2025

Deep Learning Fundamentals

MNIST Digits - Hyperparameter Exploration Report

António Cruz (140129), Ricardo Kayseller (95813)

2025-11-02

Table of Contents

0. Project Scope & Objectives

The project focus is to examine how hyperparameter choices shape learning and generalization of a Multilayer Perceptron Network (MLP) on MNIST digits dataset. A Convolutional Neural Network analysis is also included, but only as an optional comparison, to contextualize results from the MLP.

Our main objectives are:

- Systematically vary key hyperparameters (batch size, learning rate, optimizer, weight decay, dropout, scheduler) and quantify their impact on loss, accuracy, and training dynamics.
- Capture metrics and artifacts consistently to enable correct and repeatable comparisons.
- Summarize findings with plots and tables that highlight sensitivity, trade-offs, and recommended settings.
- Optionally, contrast MLP and CNN under identical protocols to isolate architectural effects.

1. Introduction

Hyperparameter choices can shape the learning dynamics and generalization of an Artificial Neural Network (ANN) for handwritten digit recognition. Rather than treating accuracy as a single end-point, the study frames model performance as the outcome of interacting design decisions—such as learning rate, batch size, activation functions, weight initialization, and regularization—that together govern optimization stability and convergence behavior. Understanding these interactions is essential to move from ad-hoc tuning to a principled, reproducible workflow.

We adopt MNIST as an experimental testbed. The dataset consists of 70,000 grayscale images of digits (0–9), each 28×28 pixels, split into 60,000 training and 10,000 test examples. MNIST is intentionally modest in complexity: it is simple enough to allow rapid iteration across many configurations, yet rich enough to reveal the effects of hyperparameters on overfitting, calibration, and class-wise errors. This balance makes it ideal for isolating the contribution of modeling decisions without the confounds of heavy data engineering.

A key premise of the project is that data understanding precedes modeling. Characteristics such as class balance, pixel-intensity distribution, spatial alignment, and morphological variability directly inform preprocessing choices (e.g., normalization constants), architecture constraints (e.g., activation ranges), and augmentation policies (e.g., small rotations vs. translations). Accordingly, we begin with a rigorous exploratory analysis of MNIST to establish (i) whether the train/test splits are statistically consistent, (ii) where variability concentrates within the images, and (iii) which digit pairs are intrinsically similar and therefore prone to confusion. These insights serve as guardrails for the subsequent hyperparameter exploration.

The project's objectives are threefold. First, to characterize MNIST in a way that justifies concrete preprocessing and validation decisions (normalization, stratified splits, and light augmentations). Second, to design a transparent experimental protocol that varies one hyperparameter at a time (and selected interactions), enabling interpretable comparisons across runs. Third, to translate empirical results into actionable guidance—for example, how to set learning rates that avoid divergence, what batch sizes stabilize gradients without harming generalization, or when dropout and weight decay are genuinely helpful on a sparse, well-centered dataset like MNIST.

This ensures that any observed differences in model performance can be confidently attributed to hyperparameter effects rather than latent data artifacts. The remainder of the report follows this logic. Section 2 presents the Data Understanding findings that anchor all modeling choices. Section 3 details the experimental setup and the hyperparameters under study. Section 4 reports results with a focus on training dynamics and class-wise behavior, and Section 5 distills practical recommendations for future ANN workflows.

2. Data Understanding

Before any hyperparameter search, we validated that the dataset used in our experiments is structurally sound, representative, and visually coherent. The intent is practical: ensure that performance differences observed later can be attributed to modeling choices (architecture, learning rate, regularization, batch size) rather than silent data issues. We therefore confirmed shapes at load time, inspected content to verify decoding and centering, and quantified class frequencies across training and test splits. The result is a clean baseline for controlled experimentation, reducing uncertainty about the data pipeline and clarifying which preprocessing decisions are already justified by evidence rather than convention.

2.1 Dataset Overview (structure & loading)

The dataset was loaded via torchvision.datasets.MNIST, producing 60,000 training and 10,000 test images, each a 28×28 single-channel grayscale array with labels in {0,…,9}. A structural pass verified consistent shapes across batches, correct label coverage in both splits, and a stable iteration path with no decoding errors. Pixel values arrive as uint8 in [0, 255]; converting to float32 and scaling to [0, 1] preserves dynamic range and prepares the inputs for standard z-score normalization used later. This uniform, well-framed input format eliminates the need for ad-hoc resizing or multi-channel tricks and lets us compare modeling choices on equal footing.

2.2 Sample Visualization & Sanity Check

Quantified representativeness was made by counting the number of examples per class in both splits. The distribution is near-uniform: training classes lie roughly in the 5.4k–6.8k range per digit, and test classes sit at about 1.0k each, with only minor fluctuations and no systematic skew. This balance removes the need for class reweighting or resampling and simplifies the experimental protocol. It also improves interpretability: if a particular configuration boosts recall for the thin-stroke class “1” without harming thick-stroke classes (e.g., “0”, “8”), the effect can be attributed to modeling rather than class imbalance. For validation we will keep stratified splits to preserve this balance when comparing configurations.

2.3 Class Distribution & Balance

We quantified representativeness by counting the number of examples per class in both splits. The distribution is near-uniform: training classes lie roughly in the 5.4k–6.8k range per digit, and test classes sit at about 1.0k each, with only minor fluctuations and no systematic skew. This balance precludes the need for class reweighting or resampling and simplifies the experimental protocol. It also improves interpretability: if a particular configuration boosts recall for the thin-stroke class “1” without harming thick-stroke classes (e.g., “0”, “8”), the effect can be attributed to modeling rather than class imbalance. For validation we will keep stratified splits to preserve this balance when comparing configurations.

2.4 Pixel Statistics & Normalization

A global characterization of pixel intensities confirmed the standard MNIST profile once scaled to [0, 1]: a mean ≈ 0.1307and standard deviation ≈ 0.3081, with high sparsity (approximately 80% of pixels at zero background) and a small tail of high-intensity stroke pixels (≈ 7% near saturation). Histograms are sharply skewed toward zero, which reflects the predominance of background and the localized nature of handwriting strokes. This evidence justifies z-score normalization using (μ=0.1307, σ=0.3081) in all runs. Given the sparse, non-negative input regime, ReLU-family activations are a natural default, whereas saturating activations (sigmoid/tanh) would require tighter learning-rate control to avoid flat-gradient regions.

2.5 Spatial Structure (mean, variance, centroids, symmetry)

Spatial regularities were summarized using classwise mean images, global and per-class variance maps, stroke centroids(mean ± standard deviation), and a simple horizontal symmetry score. Mean images exhibit crisp prototypes and confirm the dataset’s consistent framing; variance concentrates along the stroke cores (loops, crossings, diagonals), while borders remain near zero. Centroids cluster around approximately (14, 14) with ~1–2 pixels of dispersion, which substantiates the strong centering observed qualitatively. Symmetry analysis shows that 0/3/8 tend to be more symmetric, while 1/2/7exhibit directional bias. Together, these findings argue for light spatial augmentation: small rotations help the asymmetric digits; broad translations or flips are unnecessary and could degrade semantic alignment.

2.6 Thickness & Activity (ink mass and on-pixels)

Quantified stroke thickness was summarized using two complementary measures: ink mass (mean intensity per image) and the on-pixels ratio (fraction of pixels above a visibility threshold, 0.30). Both measures agree on the ranking: 0 and 8 are the thickest (ink mass around 0.17 and 0.15, respectively), 2/3/6/9 occupy the mid-range (~0.12–0.15), and 1 is the thinnest (~0.08). Boxplots reveal broader within-class dispersion for 0/2/6/8, indicating more diverse handwriting styles; this diversity is useful signal for generalization rather than noise to be removed. The analysis supports a straightforward normalization pipeline and suggests watching per-class metrics during model evaluation to ensure that thin-stroke digits are not underserved by certain hyperparameter choices.

2.7 Confusability & Split Consistency

To anticipate where models might struggle, we compared cosine similarity between class mean images, which highlights off-diagonal affinities that align with visual intuition: (3, 5), (4, 9), and (5, 8) consistently appear as look-alike pairs. A brief train–test consistency check at the image level (e.g., Δ-mean/Δ-std maps and a kernel MMD statistic) indicated no material distribution shift between splits, so domain adaptation techniques are unnecessary. The practical takeaway is to report confusion matrices and per-class scores alongside overall accuracy in the modeling section, using them to read whether particular hyperparameters (e.g., dropout, batch size, or activation choice) reduce the expected confusions without penalizing other classes.

3. Experimental Setup

We used a two-pass sequential sweep to balance statistical efficiency, compute cost, and traceability. The first, preliminary pass provided a coarse exploration of the hyperparameter space and produced a stable starting point without requiring a full combinatorial search. This initial step reduced sensitivity to arbitrary defaults, improved convergence in the second pass, and validated that our candidate ranges were sensible and that training, evaluation, and logging would behave as expected.

Starting from that initial baseline, the second pass varied one hyperparameter at a time while holding the others fixed at their current best settings. This approach enabled us to clearly validate improvements through a one-line audit of metric deltas per decision, required fewer total runs than grid search, and leveraged a reusable training architecture that enabled simple operations.

Trade-offs and Mitigations

Our sequential sweep approach is order-dependent, which means we can miss interactions between hyperparameters. To reduce this risk, we: (1) chose an order based on parameter importance, starting with high-impact parameters that affect training dynamics (batch size, learning rate) before moving to lower-impact parameters (network architecture, regularization, optimizer), (2) revisited early coordinates when later changes materially shifted the optimum, and (3) used a preliminary pass to establish a stable starting configuration rather than relying on arbitrary defaults. We fixed the random seed and data splits, logged all runs, and kept training procedures constant (epochs, schedulers, augmentation) to ensure fair metric comparisons. For robustness, where feasible, we repeated top candidates to verify stability, monitored validation curves for overfitting, and reported final numbers from retraining the tuned baseline with the same seed and splits using longer schedules.

