LEMUR 2: Unlocking Neural Network Diversity for AI

Supplementary Material

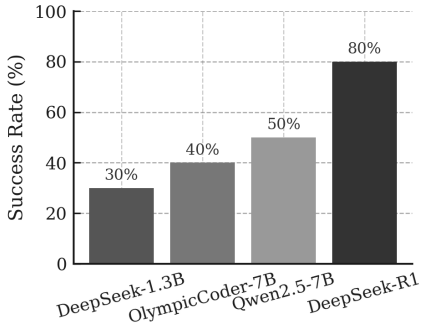

Figure 1. Success rates for different LLMs in image captioning pipeline.

1. Image Captioning Details

1.1. LLM and Prompting Strategy

Generative model. All captioning architectures analysed in NN-Caption are generated with **DeepSeek-R1-0528-Qwen3-8B**, an 8B-parameter instruction-tuned code LLM. Preliminary trials with a smaller 1.3B code model and a 7B variant showed a lower rate of schema-compliant generations and more frequent hallucinated PyTorch APIs (e.g., spurious layers such as `nn.SelfAttention`). The 8B model was therefore adopted for the experiments reported here. The overall success rates with different LLMs can be seen in figure 1.

Unless otherwise stated, we sample with temperature 0.8 and generate a *single* model per prompt (no beam search or sampling of multiple candidates). Other decoding hyperparameters are left at their library defaults.

Baseline code and Net API. Each prompt includes the code of a baseline ResNet–LSTM captioner (ResNet-50 encoder + LSTM decoder) implemented in LEMUR and conforming to the unified `Net` interface. The LLM is instructed to modify this base architecture using inspiration from classification snippets while *strictly preserving* a required API:

- A class `Net(nn.Module)` with constructor `__init__(self, in_shape, out_shape, prm, device)`.
- Methods `train_setup(self, prm)`, `learn(self, train_data)`, and `forward(self, ...)`.
- A top-level function `supported_hyperparameters()` returning exactly {`'lr'`, `'momentum'`}.

The prompt emphasizes that the output must be a single, runnable Python file that implements this contract.

Prompt structure. We use a one-shot prompting strategy with three main components:

1. **System role.** A system message instructs the LLM to act as an expert PyTorch architect and to output *only* Python code inside a single fenced block, with no natural-language commentary or partial answers.
2. **API, shapes, and constraints.** The prompt enumerates the mandatory methods listed above and specifies tensor shapes and training behavior:
   - The encoder must reuse or extend a CNN- or ViT-style backbone from the provided classification snippets. For CNNs, the classification head must be removed and replaced by a feature head that produces a tensor of shape `[B, 1, H]` via `AdaptiveAvgPool2d` followed by a linear layer; for ViT-style encoders, patch tokens may be returned as encoder memory.
   - The decoder must be either recurrent (LSTM or GRU) or Transformer-based. The prompt provides guidelines for typical hidden sizes (e.g., 640 or 768) and examples of valid uses of `nn.TransformerDecoderLayer` and `nn.MultiheadAttention`.
   - The LLM is explicitly forbidden from introducing undefined symbols or custom layers outside the `torch.nn` namespace (e.g., `nn.SelfAttention` is disallowed; standard `nn.MultiheadAttention` should be used instead). It is also reminded that the Transformer decoder must not expect the vocabulary size in its constructor.
   - The model must train with teacher forcing: inputs are `captions[:, :-1]`, targets are `captions[:, 1:]`, and the `forward` method must return a tuple `(logits, hidden_state)`, where `logits` has shape `[B, T-1, vocab_size]`. The prompt encourages adding assertions to enforce this shape contract.
   - In `train_setup`, the model must define `self.criterion = nn.CrossEntropyLoss(ignore_index=0, label_smoothing=0.1)` and use an `AdamW` optimizer, reading `'lr'` and `'momentum'` from the `prm` dictionary when present.
3. **Architectural inspiration and diversity.** In addition to the baseline captioner, the prompt includes code snippets from LEMUR classification models to act as "building blocks" (e.g., ResNet, EfficientNet, ConvNeXt, ViT). We use two main inspiration regimes:
   - **5-snippet context (C5C).** A random subset of 5 clas-

sification models (e.g., EfficientNet-B0, ConvNeXt-T, DenseNet-121, VGG16, ViT) is provided, excluding the snippet corresponding to the baseline encoder to avoid trivial copies.

