

NNGPT: Rethinking AutoML with Large Language Models

Supplementary Material

1. Demo Video: TuneNNGen in Action

We include a time-lapse screen recording that demonstrates the end-to-end workflow of `TuneNNGen.py`: prompt assembly, one-shot YAML/code generation, schema validation, training and logging, and LoRA updates. It also shows typical failure modes (e.g., schema violations, missing functions) and how NNGPT surfaces them through the validator and executor stack. Refer to the supplementary video for the full run, including intermediate console outputs and per-model evaluation messages.

2. Extended Related Work and Detailed Comparisons

Automated neural network configuration has long been a focus of research, with prior work falling broadly into two categories: classical search-based optimization and recent LLM-driven generative approaches. Below we review key methods from both domains, focusing on the limitations addressed by NNGPT.

Search-Based Hyperparameter Optimization. Frameworks like Optuna [3], Hyperopt [6], and Google Vizier [19] use Bayesian optimization and Tree-structured Parzen Estimators (TPE) [5] to iteratively tune hyperparameters. Though effective, these methods demand significant compute, often requiring hundreds of training runs, and act as black-box optimizers, offering limited transparency into the configuration semantics or model behavior beyond their final scores.

AutoML and Neural Architecture Search (NAS). AutoML systems such as Optuna [3] and Hyperopt treat tuning as black-box optimization, while NAS frameworks like ENAS [50], AutoKeras [26], and DARTS [35] (plus variants like CDARTS, SDARTS, iDARTS) expand this to architecture generation via reinforcement learning or differentiable search. These methods, however, remain resource-intensive and adapt slowly to new tasks.

Efforts to stabilize NAS, such as RC-DARTS [63], NDARTS [10], and MSR-DARTS [39], still rely on costly infrastructure. NNGPT bypasses iterative search entirely: a single LLM call produces a complete, schema-valid configuration that is directly executable and logged for traceability. This design reduces computational cost while maintaining competitive accuracy, as shown by our DeepSeek-based results.

LLMs for Code and Configuration Generation. Recent advances in code-oriented LLMs such as Codex [8], Code Llama [51], and DeepSeek-Coder [21] enable automatic generation of hyperparameters, architectures, and

training scripts. Tools like GitHub Copilot[49] and Alpha-Code [31] further highlight their utility for programming tasks.

Most existing systems, however, lack integration with execution pipelines. Outputs are rarely validated empirically or tested for reproducibility. EvoPrompting [7] embeds LLMs into evolutionary search but requires repeated inference and lacks schema control.

NNGPT employs a one-shot generation strategy: a fine-tuned LLM produces a schema-compliant YAML configuration that is immediately validated and executed. Structured output enforcement (via Pydantic) and empirical feedback support reliable AutoML automation.

Prediction of Neural Network Accuracy. Despite advances in LLMs, many methods rely on high-level descriptions that miss implementation-specific patterns, or predict isolated metrics instead of full training dynamics. Yet performance prediction is vital for tuning efficiency and resource use in AutoML.

We address this via a multi-modal predictor that processes both structured metadata (e.g., task, dataset) and executable code (e.g., `nn_code`, `metric_code`). Key contributions include: (1) joint modeling of code and metadata, (2) efficient 4-bit QLoRA fine-tuning [14] to predict accuracy and stopping points, and (3) stratified data splits that prevent leakage while preserving balance.

LLM-Driven Neural Network Generation. LLMs are increasingly integrated into NAS pipelines. EvoPrompting uses a pretrained LLM as a mutation operator, requiring multiple inference calls and soft prompt-tuning [7]. LLMatic replaces hand-written mutations with CodeGen-generated code [44]. GPT-NAS [62] completes partial architectures, and Self-Programming AI [54] revises its own configs.

While effective, these methods rely on expensive iterative queries. In contrast, NNGPT performs one-shot generation: a single LoRA-tuned LLM call yields an executable, schema-compliant configuration with no further iteration required.

Schema-Constrained Generation. LLMs frequently generate syntactically invalid code in structured tasks. Grammar Prompting mitigates this by embedding formal grammars (e.g., Backus–Naur Form) into prompts [57], while RAG methods reduce errors by referencing external examples [4]. NNGPT adopts a different approach: it employs a strict Pydantic schema tailored to YAML configurations. Minor issues are auto-corrected; major ones trigger a single re-prompt, ensuring valid, executable outputs without relying on grammars or retrieval.