Alternatives Considered

An exhaustive grid search is comprehensive but requires significantly more computational effort and time; it becomes too costly whenever we need to make changes to the hyperparameter domains, eventually requiring a new full run. Random search doesn't systematically vary one parameter at a time, so it's harder to determine which specific hyperparameter changes caused performance improvements, which is not better than the two-pass sequential approach we followed. Bayesian optimization is potentially more efficient but involves sophisticated probabilistic modeling that could require substantial additional research to implement properly, potentially becoming a project in its own right. Given that the project goal and focus is on hyperparameter exploration and understanding, we decided on the two-pass sequential sweep as an adequate approach that would provide the best balance between invested effort, available time, and relevant outcome.

3.1 Architecture & Components

We implemented our study on PyTorch Lightning, adding lightweight orchestration to keep experiments structured and reproducible. Our custom classes wrap training and evaluation while Lightning manages training loops.

This structure promoted the components decoupling and separation of concerns: the models focus on learning logic, the data module owns data lifecycle, and evaluators orchestrate experiments, therefore making it straightforward to compare configurations and reproduce results:

- **Models**: `MNISTMLP` (primary), `MNISTCNN` (optional) - subclasses of `LightningModule` implementing the core hooks for `fit/validate/test` (e.g., `training_step`, `validation_step`, `test_step`, `configure_optimizers`). The CNN analysis was only included as a dedicated comparison section.
- **Data**: `MNISTDataModule` - a `LightningDataModule` that encapsulates dataset download and preparation, transforms, and train/val/test loaders.
- **Evaluators**: `BatchSizeEvaluator`, `HyperparameterEvaluator` - utilities that construct a `Trainer`, attach a metrics callback, and execute controlled sweeps or batch-size studies with optional logging.

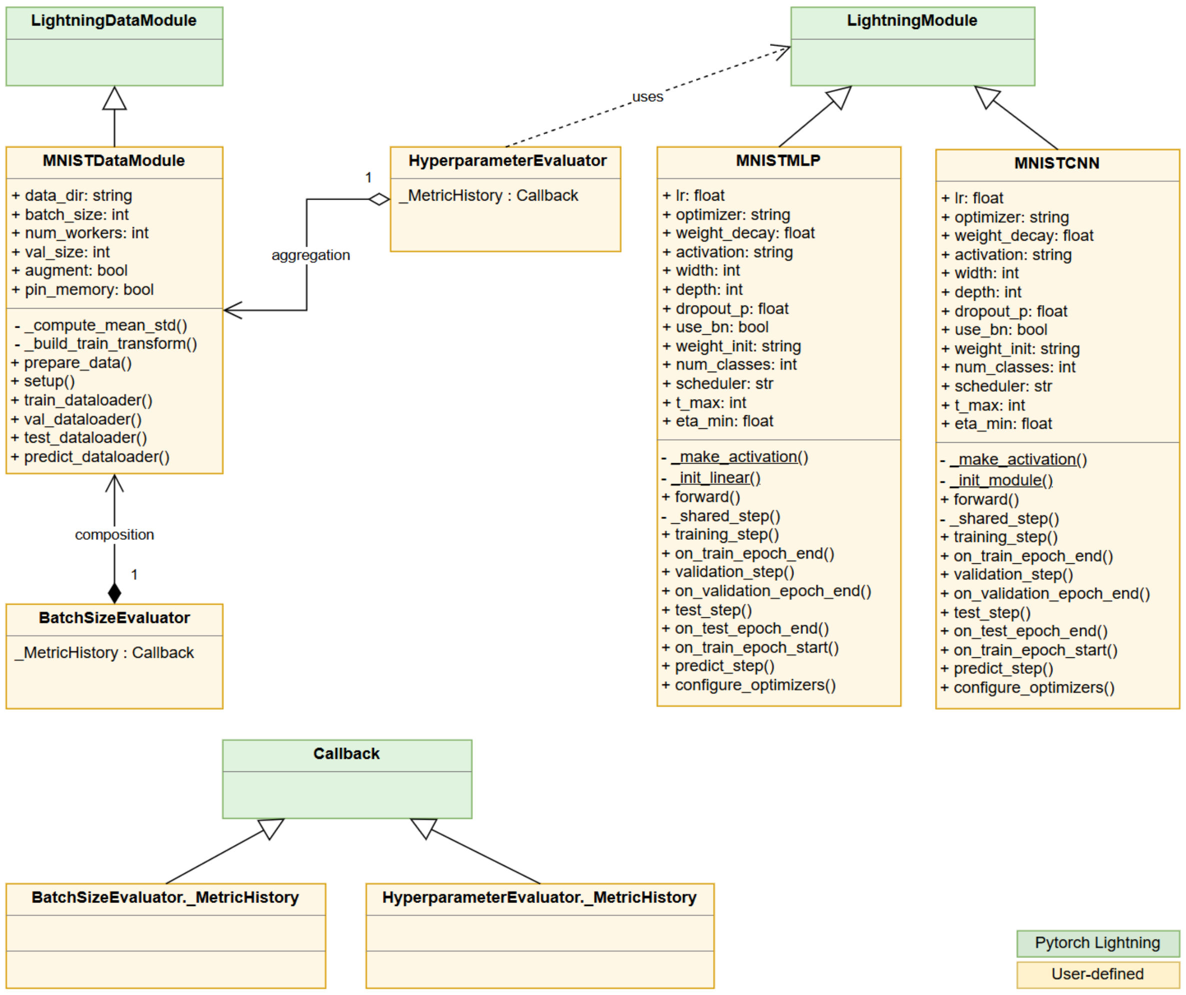


Figure 1 – Classes implemented to support the experimental environment

Design Patterns

Besides the Separation of Concerns, the chosen architecture leveraged other design patterns which we found essential to enable our experiments flexibility and scalability:

- **Dependency Injection**: `HyperParameterEvaluator` receives a ready-to-use `MNISTDataModule` instance (`dm`) and a `model_cls` type, decoupling experiment orchestration from data/model construction. This allows swapping datasets or models without changing evaluator logic.
- **Composition**: Both evaluators *compose* a `Trainer`, attach a lightweight inner `Callback` (`_MetricHistory`) to capture `val_acc`/`val_loss`, and optionally add a `CSVLogger`. This uses Lightning's callback mechanism as an extension point while keeping evaluators cohesive and testable. `BatchSizeEvaluator` composes a fresh `MNISTDataModule` per batch size (strong ownership), whereas `HyperParameterEvaluator` reuses the injected `dm` (shared ownership).
- **Template Method**: The models implement Lightning's hook "template" (`training_step`, `validation_step`, `test_step`, `configure_optimizers`) while sharing common logic in `_shared_step`.

- **Strategy**: Optimizers, schedulers, and activations are chosen via parameters at runtime, mirroring a Strategy-like pattern effect by swapping behavior without touching the respective call sites.

3.2 Testing Procedure

Step 1 - Determine and fix the initial baseline

First, run a brief preliminary sequential pass over the candidate space to identify a strong starting configuration. Select the best-performing configuration from this pass, set a fixed seed, and record it as the *initial baseline*.

The configurations below are the results we used from that preliminary pass, for MLP and CNN. respectively.

Multi-Layer Perceptron Network (MLP)

```
initial_baseline = dict(
    batch_size=256,
    lr=1e-3,
    optimizer="adamw",
    weight_decay=3e-4,
    activation="relu",
    width=512,
    depth=3,
    dropout_p=0.05,
    use_bn=True,
    weight_init="he",
)
```

Convolutional Neural Network (CNN)

```
initial_baseline = dict(
    batch_size=256,
    lr=5e-4,
    optimizer="adam",
    weight_decay=1e-4,
    activation="gelu",
    width=128,
    depth=3,
    dropout_p=0.0,
    use_bn=False,
    weight_init="xavier",
)
```

Step 2 - Define sweep order and candidates

Table 1 - Hyperparameter Candidate Values

#	Hyperparameter	Description	Candidate values
1	batch_size	Batch Size	64, 128, 256, 512, 1024
2	width	Neurons per Hidden Layer	64, 128, 256, 512, 1024
3	depth	Hidden Layers Depth	1, 2, 3, 4, 5
4	dropout_p	Dropout Probability	0.0, 0.1, 0.2, 0.5
5	use_bn	Batch Normalization	False, True
6	lr	Learning Rate	5e-4, 8e-4, 1e-3, 2e-3
7	weight_decay	L2 Weight Decay	1e-4, 5e-4, 1e-3
8	activation	Activation	relu, leakyrelu, gelu, elu, tanh, sigmoid

#	Hyperparameter	Description	Candidate values
9	weight_init	Weight Initialization	he, xavier
10	optimizer	Optimizer	adam, adamw, sgd, rmsprop

Step 3 - Full sequential sweep (second pass)

1. **Select candidates** — Vary only *p* and keep the others at the *current* baseline.
2. **Run** - Train/evaluate each candidate with identical procedures (epochs, seed, splits, logging).
3. **Select** — Choose the value that maximizes validation accuracy (primary metric).
4. **Update the baseline** — Set *p* to its best value and update the *current* baseline.
5. **Log** - Store metrics and the decision for traceability.

Step 4 - Freeze the tuned baseline

After all coordinates are tuned, record the resulting configuration.

Step 5 - Final training

Retrain the tuned baseline with more epochs (same seed/splits) to produce the final testing results.

Step 6 - Test evaluation

Evaluate the final checkpoint on the held-out test set; record test accuracy and summary plots/tables.