- **10-snippet context (C10C).** The same pool is expanded to 10 distinct classification snippets (adding, e.g., MobileNetV2, Inception-v3, SqueezeNet), resulting in longer prompts and a more challenging generation task.

The LLM is required to make at least three structural changes relative to the baseline model (e.g., inserting SE or CBAM blocks, changing the decoder type, adjusting hidden sizes, or adding attention) while maintaining the required API. Hints such as "larger hidden sizes and multi-head attention generally improve BLEU" and "keep dropout modest" nudge the LLM toward plausible captioning architectures.

In addition to the main C5C and C10C settings, a small number of models come from earlier runs with 1 and 8 classification snippets, denoted C1C and C8C in the tables below. These runs use the same `Net` API and training protocol but slightly different context lengths.

1.2. Code Extraction, Sanitization, and Repair

The raw LLM output may contain extra text, multiple fenced blocks, or minor syntactic errors. To maximize the yield of runnable models, we use an automated extraction and repair loop before integrating code into LEMUR.

- **Code extraction.** A parser locates the most likely complete Python block (prioritizing the one containing `class Net` and the required methods) and discards any surrounding natural-language text.
- **Sanitization.** We apply several deterministic fix-ups:
  - Strip markdown fences and any `<think>` or similar trace annotations.
  - Ensure that essential imports (e.g., `import torch.nn as nn`) are present exactly once.
  - Override `supported_hyperparameters()` so that it always returns `{'lr', 'momentum'}`, regardless of what the LLM proposed.
  - Perform bracket balancing at end-of-file to fix occasional missing closing brackets or parentheses.
- **AST validation and repair loop.** After sanitization, we run Python's abstract syntax tree (AST) parser. If parsing fails or basic structural checks fail (e.g., missing `Net` class or required methods), we enter a repair loop: the error message and the offending code are fed back to the LLM with a strict "repair" prompt. We allow up to two repair iterations, after which the model is discarded if it still fails to parse.

Only models that pass this pipeline are registered into LEMUR and scheduled for training.

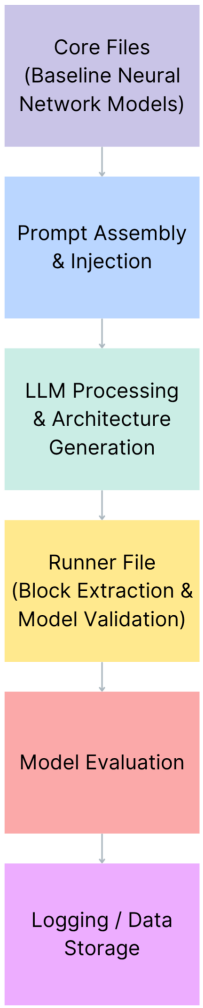

Figure 2. Image Captioning System Architecture

1.3. Training and Evaluation Protocol

Dataset and preprocessing. All models are trained on **MS COCO 2017** using the standard split: 118k training images and 5k validation images. Images are resized and center-cropped to 224×224 and normalized with ImageNet statistics, reusing LEMUR's standard preprocessing pipeline. Captions are tokenized into a vocabulary of 10k words and wrapped with `<SOS>` and `<EOS>` tokens; the padding token `<PAD>` is mapped to index 0, which is used as `ignore_index` in the loss.

Definition of a successful model. A generated architecture is considered *successful* and included in the final pool if

1. its code passes the sanitization and AST checks,
2. it trains for three full epochs on MS COCO without runtime errors or NaNs, and

3. it produces finite BLEU-4 scores on the validation set.

In total, NN-Caption contributes **357** successful captioning architectures that meet these criteria.

Training hyperparameters. Unless overridden by the model's own `learn` method, we use the following defaults:

- **Epochs:** 3 per model (limited by the computational cost of evaluating hundreds of candidates).
- **Batch size:** 32.
- **Optimizer:** AdamW (`torch.optim.AdamW`).
- **Learning rate:** 1×10^{-3} by default; models may override this via the `'lr'` entry in `prm`.
- **Loss:** cross-entropy with `label_smoothing=0.1` and `ignore_index=0`.
- **Teacher forcing:** inputs `captions[:, :-1]`, targets `captions[:, 1:]` throughout training; `forward` returns logits over the target sequence length.

The only hyperparameters exposed through `supported_hyperparameters()` are `'lr'` and `'momentum'`. Some LLM-generated models introduce additional training tweaks inside their `learn` method, such as learning rate schedulers (e.g., `ReduceLROnPlateau`) or modified default learning rates (e.g., 5×10^{-4}); these are preserved but have modest impact within the 3-epoch budget.