LLMs for Pipeline Optimization. LLMs have also been explored for broader pipeline automation. LLAMBO [36] frames Bayesian optimization as an LLM-driven dialogue, where the model proposes new candidates based on previous evaluations, outperforming classical BO in sparse regimes. Prompt-based HPO uses GPT-4 to refine hyperparameters from task descriptions[64], and NADA[23] generates full control algorithms, filtering them through simulation. In contrast, NNGPT focuses on end-to-end generation: a single LLM call outputs complete, runnable training recipes — data loaders, models, and optimization logic, without requiring iterative refinement.

Training Performance Prediction. Efforts to predict training outcomes fall into three main groups, each with key limitations. Statistical curve fitting [2, 16] extrapolates final metrics from early trends but ignores architecture and implementation, reducing generalizability. These approaches treat training as a numeric process, overlooking structural factors that influence convergence behavior.

Architecture-Based Performance Predictors. Methods like [38, 58] use textual summaries or engineered features to estimate model performance. However, such abstractions overlook critical implementation-level details, limiting their ability to capture the nuanced behaviors encoded in executable code.

Graph-Based Approaches. Techniques such as [15] represent models via explicit computational graphs, sometimes enhanced with differential modeling. Although more expressive than text, these methods require costly manual graph construction and still miss fine-grained code-level patterns such as custom regularization or dynamic scheduling, which strongly affect training outcomes.

Multimodal Prediction via Executable Code. While prior approaches either discard architectural context [2, 16] or rely on lossy text abstractions [38, 58], our method directly processes executable code snippets (`nn_code`, `metric_code`, `transform_code`) within the LLM prompt. This enables the model to learn implementation-specific patterns, e.g., custom regularization, branching, or metric logic, that strongly influence convergence but are inaccessible via abstract representations.

In contrast to graph-based methods [15] requiring manual topology construction, we combine three complementary inputs: (i) structured metadata, (ii) raw executable code, and (iii) early training dynamics. This multimodal context allows the model to align syntactic code patterns with numerical trends, revealing complex interactions that are missed by unimodal models.

Moreover, our LLM jointly predicts final accuracy and optimal stopping points within a shared latent space, capturing their intrinsic correlation — unlike prior works that treat them separately. This unified structure enhances generalization across tasks and architectures.

Table 1 situates NNGPT among classical AutoML toolkits and recent LLM-driven approaches. Bayesian HPO and NAS frameworks provide strong baselines for search over hyperparameters and architectures but rely on iterative evaluation and do not synthesize full training programs. Recent LLM-based methods introduce generation and closed-loop ideas, yet typically cover only parts of the pipeline or lack robust execution and prediction components. In contrast, NNGPT combines one-shot generation of executable specifications, code-aware accuracy prediction, and a closed feedback loop inside a single open-source framework.

3. Implementation

The NNGPT framework is implemented in pure Python and distributed as a pip-installable package (`pip install nn-gpt`). It requires Python 3.10 or higher and CUDA 12.6 for GPU-based training. The core components are exposed through three command-line interfaces: `NNAlter.py`, `TuneNNGen*.py`, and the container-friendly script `ab.gpt.train_neval`. These tools together compose the full pipeline introduced in Section 4.

3.1. Repository Structure

The codebase is organized into modular subdirectories. The `ab/gpt` directory contains the main scripts. Prompt templates, training configurations, and YAML schema definitions are defined in `ab/gpt/conf`, while LoRA fine-tuning utilities and other helper functions reside in `ab/gpt/util`.

3.2. Command-Line Interface

The script `NNAlter.py` performs seed model alteration using architectures from the LEMUR corpus. Each modified model is trained for a small number of epochs (default: 8, configurable via the `-e` flag). This utility relies on the `ab.gpt.util.AlterNN.alter` method and defaults to the `deepseek/ai/DeepSeek-R1-Distill-Qwen-7B` model.

The `TuneNNGen*.py` scripts handle one-shot LLM-based generation, YAML schema validation, full model training, and optional LoRA fine-tuning. A fast iteration mode can be enabled by passing the `-s` flag, which skips the alteration phase.

Additionally, `NNEval.py` offers standalone evaluation of generated architectures without invoking LoRA, enabling targeted testing of LLM outputs.