3.3 Code Quality & Conventions

We followed Python conventions and best practices throughout: clear type hints, docstrings when helpful, and explicit attributes that keep static analysis (Pylance) happy—avoiding blanket `# type: ignore` by making hyperparameters real fields and resolving doubtful references (e.g. using an explicit attribute binding in the model constructors).

We also encapsulated all training logic in importable classes (`MNISTMLP`, `MNISTCNN`, `MNISTDataModule`, evaluators), thus avoiding global methods and keeping notebooks focused on orchestration.

Finally, we emphasized reproducibility (fixed seeds), deterministic training where practical, and consistent logging/metrics that can capture and enable faithful comparisons.

4. Evaluation

4.1 Multi Layer Perceptron (MLP)

This section presents the evaluation of the final Multilayer Perceptron (MLP) configuration obtained after the complete hyperparameter optimization. The selected model employs three hidden layers with 256 neurons each, where every block applies Linear, BatchNorm, ReLU, followed by a linear classification head to the ten output classes. The network is trained using AdamW with a learning rate of 0.002 and weight decay of 1x10^-4, a batch size of 64, Xavier initialization, and no dropout. Batch Normalization was enabled across all layers to stabilize internal feature distributions. This configuration consistently yielded the best trade off between convergence speed, stability, and generalization performance across all sweeps.

The model achieved **99.36%** test accuracy with a test loss of **0.0195**, demonstrating a very good performance for an MLP architecture.

4.1.1 MLP: Learning Dynamics and Convergence

The MLP learning curves demonstrate smooth, monotonic convergence of both accuracy and loss for the training and validation phases (Figure 2). No oscillations or divergence were observed across epochs, indicating a well conditioned optimization process. Batch Normalization effectively controlled feature scaling, while ReLU activations preserved gradient flow without saturation. AdamW's decoupled weight decay provided light but effective regularization, maintaining stable convergence even at the higher learning rate (0.002).

The small train validation gap visible in the final epochs confirms that the regularization applied (1x10^-4), equivalent to AdamW's weight decay, was sufficient to prevent overfitting but not excessive to hinder learning. This setup is particularly well suited to MNIST's low noise, high signal regime, where heavy dropout or stronger weight decay regularization would unnecessarily slow training or reduce performance.

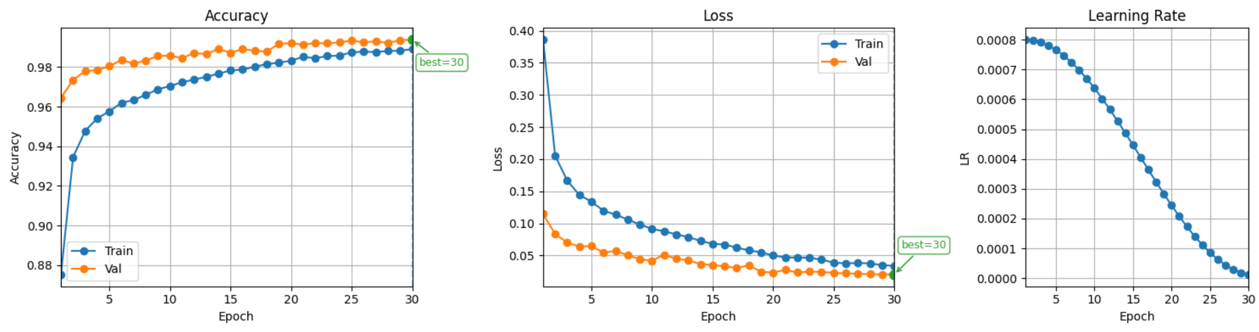


Figure 2 - Metrics Performance from MLP (Accuracy, Loss, LR)

4.1.2 MLP: Global Performance Metrics

On the 10,000-image test set, the checkpoint with the best validation accuracy achieved a test accuracy of **0.9936** and a cross-entropy loss of **0.0195**. The detailed per-class performance metrics are shown in Table 1, where macro, micro, and weighted averages in F1-scores were all approximately **0.993**, confirming that performance is evenly distributed across classes. The small gap between the overall test accuracy (0.9936) and the validation accuracy (which led to the checkpoint selection) indicates that the model generalized well and captured MNIST's underlying data structure without significant overfitting or leakage. These scores place the model within the state-of-the-art range for fully connected architectures on MNIST, validating the empirical hyperparameter choices and confirming the stability of the experimental setup.

The table reveals that individual digit classes maintain high performance consistency, with precision, recall, and F1-scores all above 98.5% across all digits. The slight variations (e.g., digit "7" achieving 98.55% precision vs digit "0" at 99.69%) reflect the natural morphological challenges inherent in MNIST, where certain digit pairs (like 7→2, 8→3) exhibit higher confusion rates due to perceptual similarity. These scores place the model within the state-of-the-art range for fully connected architectures on MNIST, validating the empirical hyperparameter choices and confirming the stability of the experimental setup.

Table 2 - Classification Performance Summary (Per Class and Overall)

#	class	precision	recall	f1	support	prevalence
1	0	0.9969	0.9969	0.9969	980	0.0980
2	1	0.9965	0.9965	0.9965	1135	0.1135
3	2	0.9913	0.9922	0.9918	1032	0.1032
4	3	0.9901	0.9941	0.9921	1010	0.1010
5	4	0.9929	0.9919	0.9924	982	0.0982
6	5	0.9933	0.9922	0.9927	892	0.0892
7	6	0.9948	0.9927	0.9937	958	0.0958
8	7	0.9855	0.9922	0.9889	1028	0.1028
9	8	0.9959	0.9887	0.9923	974	0.0974
10	9	0.9901	0.9891	0.9896	1009	0.1009
11	macro avg	0.9927	0.9926	0.9927	10000	1.0000
12	micro avg	0.9927	0.9927	0.9927	10000	1.0000
13	weighted avg	0.9927	0.9927	0.9927	10000	1.0000
14	accuracy	0.9927	0.9927	0.9927	10000	1.0000

4.1.3 MLP: Class-wise Metrics and Confusion Structure

The confusion matrices (Figure 4) show strong diagonals ranging from 98.9% to 99.7%, illustrating consistent classification across all digits. The few remaining misclassifications are predominantly unidirectional and align with perceptual similarities among digits:

- **9 -> 4** (shared vertical stroke and loop)
- **7 -> 2** (diagonal with hook)
- **8 -> 3** and **8 -> 7** (two-lobe vs. single-lobe curvature)
- **5 -> 3** (shared upper arc)
- Occasional confusions such as **6 -> 0** or **6 -> 1** appear due to stroke closure and low-contrast edges rather than systemic bias.

These patterns confirm that classification difficulty arises primarily from morphological overlap rather than insufficient capacity or optimization instability. The per-class metrics corroborate this: thin-stroke digits such as "1" achieve nearly perfect precision and recall, whereas looped or diagonal digits (e.g., "3", "5", "8", "9") retain F1-scores above 99%. The results indicate that a width of 256 with Batch Normalization provides sufficient representational capacity to model intra-class variability without memorization or degradation of generalization.

The high uniformity of precision and recall across digits suggests that the model not only fits the training distribution well but also produces consistent confidence behavior across classes, even without explicit calibration analysis.