1.4. Extended Results and Model Families

Model families and prefix codes. Successful models are grouped into families by (i) the context setting (number of classification snippets) and (ii) the decoder type chosen by the LLM. Table 1 summarizes the resulting 357 architectures. Prefixes such as `C5C-RESNETLSTM` and `C5C-ResNetTransformer` correspond to schema-valid, archived models in the LEMUR database.

Across all families, we observe that the 5-snippet prompts (C5C) yield a higher proportion of successful models than the 10-snippet prompts (C10C). With more snippets, the LLM tends to attempt more ambitious hybrids, which increases the frequency of tensor shape errors that are not fully resolved by the repair loop and leads to fewer models passing end-to-end training.

Validation performance on MS COCO. Table 2 reports BLEU-4 on the COCO 2017 validation split after 3 epochs for the main baselines and representative generated families. The two manual baselines are fully engineered ResNet-50 captioners (LSTM and Transformer). Generated models are evaluated under the same training budget and data split.

As expected under the restrictive 3-epoch training budget, the LLM-generated models do not match the performance of the carefully engineered manual baselines. Under this matched-budget setting, the best generated model—a ResNet-50 encoder with a Transformer decoder and a 768-dimensional hidden size with 8 attention heads—reaches BLEU-4 of 0.1192, which is below the manual ResNet-50 + LSTM and ResNet-50 + Transformer baselines but still represents valid captioning performance. Separately, a longer 50-epoch follow-up run of the strongest generated ResNet-50 + Transformer configuration reaches BLEU-4 = 0.317, narrowing the gap to the manual ResNet-50 + LSTM baseline at 0.3246.

1.5. Qualitative Architectural Analysis

Beyond aggregate metrics, we briefly highlight qualitative patterns observed in the NN-Caption search space.

1. **Attention mechanisms.** Transformer-based decoders naturally implement multi-head attention over encoder features. In several LSTM-based models, the LLM additionally introduced explicit attention modules on top of the encoder output, typically in the form of simple additive or dot-product attention mechanisms. These were not requested explicitly in the prompt but emerged from the combination of instructions and attention-related snippets (e.g., ViT self-attention blocks), indicating that the LLM is able to transplant attention concepts across architectures.
2. **Backbone augmentation vs. replacement.** Many successful models retain a ResNet-50-style backbone but augment it with components borrowed from classification snippets. A common pattern is the insertion of Squeeze-and-Excitation (SE) or Convolutional Block Attention Modules (CBAM) into the encoder pipeline, often after the final convolutional stage. Such models (e.g., ResNet+SE+LSTM) tend to approach but not exceed the best generated Transformer-based architecture within 3 epochs. All generated models, in turn, remain below the performance of the fully tuned manual baselines in Table 2.
3. **Alternative encoders.** A smaller subset of models replaces ResNet-50 entirely with other LEMUR backbones such as ConvNeXt-T, while retaining a Transformer decoder. These models generally train successfully and achieve BLEU-4 scores around 0.10–0.12, qualitatively producing somewhat richer captions than many LSTM-based variants, but they remain slightly below the best ResNet-Transformer hybrid and well below the manual baselines in this short-training regime.
4. **Decoder choice and complexity.** Across runs, the LLM predominantly chooses either LSTM or Transformer decoders. GRU-based decoders were explored only in a few preliminary, manually forced variants and did not yield improvements, so they are not included among the 357 models summarized in Table 1. Transformer decoders tend to yield the highest BLEU-4 among generated models but require the LLM to correctly handle more complex API usage (e.g., `nn.MultiheadAttention`, masking). LSTM-based models are simpler and more robust but typically

Table 1. Distribution of successful NN-Caption architectures by context and decoder type. C1C/C8C correspond to earlier 1- and 8-snippet runs; C5C/C10C are the main 5- and 10-snippet settings used in the report.

Prefix Code	Inspiration Context	Decoder Type	Count
C1C-RESNETLSTM	1 snippet (C1C)	LSTM	1
C5C-RESNETLSTM	5 snippets (C5C)	LSTM	100
C10C-RESNETLSTM	10 snippets (C10C)	LSTM (+Attn)	3
C5C-ResNetTransformer	5 snippets (C5C)	Transformer	250
C8C-ResNetTransformer	8 snippets (C8C)	Transformer	3
Total			**357**

Table 2. Validation BLEU-4 on MS COCO 2017 after 3 epochs. Manual baselines are fully engineered. Generated families correspond to NN-Caption architectures discovered by the LLM.