All required CUDA and MPI dependencies are pre-packaged in the public Docker image `abrainone/ai-linux`.

3.3. Prompting and Inference Flow

Prompt assembly begins with encoding task specifications, dataset, metric, and compute constraints into a structured JSON block. This prompt is then transformed into strict

YAML by the LLM. The DeepSeek 7B model is loaded via `transformers.AutoModelForCausalLM` in 8-bit precision, with LoRA adapters applied as needed.

3.4. Distributed Training Back-End

Model training is executed using PyTorch [48] and `torch.distributed`. MPI support is configured at install time using system-level dependencies. During training, per-epoch statistics such as loss, accuracy, mAP/IoU, GPU utilization, and Git SHA are recorded in a local SQLite database via SQLAlchemy. If an optional statistics module is available, the database can be exported to Excel-compatible dashboards using a single API call. If the optional `nn-stat` module is present, the database can be exported to Excel-compatible dashboards using a single call to `ab.stat.export`.

3.5. LoRA Fine-Tuning Loop

LoRA adapters are fine-tuned after every 50 new model training runs. The fine-tuning procedure consists of three epochs using a linear warm-up and cosine decay schedule. The resulting adapter checkpoints are published to the framework’s Hugging Face namespace, making them available for the next generation cycle.

3.6. Consistent Dependencies

NINGPT ensures consistency and reproducibility through a pre-built Docker image and a pinned `requirements.txt` file that specifies fixed versions of key libraries such as torch [48], transformers [61], and peft [41].

3.7. Prompt Generation Pipeline

Prompt generation is template-driven. A JSON file defines the structure of the prompt, including optional insertion points for code samples. This enables dynamic construction of prompts containing one or two models sampled from the database, under a given task. For training data generation, prompts may be further constrained to ensure shared or divergent characteristics between samples, improving task alignment while maintaining flexibility.

3.8. Offline Initialization of the Training Set

During early experiments, the raw DeepSeek-Coder-1.3B model frequently refused to modify input code, either claiming sufficiency or producing rule-violating changes. Introducing stricter task constraints often led to outputs that ignored parts of the prompt or simply returned unmodified code. These observations motivated the need for a bootstrapped training set.

To address this, the larger DeepSeek-R1-Distill-Qwen-7B model was used to generate initial examples. During

20 offline epochs, this model produced modified architectures by applying simple transformations, such as changing channel widths or digits in `Conv2d` layers, without requiring accuracy improvements. The first prompt template is shown in Listing 1. To diversify the training set, additional prompts requested the model to modify 2 to 4 digits randomly. Since DeepSeek-R1 7B reliably introduced at least one valid change, its outputs served as the initial training corpus for fine-tuning the smaller model.

During prompt engineering, both one-shot and few-shot examples were evaluated. However, neither significantly improved task fidelity on the raw models. In fact, DeepSeek-Coder-1.3B and DeepSeek-R1-Distill-Qwen-7B often lost track of task requirements or confused multiple examples. Zero-shot prompting consistently yielded more predictable behavior and was therefore adopted.

```
Define two 'nn.Conv2d()' layers as a neighboring
pair
when there are only non-Conv2d layers between
them.
```

```
Your task:
- Randomly pick and modify 1 or 2
  neighboring pairs
  of 'nn.Conv2d()' layers based on the
  rules below.
- Return ONLY the full result.
```

```
Instructions:
1. For each picked neighboring pair:
   - Change the out-channels of the
     former 'nn.Conv2d()'
     and the in-channels of the latter to
     the same new integer.
2. Modify the numbers directly in the code.
   DO NOT introduce new methods or
   parameters.
3. Do NOT pick the first neighboring pair;
   select randomly.
4. You MUST return the FULL CODE, including
   all classes - even if unmodified.
5. If no such pair exists:
   - Randomly change 2 digits in the code
     to new values.
6. Do not make any changes beyond the rules
   above.
```

```
Here is the input code for you to modify:
'''\n{nn_code}\n'''
```

Listing 1. Prompt instructing the LLM to modify neighboring `nn.Conv2d()` layers under strict rules, ensuring in-place edits and structural preservation of PyTorch code.

3.9. Automated LEMUR Dataset Extension

The extended dataset was generated using prompt templates, such as those of FractalNet [43], similar to those used during fine-tuning (e.g., Listing 2). Generated samples were stored alongside their originating code and task context to ensure proper evaluation. Certain datasets such as

COCO required specific preprocessing (e.g., fixed `Resize` operations), and segmentation tasks mandated the use of `IoU` rather than accuracy.