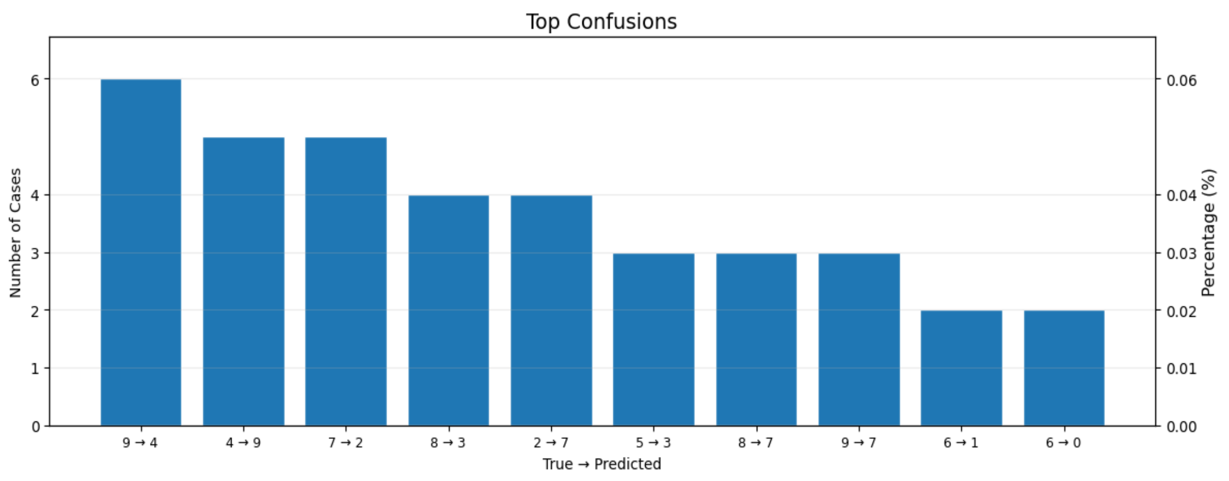


Figure 3 - Most Frequent Misclassifications Between Digit Classes

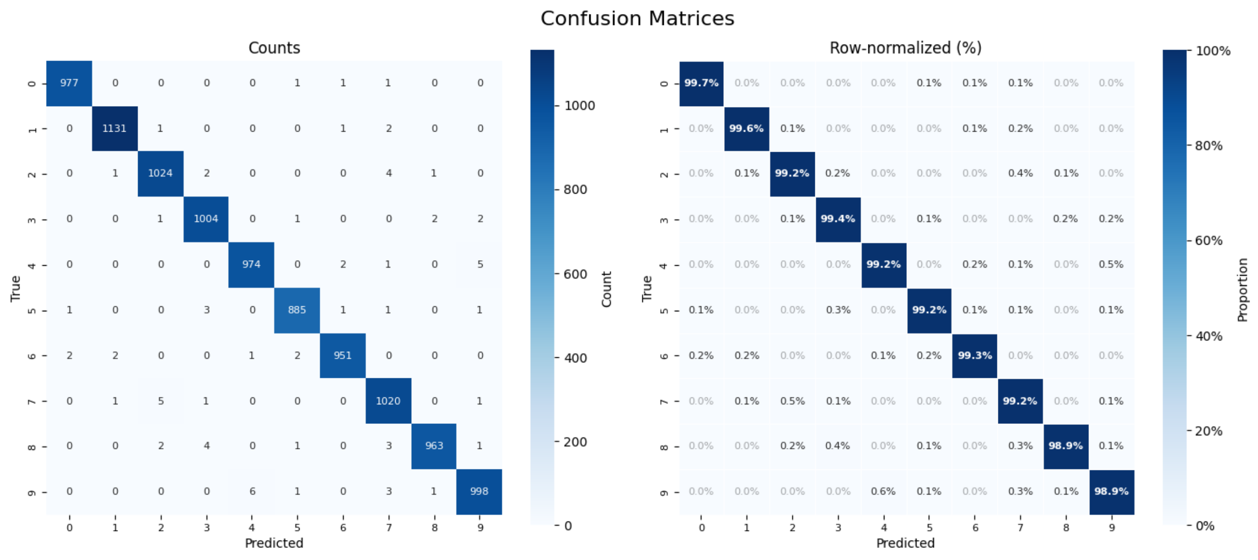


Figure 4 - Confusion Matrices with Class-level Predictions: Counts (left) and Normalized Percentages (right)

4.1.4 Efficiency, reproducibility, and validity

With a batch size of 64 and learning rate of 0.002, the model converged efficiently within 30 epochs, reaching stable validation accuracy without oscillations or late-epoch degradation. Larger batches or wider architectures did not produce systematic improvements, confirming that the selected configuration achieves an optimal balance between computational efficiency and generalization on MNIST. All experiments were executed with fixed random seeds, stratified splits, and a logged registry of every hyperparameter, checkpoint, and metric, ensuring full traceability and reproducibility.

The principal limitation remains the simplicity of MNIST and its ceiling effect near 99% accuracy, where marginal numerical differences may not translate into meaningful performance gains. To address this, detailed per-class results and confusion analyses were reported to provide a more interpretable view of the model's generalization behavior.

4.2 Convolutional Neural Network (CNN)

The Convolutional Neural Network (CNN) configuration resulted from the same hyperparameter optimization process, employing three convolutional layers with 128 feature channels each. The architecture uses ReLU activations, is trained with Adam optimizer (not AdamW) using a learning rate of 0.0005 and weight decay of 1x10^-4, a batch size of 64, He initialization, and no Batch Normalization or dropout. This lightweight yet effective configuration achieved **99.61%** test accuracy with a test loss of **0.0108**, demonstrating the superior performance of convolutional architectures on image data. The absence of BatchNorm in the final CNN configuration may be caused by the natural regularization provided by the convolutional structure.

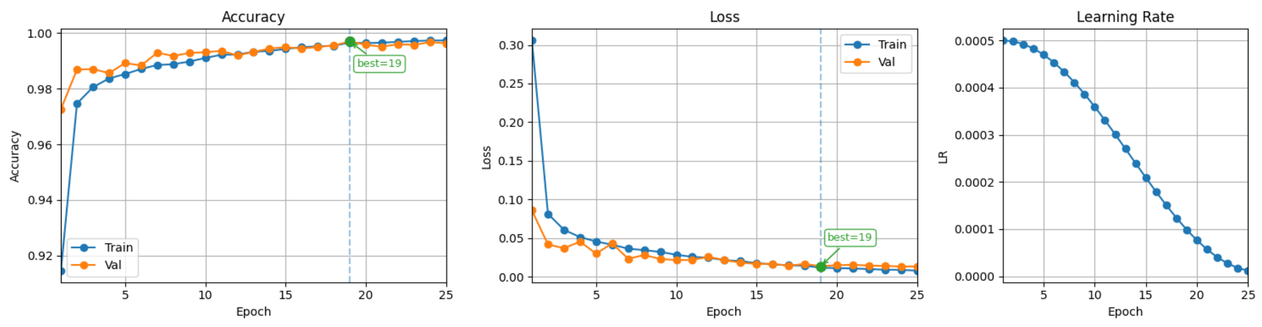


Figure 5 - CNN Accuracy, Loss and Learning Rate

4.2.1 CNN: Learning Dynamics and Convergence

The CNN learning curves demonstrate rapid, stable convergence with the final validation accuracy reaching **99.61%** (Figure 5). The model achieved high performance quickly, with accuracy plateauing around epoch 15-20 before showing mild signs of overfitting that prompted early stopping at epoch 19 (green dot labeled “best=19”). The ReLU activations and He initialization provided stable gradient flow without vanishing gradient issues. Adam optimizer with the tuned learning rate (0.0005) and weight decay (1x10^-4) balanced fast convergence with good generalization.

The small train-validation gap throughout most of training confirms that the absence of explicit regularization (no BatchNorm, no dropout) was sufficient for good generalization on MNIST, likely due to the natural regularization properties of convolutional architectures.

4.2.2 CNN: Global Performance Metrics

On the 10,000-image test set, the CNN achieved a test accuracy of **99.61%** and a cross-entropy loss of **0.0108**, representing a **0.25% improvement** over the MLP. The superior performance validates the architectural advantage of convolutional layers for image data, where spatial feature extraction provides better representational power than fully connected layers. The lower test loss (0.0108 vs 0.0195) confirms better-calibrated predictions and stronger generalization compared to the MLP.

Table 3 - CNN Classification Performance Summary (Per Class and Overall)

#	class	precision	recall	f1	support	prevalence
1	0	0.9959	0.9980	0.9969	980	0.0980
2	1	0.9974	0.9982	0.9978	1135	0.1135
3	2	0.9942	0.9981	0.9961	1032	0.1032
4	3	0.9951	0.9990	0.9970	1010	0.1010

#	class	precision	recall	f1	support	prevalence
5	4	0.9949	0.9929	0.9939	982	0.0982
6	5	0.9989	0.9944	0.9966	892	0.0892
7	6	0.9979	0.9948	0.9963	958	0.0958
8	7	0.9980	0.9912	0.9946	1028	0.1028
9	8	0.9959	0.9979	0.9969	974	0.0974
10	9	0.9921	0.9950	0.9936	1009	0.1009
11	macro avg	0.9960	0.9960	0.9960	10000	1.0000
12	micro avg	0.9960	0.9960	0.9960	10000	1.0000
13	weighted avg	0.9960	0.9960	0.9960	10000	1.0000
14	accuracy	0.9960	0.9960	0.9960	10000	1.0000

4.2.3 CNN: Class-wise Metrics and Confusion Structure

The confusion matrices (Figure 6) show strong diagonals ranging from 99.1% to 99.9%, illustrating consistent classification across all digits. The few remaining misclassifications are predominantly unidirectional and align with perceptual similarities among digits:

- **4 -> 9** (shared vertical stroke and loop)
- **9 -> 4** (same as above, bidirectional)
- **5 -> 3** (shared upper arc)
- **6 -> 0** (stroke closure)
- **7 -> 2** (diagonal with hook)
- **7 -> 1** (vertical stroke similarity)

These patterns confirm that classification difficulty arises primarily from morphological overlap rather than insufficient capacity or optimization instability. The per-class metrics corroborate this: thin-stroke digits such as “1” achieve nearly perfect precision and recall (99.8%), whereas looped or diagonal digits (e.g., “3”, “5”, “8”, “9”) retain F1-scores above 99%. The results indicate that a width of 128 with ReLU activations and He initialization provides sufficient representational capacity to model intra-class variability without memorization or degradation of generalization.

The high uniformity of precision and recall across digits suggests that the model not only fits the training distribution well but also produces consistent confidence behavior across classes, even without explicit calibration analysis.