Model / Family	Origin	BLEU-4
ResNet-50 + LSTM (Manual)	Manual	**0.3246**
ResNet-50 + Transformer (Manual)	Manual	0.2336
C5C-ResNetTransformer (best member)	Generated	0.1192
C5C-RESNETLSTM (best member)	Generated	0.0914
C10C-RESNETLSTM (best member)	Generated	0.0637

plateau at lower BLEU-4 scores.

5. **Prompt length vs. robustness.** Consistent with the quantitative success-rate analysis, C10C runs (10 snippets) exhibit more frequent failure modes during code execution: over-ambitious attempts to chain multiple heterogeneous blocks (e.g., mixing ConvNeXt stems with ViT patch embeddings) often lead to tensor rank mismatches that cannot be repaired automatically. C5C runs, with shorter and more focused prompts, tend to produce architectures with fewer such issues, trading some diversity for higher reliability.

2. Text-to-Text Details

Metrics. We add two standard metrics for text generation:

- **Perplexity (PPL)** To unify optimization and dashboards across tasks, all metrics in LEMUR are reported as "higher is better" scores in $[0, 1]$ (the higher is better):

$$s_{\text{ppl}} = \text{clip}_{[0,1]}\left(1 - \frac{\ln(\text{PPL}) - \ln P_{\min}}{\ln P_{\max} - \ln P_{\min}}\right). \quad (1)$$

Reverse conversion:

$$\text{PPL} = \exp\Big(\ln P_{\min} + (1 - s_{\text{ppl}})\big(\ln P_{\max} - \ln P_{\min}\big)\Big). \quad (2)$$

- **BLEU** For generated text $\hat{y}$ against a reference y, BLEU estimates corpus-level n-gram precision with a brevity penalty: $\text{BLEU} = \text{BP} \cdot \exp\big(\sum_{n=1}^{N} w_n \log p_n\big)$, where p_n is modified n-gram precision and w_n are weights (by default uniform). Higher is better. Our `BLEUMetric` aggregates over the validation set and supports smoothing for low-count n-grams.

3. Text-to-Image Details

3.1. Experimental Setup and LEMUR Integration

To keep the task focused and computationally tractable, all experiments were conducted on a subset of MS COCO filtered to *car-related* captions. This car-focused subset was used consistently across UNet-D, the GAN, and the CVAE-GAN, allowing a direct comparison of semantic alignment via CLIP scores. CLIP text–image similarity is computed between generated images and their conditioning prompts and stored in the LEMUR backend alongside loss curves and configurations.

Two technical aspects required additional work beyond the main LEMUR design:

- **Non-tensor text labels.** The existing data loaders primarily handle tensor-valued labels. For text-to-image, we extended the pipeline to treat natural-language captions as first-class inputs while still exposing integer tokenizations where needed (e.g., for LSTM encoders).
- **Metric integration and hyperparameter search.** We added tuple-aware handling in the validation loop so that CLIP metrics can consume both images and raw text. Optuna was integrated to explore learning rates and loss weights for the CVAE-GAN, which proved particularly sensitive.

3.2. UNet-D: Diffusion Model

3.2.1. Architecture

The diffusion family is instantiated as **UNet-D**, a DDPM-style model with a custom U-Net backbone for text conditioning.

- **Text conditioning.** A pre-trained CLIP text encoder produces 512-dimensional embeddings for each prompt. These are concatenated with sinusoidal time-step embeddings to form a unified conditioning vector. Instead of classifier-free guidance, the conditioning signal is injected directly into each residual block of the U-Net, giving fine-grained control over the denoising trajectory.

- **Encoder–decoder design.** The encoder uses three downsampling stages with `ResBlockWithAttention` layers, expanding channel width from 128 → 256 → 512, followed by a 1024-channel bottleneck. The decoder mirrors this structure using transposed convolutions and skip connections to recover spatial detail.
- **Attention integration.** In early encoder stages, `CrossAttention` layers fuse CLIP text context with image features. Deeper in the network and near the bottleneck, `SelfAttention` layers maintain spatial and global consistency.

