PyTorch implementations can significantly affect robustness to perturbations.

6 Discussion

Systematic exploration of 12 configurations reveals which factors contribute most to performance. Increasing parameters from 112k to 445k resulted in the most substantial improvement (+1.13 pp in test accuracy), exceeding the effect of learning rate scheduling. However, M0 (112k parameters, 87.45%) vs M3b (445k parameters, 88.83%) evidences that architecture and regularization are equally crucial. Adam outperformed SGD with momentum by +6.11 pp, while learning rate scheduling did not provide improvements given that Adam already provides automatic adaptation. With equivalent capacity, deep architectures [400, 240, 120] slightly outperformed wide ones [370, 370] (88.51% vs 88.23% val acc).

The <0.3% difference between M1g (custom, 88.58%) and M2 (PyTorch, 88.36%) validates the from-scratch implementation, confirming that manually implemented components faithfully replicate PyTorch. M3b with batch normalization achieved the best performance (88.83% test acc, 0.3122 CE) and lowest train-val gap (1.05 pp), indicating effective regularization. Robustness analysis reveals that batch normalization dramatically increases noise resistance: M3b maintains 87.60% with $\sigma = 0.1$ (1.39% degradation) vs 80.04% in M0 (8.48%), and with high noise ($\sigma \geq 0.3$) achieves 20.36% average accuracy vs 5.08% in M2.

Limitations include GPU restrictions (4 GB) that prevented larger batch sizes, absence of data augmentation or ensemble methods, and lack of strategies for class imbalance (15:1 ratio) such as class weighting.

7 Conclusions

This work implemented from scratch a complete training system for multilayer neural networks on the EMNIST Bymerge dataset (47 classes, 15:1 imbalance). The progression M0 (87.45% test acc) $\rightarrow$ M1g (88.58%) $\rightarrow$ M3b (88.83%) evidences incremental improvements: +1.13% from increasing parameters (112k to 445k) and Adam, +0.25% additional from batch normalization, dropout, and SiLU. Adam outperformed SGD with momentum by 6.11 pp, while learning rate scheduling did not provide significant improvements.

The <0.3% difference between M1g (custom, 88.58%) and M2 (PyTorch, 88.36%) validates the from-scratch implementation, demonstrating that manually implemented components faithfully replicate professional frameworks. M3b achieved the best test accuracy (88.83%) and exceptional robustness: it maintains 87.60% with $\sigma = 0.1$ (1.39% degradation) vs 80.04% in M0 (8.48%), and with high noise ($\sigma \geq 0.3$) achieves 20.36% average accuracy vs 5.08% in M2, confirming that batch normalization dramatically increases robustness.

The CuPy implementation with hybrid CPU-GPU memory management enabled efficient training of models on limited hardware (4 GB GPU). As future work, it would be valuable to explore convolutional architectures, data augmentation, techniques for imbalanced classes (class weighting, focal loss), and ensemble methods.

A GPU Memory Management

Training neural networks with large-scale datasets presents practical challenges related to available hardware memory. In this work, the equipment used has an NVIDIA RTX 3050 Ti GPU with 4 GB VRAM, an insufficient amount to maintain the complete dataset of 809,555 samples in GPU memory simultaneously.

To resolve this limitation without sacrificing GPU acceleration, we implemented a hybrid memory management strategy: the complete dataset remains in system RAM (using NumPy), while only the mini-batches required in each training iteration are transferred on-demand to GPU memory (using CuPy). This approach allows computationally intensive matrix operations of the forward pass and backward pass to execute on GPU, leveraging its massive parallelism, without incurring out-of-memory errors.

CPU-GPU transfer occurs synchronously at the beginning of each training iteration, where a batch of 512 samples is copied to the GPU, processed, and the resulting gradients are used to update model weights. With this batch size, VRAM consumption during training remains at approximately 1.2 GB, leaving a safe margin relative to the hardware limit.

The selection of batch size 512 responds to a balance between training speed and convergence stability. Smaller batch sizes (128, 256) result in more frequent updates but with noisier gradients and greater CPU-GPU transfer overhead. Larger batch sizes (1024, 2048) could improve gradient stability, but significantly increase memory usage and may lead to out-of-memory errors on limited hardware. The value of 512 was determined empirically as the optimal point for this specific hardware.

B Complete Equations for Adam Optimizer

The Adaptive Moment Estimation (Adam) optimizer maintains estimates of the first moment (mean) and second moment (uncentered variance) of the gradient. The complete updates are calculated as:

$$\begin{aligned} m_{t+1} &= \beta_1 m_t + (1-\beta_1)\nabla_\theta \mathcal{L}(\theta_t) \\ v_{t+1} &= \beta_2 v_t + (1-\beta_2)(\nabla_\theta \mathcal{L}(\theta_t))^2 \\ \hat{m}_{t+1} &= \frac{m_{t+1}}{1-\beta_1^{t+1}} \\ \hat{v}_{t+1} &= \frac{v_{t+1}}{1-\beta_2^{t+1}} \\ \theta_{t+1} &= \theta_t - \eta \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \epsilon} \end{aligned} \tag{1}$$

where m_t and v_t are the estimates of the first and second moment respectively, β_1 and β_2 are the decay factors (typically $\beta_1 = 0.9$, $\beta_2 = 0.999$), $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected estimates, η is the learning rate, and ϵ is a small constant for numerical stability (typically 10^{-8}).

The bias correction ($\hat{m}_t$ and $\hat{v}_t$) is crucial in the first iterations when m_t and v_t are initialized at zero, preventing the estimates from being biased toward zero especially when the decay factors are close to 1.