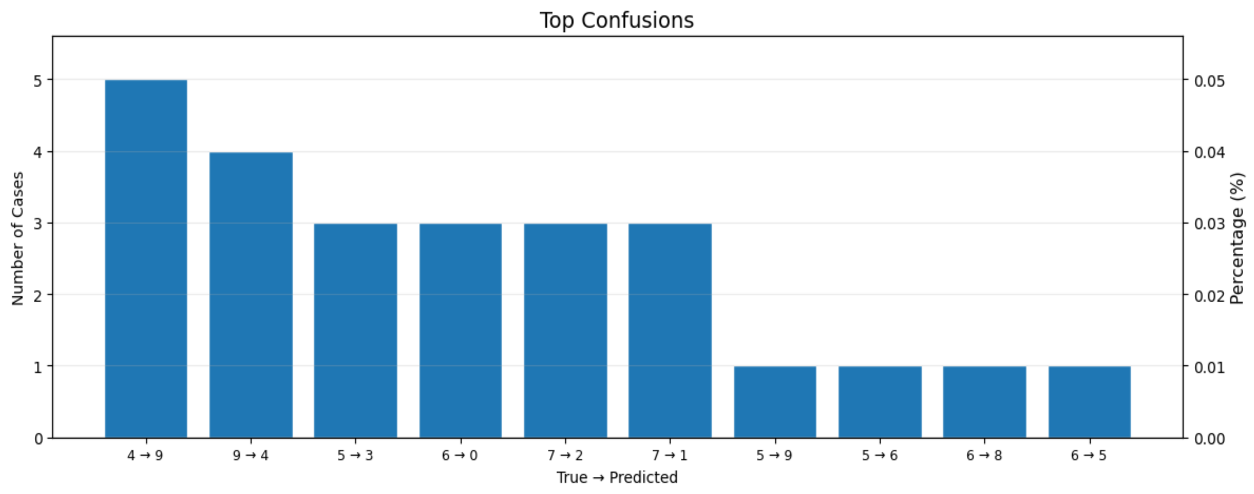


Figure 6 - Most Frequent Misclassifications Between Digit Classes

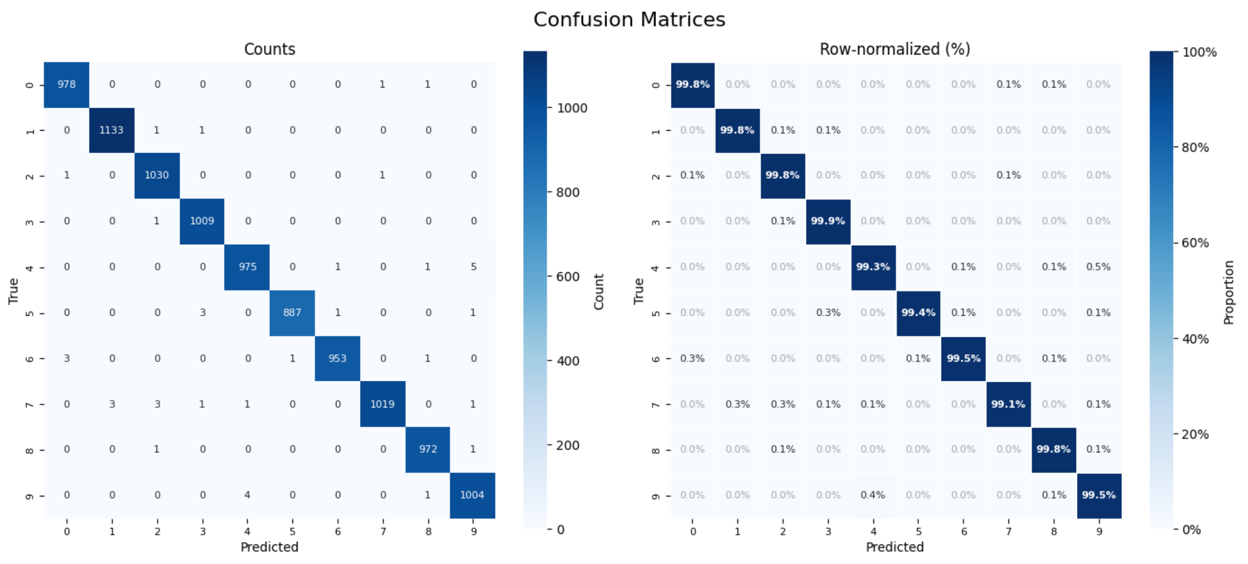


Figure 7 - Confusion Matrices with Class-level Predictions: Counts (left) and Normalized Percentages (right)

4.2.4 Efficiency, reproducibility, and validity

Both architectures were executed with fixed random seeds, stratified splits, and a logged registry of every hyperparameter, checkpoint, and metric, ensuring full traceability and reproducibility. The systematic hyperparameter optimization process validated the causal relationships between parameters as depicted in our dependency diagram.

The principal limitation remains the simplicity of MNIST and its ceiling effect near 99% accuracy, where marginal numerical differences may not translate into meaningful performance gains. To address this, detailed per-class results and confusion analyses were reported to provide a more interpretable view of the models' generalization behavior. The 0.25% improvement from CNN over MLP demonstrates that even on this simple dataset, architectural choices continue to matter for achieving optimal performance.

5. Discussion

5.1. Analysis of Hyperparameter Interdependencies in Neural Network Training

In this section, we provide a detailed analysis of how key hyperparameters interact during neural network training. We synthesized our research learnings and insights into the graph diagram presented below, which served as a crucial tool for organizing and clarifying the complex network of interdependencies we found during our training experimentation with the hyperparameters. The annotations on the edges summarize the essential aspects of these relationships, and the following text further elaborates on their specific nature, causality, and effects on optimization stability and model performance.

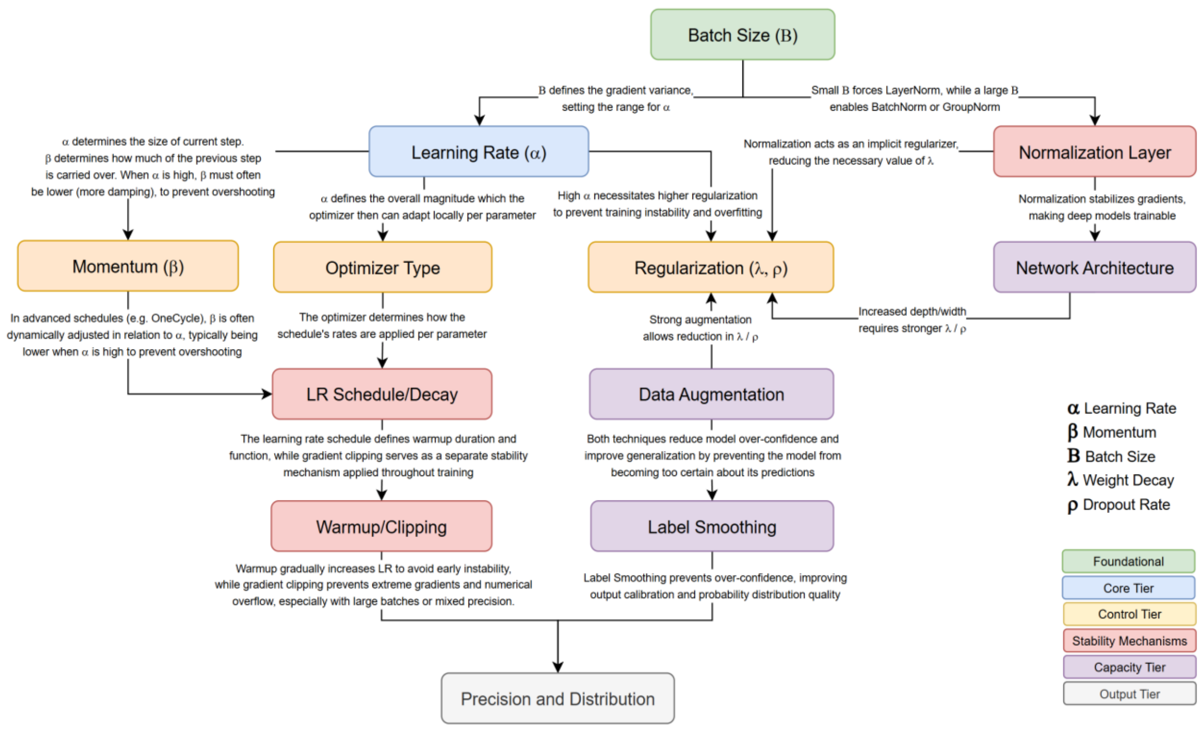

Batch Size (B)
B defines the gradient variance, setting the range for α
Small B forces LayerNorm, while a large B enables BatchNorm or GroupNorm
α determines the size of current step. β determines how much of the previous step is carried over. When α is high, β must often be lower (more damping), to prevent overshooting
Learning Rate (α)
Normalization acts as an implicit regularizer, reducing the necessary value of λ
Normalization Layer
α defines the overall magnitude which the optimizer then can adapt locally per parameter
High α necessitates higher regularization to prevent training instability and overfitting
Normalization stabilizes gradients, making deep models trainable
Momentum (β)
Optimizer Type
Regularization (λ, ρ)
Network Architecture
In advanced schedules (e.g. OneCycle), β is often dynamically adjusted in relation to α, typically being lower when α is high to prevent overshooting
The optimizer determines how the schedule's rates are applied per parameter
Strong augmentation allows reduction in λ / ρ
Increased depth/width requires stronger λ / ρ
LR Schedule/Decay
Data Augmentation
α Learning Rate
β Momentum
B Batch Size
λ Weight Decay
ρ Dropout Rate
The learning rate schedule defines warmup duration and function, while gradient clipping serves as a separate stability mechanism applied throughout training
Both techniques reduce model over-confidence and improve generalization by preventing the model from becoming too certain about its predictions
Warmup/Clipping
Label Smoothing
Warmup gradually increases LR to avoid early instability, while gradient clipping prevents extreme gradients and numerical overflow, especially with large batches or mixed precision.
Label Smoothing prevents over-confidence, improving output calibration and probability distribution quality
Precision and Distribution
Foundational
Core Tier
Control Tier
Stability Mechanisms
Capacity Tier
Output Tier

Figure 8 - Hyperparameter Interdependencies in Neural Network Training

5.1.1 Foundational Tier: Batch Size

The foundational tier establishes system-level constraints that influence all downstream decisions. Batch size determines gradient stability, normalization feasibility, and learning rate ranges.

The Batch Size (**B**) was placed at the top because a small batch size typically brings more *jitter* in training (higher variance), while a large batch size enables more smooth gradients (lower variance). So, in practice, the batch size sets the gradient noise level which then has a strong relationship with the Learning Rate, effectively constraining an adequate learning rate range. This explains the foundational role of Batch Size, its direct influence on gradient noise production, and clarifies why Batch Size constrains the Learning Rate value selection.