1. Change the layers in the following implementation to improve accuracy on image classification tasks.
2. Strictly follow these constraints:
 - Modify ONLY the layers in the code.
 - DO NOT introduce new methods or parameters.
3. Your response MUST:
 - Include the full code (including unchanged classes or functions).
 - Indicate the modified class explicitly.
 - Actually make at least one layer modification.
 - Not worry about decreasing accuracy - it's acceptable

```
Input code: '''\n{nn_code}\n'''
```

Listing 2. Prompt directing the LLM to modify existing layers in a neural network implementation under strict constraints, requiring full-code output and explicit identification of edits.

To extract generated code, regular expressions were used to match code blocks wrapped in triple backticks. To avoid infinite loops in parsing, the number of backticks was counted before applying regex-based extraction. Evaluation was performed via the LEMUR API. If parsing or execution errors occurred, the failing code and exception trace were returned to the LLM for repair. Each sample was given up to two correction attempts. Validated models were saved in structured directories compatible with the LEMUR dataset format and integrated into the local repository for downstream use.

3.10. Fine-Tuning for Accuracy Prediction

The model `Deepseek-coder-1.3b-Instruct` was fine-tuned via supervised learning to enable the prediction of validation accuracy changes resulting from architectural modifications. To align with the offline generation procedure described earlier, the fine-tuning objective was formulated as a code-to-code transformation task, where the model is presented with an original implementation, its evaluated accuracy, and the accuracy of a known improved variant. The expected output is the modified architecture that corresponds to the target accuracy, drawn from the LEMUR dataset.

The prompt, shown in Listing 3, encodes this training setup explicitly. It instructs the model to alter only layer definitions in order to achieve the desired accuracy gain, without introducing new methods or structural components. The fields `{accuracy}` and `{addon_accuracy}` represent the original and improved accuracy, respectively, and `{nn_code}` contains the complete source code of the base model. The expected output is the full code of the improved

model, including unchanged components, so the model also learns to preserve contextually irrelevant structures.

1. Change the layers in the following implementation to improve accuracy from `{accuracy}` to `{addon_accuracy}` on image classification tasks.
2. Strictly modify ONLY the layers in the code and STRICTLY DO NOT INTRODUCE NEW METHODS OR PARAMETERS
3. You should answer not only the class you modified but with FULL CODES INCLUDING EVERYTHING (Other classes, etc.) NOT CHANGED
4. You have to reply with full codes, don't be afraid of actually reducing the accuracy. Strictly check and make sure at least one layer being modified.

Listing 3. Prompt template used to generate training data by requesting constrained layer modifications aimed at improving model performance.

After eight epochs of supervised fine-tuning, the model consistently produced syntactically valid outputs that adhered to the required formatting and structural constraints. It successfully learned to apply targeted architectural modifications that align with observed changes in validation accuracy, and reliably preserved all components mandated by the LEMUR API specification.

In addition to replicating transformations observed during training, the model demonstrated the ability to compose valid variants across different architecture families. It consistently selected effective transformation strategies for given task-model pairs, producing high-quality outputs with minimal prompt engineering. This behavior reflects an emerging structural prior that favors stable and empirically grounded edits.

Moreover, the model preserved functional integrity even in cases where only subtle changes were required, maintaining coherence across unchanged code regions. These results confirm that the fine-tuned model captures not only the mapping between code edits and accuracy shifts, but also the appropriate output syntax and modular consistency necessary for downstream execution and evaluation.

3.11. Fine-Tuning for Hyperparameter Generation

In addition to accuracy prediction, NNGPT also supports automated generation of numeric hyperparameter values tailored to a given neural network, task, and dataset. To enable this capability, we fine-tuned a diverse collection of DeepSeek models using supervised learning on entries from the LEMUR dataset. The selected set spans a wide range of model scales and pretraining strategies, including both lightweight and instruction-tuned architectures. This diversity allowed us to examine how model size and initialization influence hyperparameter reasoning ability.

Each model was fine-tuned using the LoRA method to reduce memory overhead while preserving gradient flow

across transformer layers. LoRA rank and alpha were selected from 16, 32, 64, with a dropout of 0.05. All experiments employed a consistent setup that included the AdamW optimizer, a cosine learning rate scheduler, a fixed learning rate of 3×10^{-5} , gradient accumulation, checkpointing, and a training duration of up to 35 epochs depending on convergence behavior.