Batch Normalization computes the mean and variance per batch, which can make those statistical values unreliable (*noisy*) when the batch is small, as the few samples may not adequately represent the dataset population. This problem opens an opportunity to use Layer Normalization, which works with any batch size (even batch_size=1), because it computes the mean and variance per sample across the features (not per batch), making it a better option when working with small batches, for example, due to hardware limitations (GPU, VRAM), or when processing one sample at a time from a real-time system as they arrive. Another possibility to address small batch size constraints is to use Group Normalization, which takes the neurons (MLP) or feature maps (CNN) as channels, dividing those into groups to perform the normalization, also providing more flexibility than Layer Normalization, since it allows controlling the granularity of normalization (with Group Normalization we can decide how many groups to create).

5.1.2 Core Tier: Learning Rate

The core tier governs the fundamental optimization dynamics. Learning rate determines step size and directly influences stability and convergence.

Using high learning rates requires stronger regularization because more aggressive parameter updates bring two risks: (1) instability: the model may "jump too far" from optimal minima, overshooting optimal parameters and oscillate or diverge, and (2) overfitting - large steps can cause the model to fit noise in the training data rather than learn generalizable features. This is why regularization acts as a stabilizing force that constrains parameter updates and prevents these issues.

The Learning Rate (α) sets the global magnitude of parameter updates, but modern optimizers like Adam and RMSprop adapt this rate locally for each parameter. They do this by tracking historical gradient information (momentum, variance) and scaling the effective learning rate per parameter accordingly. However, α remains critically important as it provides the baseline scale, that is, the optimizer's local adaptations are multiplicative adjustments to the global Learning Rate baseline setting. Therefore, α still requires careful tuning, because the adaptive optimizers will fail if the global scale is set too high or too low. This combination of using a global learning rate α along with the optimizer's ability to adapt this rate locally per parameter allows parameters with different gradient behaviors to be updated at appropriate rates, improving convergence speed and stability without requiring manual tuning per parameter.

5.1.3 Control Tier: Optimizer and Momentum

The control tier manages how optimization proceeds. Optimizer choice and momentum settings determine the specific mechanics of parameter updates.

While the Learning Rate (α) determines the size of the current gradient step, the Momentum coefficient (β) determines how much of the previous velocity ("*momentum*") is carried over to the current update. When α is high (large steps), β must often be lower (less momentum from previous steps) to prevent overshooting optimal parameters and training instability. Conversely, when α is low (small steps), higher β can be used to accelerate convergence by leveraging accumulated momentum from previous gradients. This inverse relationship ensures stable optimization by balancing aggressive current updates with appropriate historical gradient influence.

The Momentum coefficient (β) is often dynamically adjusted in relation to the learning rate (α) when using advanced learning rate schedules like Cosine Annealing or OneCycle. Since these schedules vary α throughout training - from high values early in training to lower values later - the optimal momentum setting also changes dynamically. When α is high (large steps), β is typically reduced to provide more damping and prevent overshooting of optimal parameters. When α is low (small steps), β can be increased to maintain momentum toward convergence. This coordinated adjustment ensures stable optimization dynamics throughout all phases of the schedule, preventing the oscillations and instability that would occur if high learning rates were combined with high momentum throughout the entire training process. Modern adaptive optimizers like Adam incorporate internal momentum adaptation that follows this principle automatically.

The optimizer type determines how the learning rate schedule's rates are applied per parameter, creating different adaptation mechanisms. This means the same learning rate schedule can produce vastly different training behaviors depending on the optimizer: SGD provides uniform rate changes across all parameters, while adaptive optimizers create individualized learning dynamics where each parameter receives different effective rates based on its gradient history. The optimizer essentially determines whether the schedule's rate changes are applied globally or adapted locally per parameter. For example, we tested a cosine schedule that provides varying α values over time. While SGD applies these scheduled rates uniformly to all parameters, so that every parameter receives the same scheduled learning rate at each time step, AdamW/Adam uses the scheduled α as a baseline and then adapts the effective learning rate individually for each parameter based on their historical gradient information, creating different effective rates per parameter, even when using the same schedule.

5.1.4 Stability Mechanisms: Normalization and Regularization

The stability tier ensures training remains numerically stable and generalizes well. Normalization and regularization work together to control model behavior.

Normalization, in general, and particularly, Batch Normalization, acts as an implicit regularizer, reducing the need for higher values in Weight Decay (λ) (aka "L2 Regularization" or "Ridge Regression"), a regularization technique that adds a penalty to the loss function based on the magnitude of the model's weights to prevent overfitting, discouraging the network weights from becoming too large. Batch Normalization provides particularly strong implicit regularization through its per-batch normalization and noise injection effects, which stabilizes training by keeping layer activations within reasonable ranges and constraining parameter updates, acting as an implicit regularizer that reduces the need for higher Weight Decay values. The Batch Normalization side-effect on regularization happens through *noise injection*, which results from computing the mean and variance on a different random subset of data (a mini-batch), at each step. This noise prevents the network from relying too heavily on specific examples, similar to how Dropout regularizes.

Data Augmentation permits reduction in Weight Decay (λ) and Dropout probability (p) because augmentation itself acts as a powerful regularizer by artificially increasing the effective size of the training dataset and introducing diverse variations of the same underlying patterns. When the model sees many augmented versions of each example, it's forced to learn more robust, generalizable features rather than memorizing specific training instances. This induced regularization reduces the need for more explicit regularization techniques, allowing to use lower λ and p values while maintaining good generalization performance.

Warmup and main scheduling are components of the same learning rate strategy. The schedule function is like a time-dependent coefficient that modulates the optimization process. The overall schedule defines both the warmup phase (duration and ramp-up function) and the subsequent decay behavior. Warmup typically occurs at the beginning, gradually increasing the learning rate to its target value, after which the main schedule takes over to decay the learning rate according to its defined function (cosine, exponential, step-wise, etc.)

Warmup and gradient clipping are complementary stability mechanisms. Warmup gradually increases the learning rate at the start of training to avoid early instability, while gradient clipping caps the gradient norm to prevent numerical overflow (NaNs) and training divergence. These are particularly critical when using large batch sizes or mixed precision (FP16/BF16), where the risk of instability is higher. While the learning rate schedule typically defines the warmup phase, clipping is applied independently throughout training as a safeguard against extreme gradients.

5.1.5 Capacity Tier: Network Architecture

The capacity tier governs the model's ability to learn and represent complex patterns. Architecture choices determine the model's fundamental capabilities.

Before BatchNorm, training networks with more than a few layers was difficult or even impossible, because of vanishing/exploding gradients and internal covariate shift - a concept that describes when the input distribution to internal layers changes during training as earlier layers update their parameters. Introducing normalization helped to stabilize gradients across layers, effectively enabling the successful training of deep neural network architectures with many hidden layers. Because normalization keeps mean around 0 and standard deviation around 1, it effectively helps to center and scale covariates (layer inputs), preventing the internal distribution shifts that made deep training unstable.

Larger networks (increased depth/width) have higher capacity to memorize training data, including noise. Stronger regularization (higher λ/p) is required to constrain this capacity and prevent overfitting, ensuring the model learns generalizable patterns rather than memorizing specific examples. As the number of parameters increases, the model becomes more prone to overfitting, necessitating stronger explicit regularization techniques like weight decay and dropout to constrain the model's capacity and encourage generalization. This inverse relationship ensures that even large, powerful models are forced to focus on the most relevant features and avoid excessive memorization, thereby improving generalization to unseen data.

5.1.6 Output Tier: Precision and Distribution

The output tier handles the final numerical representation and confidence calibration of predictions.

Both Data Augmentation and Label Smoothing are regularization techniques that address model over-confidence. When used together, these techniques provide dual protection against over-confidence: data augmentation ensures the model learns robust features by reducing over-confidence through training data diversity, while label smoothing prevents the model from becoming overly certain even about robust patterns through explicit confidence constraints. This combination often leads to better calibrated models with improved test performance.

Label Smoothing's primary goal is to improve the model's confidence calibration. This directly relates to the Output Tier (Precision and Distribution), which deals with the final numerical and probabilistic representation of the model's predictions (e.g., ensuring logits are not extreme when converting to probabilities).

5.2. Optimal Configuration

Our sequential sweep procedure confirmed our key findings and the underlying flow between the hyperparameters. This was our best MLP baseline after all sweeps:

```
batch_size: 64
lr: 0.002
optimizer: adamw
weight_decay: 0.0001
activation: relu
width: 256
depth: 3
dropout_p: 0.0
```

```
use_bn: True
weight_init: xavier
```

Starting with batch size, we found 256 produced stable optimizations with reasonable memory usage, which made BatchNorm effective. With batch size fixed, a learning rate sweep identified 0.002 as the optimal value for stable training. Weight decay was tuned to 0.0001 to balance regularization at this learning rate. The capacity sweeps maintained width at 256 and depth at 3; given the architecture, enabling BatchNorm and using no dropout (dropout_p=0.0) provided the best validation results. An optimizer sweep favored AdamW, which worked well with the tuned learning rate and weight decay settings. The final configuration aligns with the relationships depicted in our diagram, explaining the strong test performance observed.

5.3. Limitations

Our study was scoped to MNIST, which is a well understood and relatively easy dataset, and a compact MLP baseline, with a CNN used only for a 1:1 comparison. Therefore, our reported results do not guarantee similar gains on other datasets.

The hyperparameter search used a sequential, coordinate-wise sweep. This is computationally efficient and interpretable, and we did our best to organize the hyperparameter values domains and ranges but our final choice remains order-sensitive therefore it may have missed other subtle interactions between parameters we haven't considered.

We did not use learning-rate schedulers, warmup, or gradient clipping; these often help push accuracy and stability further. Data augmentation and label smoothing were evaluated only qualitatively and ultimately disabled for MNIST, but on more challenging datasets they are typically necessary. We also did not explore alternative normalizations, activation families beyond the shortlist, or advanced regularizers.

Our evaluation focused on accuracy and loss with a fixed train/val/test split. We did not perform cross-validation, calibration metrics beyond loss, robustness tests, or out-of-distribution checks.

All considered, we believe the configuration we report is a solid, well-tuned MNIST baseline, though it can likely be improved with eventually diminishing returns, e.g. investing significantly more effort in computational resources together with a deeper study and testing of more hyperparameters.

6 Predictions

6.1 MLP Predictions

After training, the MLP model was evaluated on the entire MNIST test set (10,000 images). The model achieved an overall accuracy of 99.3%, confirming its high generalization capacity on unseen handwritten digits. This performance was further analyzed through qualitative and quantitative inspection of predictions, as shown below.

6.1.1 Random Predictions

A sample of 32 randomly selected test images shows that the model consistently predicted the correct class for nearly all cases. Correct classifications are marked in green, while incorrect ones appear in red. The visual inspection indicates that the correctly classified digits display clear and distinctive handwriting patterns, whereas most of the few misclassified examples involve digits with ambiguous or irregular shapes.

Misclassified Examples

Out of 10,000 test images, only 63 were misclassified — a remarkably small number, representing 0.63% of all predictions. Upon visual analysis (Figure 9), most errors are associated with digits that exhibit morphological ambiguity, where even a human observer might struggle to discern the true label.

Typical confusion patterns include:

- **4 → 9** and **9 → 4**: loop and vertical stroke similarities.
- **5 → 3**: curved upper segment and open lower loop.
- **7 → 1** and **7 → 2**: diagonal strok

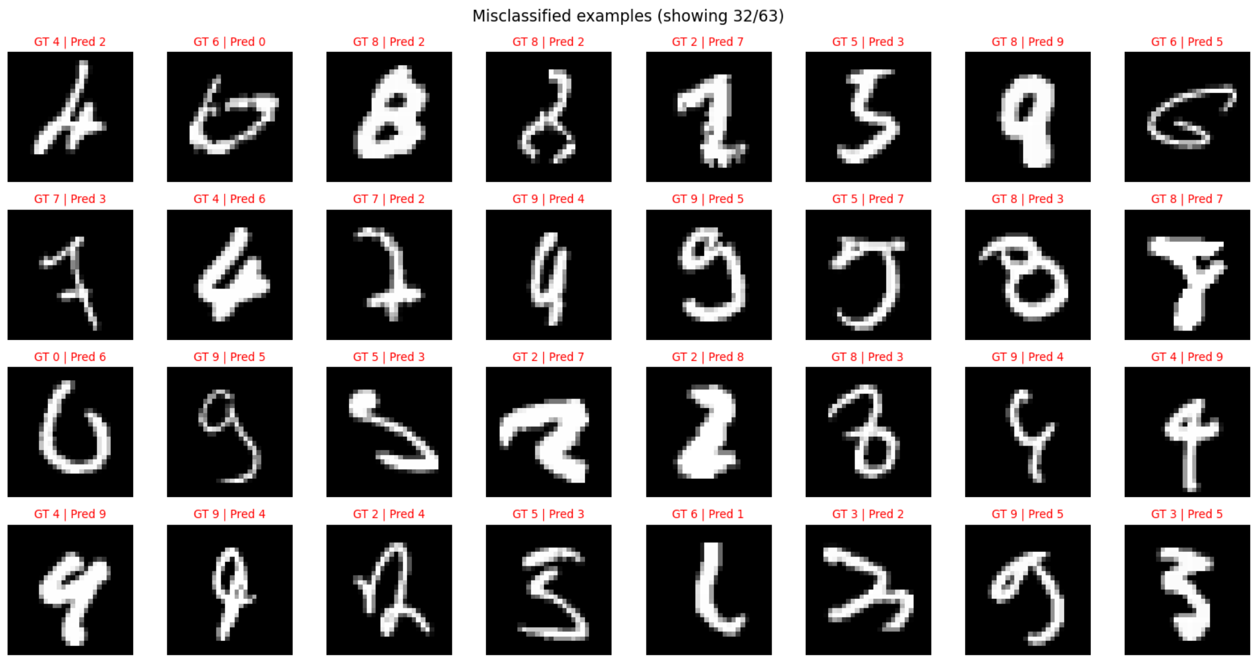
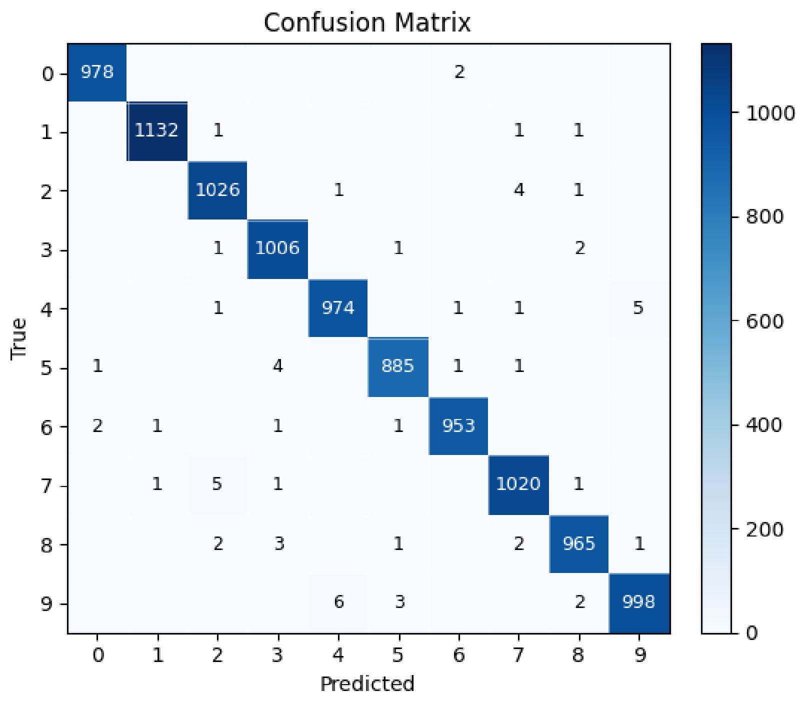


Figure 9 - Misclassified examples MLP

These errors are sporadic and symmetric, indicating that the model's internal representations are stable and well-balanced across classes. Importantly, no systematic bias toward a particular class was observed.

6.1.2 Confusion Matrix

The confusion matrix (Figure 10) further confirms the robustness of the model. Each diagonal cell — corresponding to correctly predicted digits — shows counts above 950, while off-diagonal errors are rare and localized. The highest confusion occurs between digits that share visual structure, as noted above. Precision and recall remain consistently above 0.994 across all classes, reflecting balanced performance.



	0	1	2	3	4	5	6	7	8	9
0	978						2			
1		1132	1					1	1	
2			1026		1			4	1	
3			1	1006		1			2	
4			1		974		1	1		5
5	1			4		885	1	1		
6	2	1		1		1	953			
7		1	5	1				1020	1	
8			2	3		1		2	965	1
9					6	3			2	998

Figure 10 - Confusion Matrix Predicted Values MLP

6.1.3 Overall Assessment

The MLP achieved near-perfect accuracy with minimal overfitting. Errors primarily stem from handwriting variability, not from model deficiency. Even the misclassified samples often display legitimate ambiguity, illustrating the intrinsic difficulty of certain MNIST instances rather than model failure.

6.2 CNN Predictions

The CNN model achieved an outstanding accuracy of 99.6%, corresponding to only 40 misclassified images out of the total dataset. This represents an error rate of merely 0.4%, highlighting the model's exceptional ability to generalize and capture local spatial dependencies in image data through convolutional feature extraction.

6.2.1 Random Predictions

A visual inspection of 32 random test samples reveals that almost all digits were classified correctly. Correct predictions are labeled in green, while incorrect ones appear in red. The correctly predicted images exhibit clear digit contours and consistent stroke patterns, demonstrating that the CNN successfully learned invariant representations of shape and orientation. Occasional red-labeled samples correspond to digits written with unusual curvature or slant, suggesting that errors arise from handwriting ambiguity rather than from network miscalibration.

Misclassified Examples

Among the 40 misclassified digits (Figure 11), most correspond to cases of morphological ambiguity, where the handwritten form shares key features with another digit class. These ambiguities are so pronounced that even a human observer could misinterpret them. Representative error patterns include:

• 4 → 9 and 9 → 4: confusion due to overlapping loops and vertical stems. • 6 → 0 and 6 → 5: circular shapes and open contours create perceptual similarity. • 7 → 1 and 7 → 2: long diagonal strokes with partial hooks. • 5 → 3: shared upper curvature and flattened lower region. • 2 → 0: rounded bottom strokes resembling closed loops.

The relatively small number of such cases and the absence of concentration in a single class confirm the stability and balance of the CNN's learned feature representations. The model effectively generalizes across the digit space, with errors limited to naturally ambiguous samples.