Training prompts were constructed to align with the LEMUR schema and encode structured conditioning variables: architecture code, target accuracy, number of training epochs, and required transformations. The input template, shown in Listing 4, requests the LLM to predict the values of specific hyperparameters that would lead the given model to achieve a known accuracy level.

```
### Input:
"Generate only the values (do not provide
any explanation) of the hyperparameters
({prm_names}) of a given model:
{entry['nn']} for the task:
{entry['task']} on dataset:
{entry['dataset']}, with transformation:
{entry['transform_code']}, so that the
model achieves accuracy =
{entry['accuracy']} with number of
training epochs = {entry['epoch']}. Code
of that model: {entry['nn_code']}"

### Response:
"Here are the hyperparameter values for
which the model will achieve the
specified accuracy: {prm_values}."
```

Listing 4. Prompt template used for fine-tuning LLMs to generate hyperparameter values based on model code, task, dataset, transformations, target accuracy, and training duration.

During evaluation, a modified version of the prompt (Listing 5) asked the model to produce hyperparameters aimed at maximizing accuracy, thereby testing generalization to unseen combinations.

```
### Input:
"Generate only the values (do not provide any
explanation) of the hyperparameters
({prm_names}) of a given model:
{entry['nn']} for the task: {entry['task']}
on dataset: {entry['dataset']}, with
transformation: {entry['transform_code']},
so that the model achieves the HIGHEST
accuracy with number of training epochs =
{entry['epoch']}. Code of that model:
{entry['nn_code']}"
```

Listing 5. Prompt for generating hyperparameter values that maximize accuracy, conditioned on model code, task, dataset, transformation logic, and training epochs.

All fine-tuned models were subsequently integrated into the NNGPT pipeline. Specifically, two of the best-performing checkpoints, HPGPT-DeepSeek-Coder-1.3b-Base and HPGPT-DeepSeek-R1-Distill-Qwen-7B, were

published to Hugging Face and are now directly accessible within the framework. These models can be used as plug-and-play components for automated hyperparameter suggestion in both evaluation and generation modes.

By extending the NNGPT workflow with fine-tuned LLMs for hyperparameter prediction, we close the loop between architecture generation and training configuration, offering a unified and fully automated AutoML solution. Experimental results evaluating the quality and effectiveness of these generated hyperparameters are presented in Section 3.3.

4. Methodology

NNGPT reconceptualizes LLMs as fully automated configuration agents for neural networks. Given a high-level task specification, the system follows the eight-stage pipeline depicted in Figure 1, encompassing architecture retrieval, prompt generation, model evaluation, and iterative self-improvement. Below, we detail each stage.

4.1. Stage 1: LEMUR API

NNGPT starts by querying the LEMUR Dataset API for executable architectures and metadata with standardized training and evaluation. LEMUR provides full implementations, uniform preprocessing, and reproducible metrics, so one-shot generation targets runnable code and uses LEMUR as a reliable metric oracle. Unlike torchvision [40], it adds reproducible benchmarking, native metric logging, and direct access to architecture code, making it well suited for generative AutoML.

4.2. Stage 2: Neural Network Data Retrieval

Following the API query, candidate architectures and associated metadata are retrieved from the LEMUR corpus. A dedicated script, `NNAlter*.py`, then applies a controlled set of architectural perturbations - including insertion/removal of layers, changes to channel widths, and alterations to residual connections. Each altered model undergoes a lightweight convergence check via a short warm-up training run (with the `-e` flag), ensuring that the resulting design remains viable. The resulting population of diverse, training-validated models is archived in an SQLite-backed repository, which serves as training data for LLM fine-tuning and architecture prediction.

4.3. Stage 3: Configuration Setup

After retrieval, a configuration scaffold is constructed that encapsulates the task, dataset, desired metric, and (optionally) a compute budget. This configuration is used to parameterize the prompt generator, enabling reproducible and deterministic experiment assembly. The prompt generation code is modular and extensible, supporting dynamic recon-

figuration and alternate LLM backbones through Hugging Face-compatible interfaces.