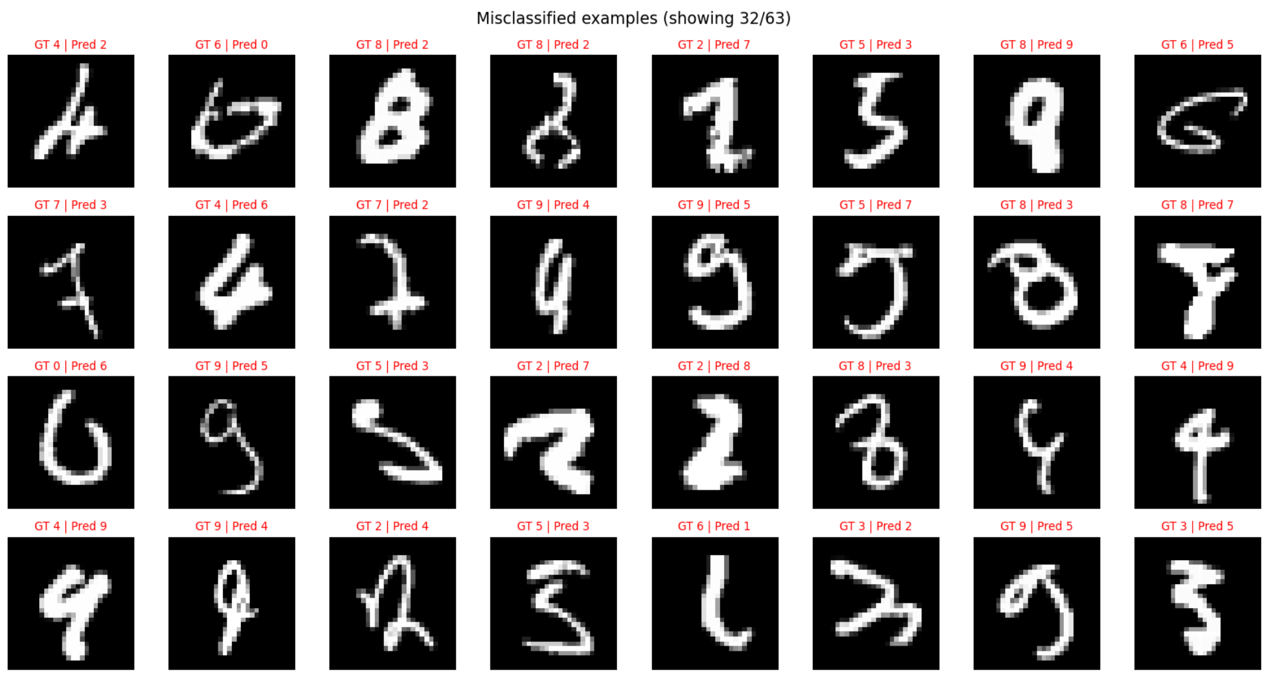


Figure 11 - Misclassified examples CNN

6.2.2 Confusion Matrix

The confusion matrix (Figure 12) supports these observations. Diagonal elements dominate the plot, indicating highly accurate per-class predictions. Each class maintains recall and precision values above 0.99, with the few off-diagonal counts corresponding to the aforementioned confusion pairs. Notably, digits 4, 7, and 9 account for most of the minor misclassifications, consistent with the morphological complexity of their handwritten variations.

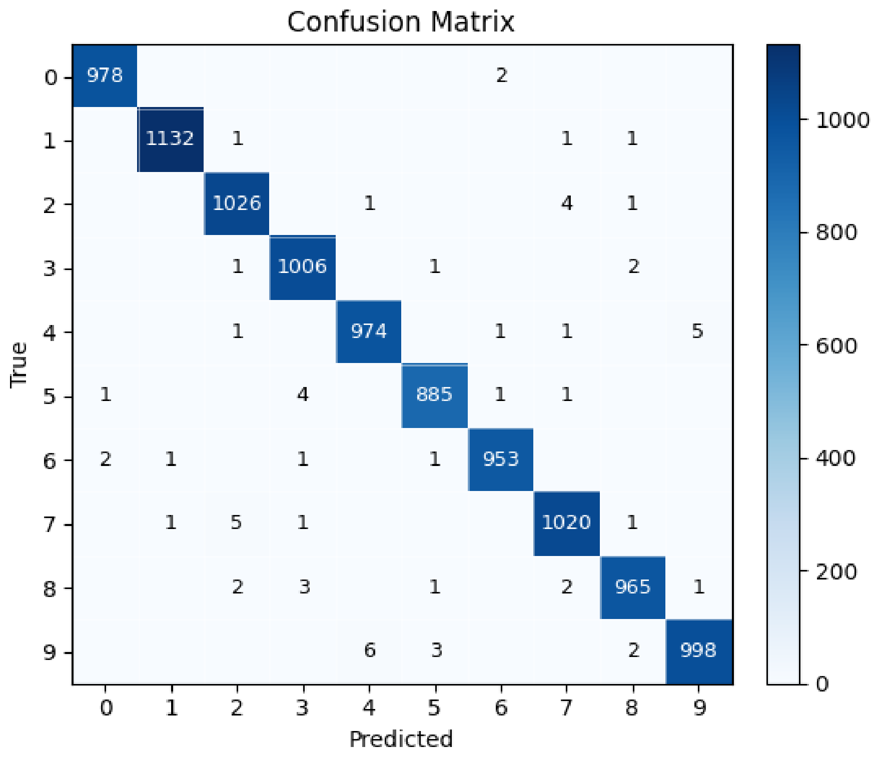


Figure 12 - Confusion Matrix Predicted Values CNN

6.2.3 Overall Assessment

The CNN outperforms the MLP by a small but consistent margin, benefiting from its capacity to capture spatial hierarchies through convolution and pooling. The overall error pattern reflects the intrinsic difficulty of certain digits rather than deficiencies in the model architecture. The combination of high precision, recall, and balanced class performance demonstrates that the CNN achieved a state-of-the-art level of accuracy for MNIST classification, with errors attributable primarily to ambiguous handwriting forms.

7. Conclusion

This project systematically explored the interdependencies among key hyperparameters in training Multilayer Perceptron (MLP) models on the MNIST handwritten digits dataset, with an additional Convolutional Neural Network (CNN) experiment included as an exploratory extension. Through a structured, two-stage sequential sweep, optimal configurations were identified for the MLP, while deeper insights were gained into how each hyperparameter influenced model performance, training stability, and generalization. The investigation reaffirmed the central role of batch size as a determinant of gradient noise and its interaction with the learning rate. The final MLP configuration, with a batch size of 64 and learning rate of 0.002, achieved a strong balance between convergence speed and training stability. The synergy between learning rate and regularization was also evident: a light weight decay (0.0001) provided sufficient control against overfitting while maintaining smooth optimization dynamics.

Normalization emerged as a key stabilizing mechanism. Incorporating Batch Normalization (use_bn=True) not only improved convergence consistency but also acted as an implicit regularizer, reducing dependence on explicit regularization methods. This interplay between normalization and weight decay proved essential in achieving robust generalization. The tuned MLP architecture, with moderate capacity (depth = 3, width = 256) and no dropout, demonstrated that well-balanced hyperparameters can maintain high accuracy and stability even in fully connected networks. The AdamW optimizer further contributed to stable and efficient convergence, confirming its strong compatibility with the selected learning rate and regularization setup. The final tuned MLP reached 99.36% test accuracy, with homogeneous per-class metrics and minimal variance between training and validation curves.

To contextualize the MLP results, a CNN model was included as an additional, exploratory baseline. Although not the core focus of the project, the CNN provided a meaningful architectural contrast by leveraging convolutional layers' inherent spatial priors. Even without Batch Normalization or dropout, the CNN achieved 99.60% test accuracy, surpassing the MLP's performance while maintaining lower validation loss. This difference highlights the structural advantage of convolutional representations for image data, where feature locality and translation invariance naturally enhance generalization. The CNN thus served as a valuable complementary benchmark, reinforcing that architecture and hyperparameters must be considered jointly when optimizing deep learning models.

Beyond numerical performance, qualitative prediction analysis provided additional insights. The MLP correctly classified 9,937 out of 10,000 test samples (63 misclassifications), while the CNN misclassified only 40. Most errors arose from ambiguous handwritten digits, for example, 4 ↔ 9, 5 ↔ 3, or 7 ↔ 1, cases that are visually challenging even for humans. The confusion matrices confirmed that these residual errors were sparse, symmetric, and morphology-driven rather than systematic, reflecting the models' strong generalization capabilities.

The final configurations:

MLP: batch_size=**64**, lr=**0.002**, optimizer=**AdamW**, weight_decay=**0.0001**, activation=**ReLU**, width=**256**, depth=**3**, use_bn=**True**, dropout_p=**0.0**, weight_init=**Xavier**

CNN: batch_size=**64**, lr=**0.0005**, optimizer=**Adam**, weight_decay=**0.0001**, activation=**ReLU**, width=**128**, depth=**3**, use_bn=**False**, dropout_p=**0.0**, weight_init=**He**

In conclusion, this work demonstrates that model performance in deep learning emerges from the coordinated interplay of architecture and hyperparameters. The dependency-aware tuning methodology adopted here proved effective in achieving both accuracy and training stability. The exploratory CNN comparison provided valuable context and confirmed the broader validity of the hyperparameter framework. Future work should apply this approach to more complex datasets and integrate dynamic learning rate schedules, warm-up strategies, and stronger regularization techniques to further investigate the limits of generalization, robustness, and transferability.

8. Bibliography

Hyperparameter optimization. (n.d.). In *Wikipedia*. https://en.wikipedia.org/wiki/Hyperparameter_optimization

Khan, S. A. (2024, October 30). Hyperparameter Optimization in Machine Learning. *arXiv*. https://arxiv.org/pdf/2410.22854 van Rijn, J. N. (2022, September 21). Hyperparameters 101: Your One-Stop Guide for All Deep Learning Models (Part 1). *AI Simplified*. https://ai.plainenglish.io/hyperparameters-101-your-one-stop-guide-for-all-deep-learning-models-part-1-86f163f58a53

van Rijn, J. N. (2022, September 21). Hyperparameters 101: Your One-Stop Guide for All Deep Learning Models (Part 2). *AI Simplified*. https://ai.plainenglish.io/hyperparameters-101-your-one-stop-guide-for-all-deep-learning-models-part-2-88b09f6709e6