4.4. Stage 4: Prompt Assembly & Injection

The core input to the LLM is a structured prompt, assembled from the configuration data. This includes a JSON object describing the task, dataset, metric, and resource constraints. The prompt is formatted with system-level instructions that force the model to emit a strict YAML block beginning with `-yaml` and following a predefined schema.

The generative pass itself is carried out using a DeepSeek-V2 16B model equipped with 8-bit LoRA adapters fine-tuned on the altered-model corpus. Sampling is configured using $T = 0.2$, $p = 0.9$, and top-1 selection to minimize unnecessary exploration. Output is parsed with `ruamel.yaml` and validated using `pydantic`. Minor schema violations are auto-corrected, while major issues trigger a re-prompt with an embedded diagnostic message. Only configurations that pass validation proceed to execution.

4.5. Stage 5: LLM-Guided Architecture Generation

Once validated, the YAML output is interpreted as an experiment specification that includes a full architecture graph, optimizer parameters, loss functions, and data transformations. This file serves as the input to the training engine. Generated architectures may differ from their seed counterparts not only in topology but also in auxiliary properties such as normalization strategies or metric implementations. This stage integrates architectural creativity with empirical feasibility, ensuring that every configuration is both novel and executable.

4.6. Stage 6: Model Validation and Evaluation

Training is conducted using a distributed PyTorch engine (`ab/gpt/TuneNNGen*.py`), which uses `mpi4py` and `torch.distributed` for multi-GPU support. All datasets are accessed through the LEMUR API to ensure consistent preprocessing and evaluation. During training, metrics such as loss, accuracy, mAP, and IoU are logged per epoch alongside GPU utilization and configuration hashes. These logs are persisted in the same SQLite database as the architecture graph, enabling traceability and post-hoc analysis through the `nn-stat` visualization suite.

In parallel, a lightweight accuracy prediction module is used to estimate final model performance based on early training signals and static features. The predictor processes three input modalities: (i) structured metadata, (ii) executable code (model, transform, metric), and (iii) early accuracy values. A stratified split strategy ensures robust generalization and prevents data leakage across similar architectures.

4.7. Stage 7: LoRA Fine-tuning

After $k = 50$ successful training runs, a new fine-tuning iteration is triggered. LoRA adapters are retrained on the extended dataset of architecture–performance pairs using three epochs of linear warm-up and cosine decay. This updates the LLM’s internal representation of the architecture space, biasing future generations toward successful configurations. The updated checkpoints are immediately used in subsequent pipeline passes.

4.8. Stage 8: Database Storage

All artifacts, including trained weights, YAML specifications, metric logs, and architectural source code, are stored in the LEMUR NN dataset [55] and its public repository (<https://github.com/ABrain-One/nn-dataset>). LEMUR uses an SQLite-backed schema to support traceable, long-term reproducibility, enabling re-analysis, dataset expansion, and model distillation. The accumulated execution traces turn raw LLM generations into a curated, executable knowledge base that NNGPT can reuse for retrieval and continual updating.

4.9. AutoML Loop

Once all eight stages complete, the pipeline automatically restarts using updated model checkpoints, prompt templates, and database entries. This iterative process enables NNGPT to function as a self-improving AutoML cycle, where each generation, validation, and fine-tuning pass enhances the quality and diversity of future outputs.

5. Additional Experimental Results

5.1. Accuracy and Early-Stop Prediction

We instantiate the predictor H_ϕ as a code-aware LLM fine-tuned with 4-bit QLoRA on 1,200+ NNGPT/LEMUR runs spanning CNNs and Transformers [56] on the CelebA-Gender [37], CIFAR-10/100 [28], ImageNet [24], MNIST [30], Places365 [65], SVHN [45], and COCO [34] datasets. Each run is converted into a structured prompt with `nn_code`, `transform_code`, `metric_code`, `hyperparameters`, `dataset/task metadata`, `max epochs`, and `validation accuracies at epochs 1–2`; the model predicts the final best validation accuracy and its epoch. To prevent leakage, we use stratified group splits by (task, dataset, architecture). Per-dataset outcomes are summarized in Tab. 1.

We report two variants. The baseline (Exp. 1) is a QLoRA-tuned code LLM without additional balancing/strong regularization; it achieves $\text{RMSE} = 0.1449$, $r = 0.7766$, and $R^2 \approx 0.55$ globally (see Tab. 8). Correlation varies by dataset, ranging from moderate on CIFAR-10/SVHN to strong on COCO, while roughly 50% and 75% of predictions fall within 5% and 10% of the true final accuracy, respectively. The dataset-wise heterogeneity is visible

Dataset	RMSE [95% CI]	Pearson r (p)	Notes
CelebA-Gender	0.0497 [0.044, 0.056]	0.2125 ($p = 2.38 \times 10^{-7}$)	Low correlation despite low RMSE
CIFAR-10	0.1589 [0.150, 0.168]	0.3472 ($p = 3.77 \times 10^{-30}$)	Moderate correlation
CIFAR-100	0.1668 [0.155, 0.178]	-0.0276 (<i>ns</i>)	Very poor ranking ability
ImageNette	0.2439 [0.221, 0.266]	0.4973 ($p = 1.25 \times 10^{-36}$)	High error, moderate r
MNIST	0.0331 [0.025, 0.040]	0.4224 ($p = 1.04 \times 10^{-22}$)	Excellent precision
Places365	0.1672 [0.135, 0.193]	-0.0297 (<i>ns</i>)	Weak generalization
SVHN	0.0605 [0.058, 0.063]	0.3587 ($p = 4.08 \times 10^{-20}$)	Low r , tolerable RMSE
COCO	0.1494 [0.138, 0.160]	0.8173 ($p < 10^{-4}$)	Strongest dataset correlation
Overall	0.1449	0.7766	Captures trends; uneven by dataset

Table 1. **Per-dataset accuracy prediction (baseline, Exp. 1).** RMSE reflects regression error for final accuracy; Pearson r measures ranking quality.

Metric	Value
RMSE	0.2567 [0.248–0.265]
MAE	0.1709
R^2	0.2885
Pearson correlation	0.6409 ($p = 1.25 \times 10^{-204}$)
Within 5% tolerance	12.30%
Within 10% tolerance	45.01%
Samples	1764

Table 2. **Experiment 2 (balanced + regularized) – global metrics.** Aggregate performance of the alternative predictor across all datasets; RMSE shows 95% CI.

in Tab. 1. We use the predictions both as a proxy for final performance and to propose an early-stop epoch $T' = \hat{t}_*$ for low-potential runs.

An experiment 2 applies inverse-frequency balancing and stronger regularization (higher dropout/weight decay, fewer epochs). It underperforms: RMSE rises to 0.2567, R^2 drops to 0.2885, r to 0.6409, and only 12.3%/45.0% of predictions fall within 5%/10% tolerance (Tab. 2; also compared side-by-side in Tab. 8). We therefore adopt the baseline predictor in the full NNGPT loop.

The baseline is already useful for early-stop and priority scheduling, but (i) correlation varies across datasets; (ii) aggressive balancing/regularization can suppress adaptation in LoRA-tuned LLMs; and (iii) adding uncertainty estimates is a promising next step for safer automated stopping.

Dataset	Baseline avg	RL avg (n)	Δ (avg)	Baseline best	RL best
CIFAR-100	0.0461	0.0163 (5)	-0.0298	0.2079	0.0253
MNIST	0.7088	0.9876 (13)	0.2788	0.9906	0.9921
SVHN	0.2526	0.8148 (14)	0.5622	0.9032	0.9068

Table 3. **RL one-epoch results vs LEMUR baselines.** Baseline avg and best come from LEMUR *AlexNet* (*ast-dimension*); RL avg is over generated *AirNet* models (n shown). Δ is RL avg minus baseline avg.

5.2. Reinforcement Learning Setup

We report our first RL results by generating 32 executable AirNet models and training each for a single epoch on MNIST [30], SVHN [45], and CIFAR-100 [28]. As a non-RL baseline we use one-epoch averages and best scores from the LEMUR AlexNet [29] (*ast-dimension*) family; the baseline values are embedded in Tab. 3. RL-generated models match or exceed the baseline on the simpler datasets: on MNIST the average and best accuracies are 0.9876 and 0.9921, and on SVHN they are 0.8148 and 0.9068. Performance on CIFAR-100 is weak with an average of 0.0163 and a best of 0.0253, while the LEMUR best reaches 0.2079. These initial results indicate that the RL loop reliably produces runnable architectures and can achieve strong one-epoch accuracy on easier datasets; harder regimes appear to require more iterations, longer training, and reward shaping. We treat this study as a diagnostic feasibility evaluation due to small per-dataset sample sizes and single-epoch budgets.