3.2.2. Training under single-GPU constraints

Training UNet-D within a single-GPU budget required several concrete optimizations beyond the generic diffusion description in the main text:

- **Memory and throughput.** We enabled `torch.compile` for the U-Net, which yielded roughly a 40% speed-up, and combined automatic mixed precision, gradient checkpointing, and gradient accumulation (over 4 steps) to keep peak memory within limits.
- **Model simplification.** An earlier, more complex design with additional cross-attention and classifier-free guidance failed to converge (CLIP ≈ 0.0). We therefore reverted to the simpler UNet-D configuration above, which produced stable gradients and usable CLIP scores.

3.2.3. Results

3.3. GAN-Based Text-to-Image Model

3.3.1. Architecture

The second model family uses a conditional GAN with both the generator and discriminator conditioned on text:

- **Generator.** The generator encodes text with an LSTM and feeds the resulting embedding into a stack of transposed convolutional layers, incorporating self-attention in later stages. The design is adapted from Self-Attention GAN (SAGAN), but integrated into LEMUR's module registry and trained on the same car-focused COCO subset.
- **Discriminator.** The discriminator is CNN-based, with an additional LSTM branch to process the text sequence. Image and text features are fused to judge the realism and text-alignment of each pair.

3.3.2. Training phases and stabilisation

GAN training required three distinct debugging and stabilisation phases:

1. **Runtime debugging.** Early runs produced the PyTorch error `"Boolean value of Tensor with more than one value is ambiguous"`. This was traced to conditional logic on tensor-valued inputs. The `forward` methods were refactored so that both raw string lists and already-tokenized tensors are handled explicitly, eliminating ambiguous boolean checks.
2. **Mode collapse.** The generator initially dominated the discriminator, leading to mode collapse. We adopted a discriminator-favoured update schedule with a *2:1* update ratio (two discriminator steps per generator step) and reduced the generator learning rate to one quarter of the discriminator's rate.
3. **Gradient explosion.** To avoid exploding gradients and oscillatory loss behaviour, we reduced the base learning rate from 0.0004 to 0.0002 and lowered Adam's `beta1` from 0.9 to 0.5, effectively reducing momentum and smoothing the training signal.

3.3.3. Results

After these changes, the GAN could be trained for 150 epochs with stable adversarial dynamics. CLIP similarity reaches approximately 0.214 on the validation prompts. Qualitative inspection shows a progression from pure noise to sky–ground layouts (epochs 1–20), then to coherent car silhouettes with plausible colour palettes (epochs 50–75), and finally to more refined shapes with highlights suggestive of metallic surfaces (epochs 120–150).

3.4. CVAE-GAN: Architectural Progression and Final Design

3.4.1. From baseline CVAE to modernized architecture

The CVAE-GAN family underwent several iterations before reaching the final configuration.

- **Version 1 (Baseline CVAE).** The first model used a DistilBERT text encoder with an image decoder producing 128×128 outputs. It achieved CLIP scores around 0.21–0.22 on the car subset but suffered from limited visual detail due to the low resolution.
- **Version 2 (High-capacity VAE).** To improve fidelity, we scaled to 1024 convolutional channels, a 512-dimensional latent vector, and 256×256 resolution, adding a VGG-based perceptual loss. However, this version exhibited severe checkerboard artefacts, traced to the heavy use of `ConvTranspose2d` upsampling.
- **Version 3 (Modernized CVAE).** The breakthrough configuration replaced transposed convolutions with an `Upsample` + `Conv2d` decoder, and swapped DistilBERT for a CLIP text encoder. This eliminated checkerboard artefacts and improved semantic alignment, with peak CLIP scores around 0.276 on the validation set.

3.4.2. Final CVAE-GAN architecture

The final **CVAE-GAN** uses Version 3 as the generator within an adversarial framework:

- **Generator.** The generator is the modernised conditional VAE from Version 3, leveraging CLIP text conditioning and the upsample-plus-convolution decoder to produce structurally coherent images at 256×256 resolution.

- **Discriminator.** A PatchGAN-style discriminator operates on image patches, simplified by removing one convolutional layer to avoid overwhelming the pre-trained generator during fine-tuning.
- **Loss composition.** Training combines:
  - L1 reconstruction loss for pixel-wise fidelity,
  - VGG-16 perceptual loss for structural similarity,
  - KL divergence for latent-space regularization,
  - adversarial loss via `BCEWithLogitsLoss`.

3.4.3. Training methodology and results

Training CVAE-GAN was highly sensitive to hyperparameter choices, so we relied on manual tuning guided by Optuna sweeps. Stable training required:

- significantly reducing the adversarial loss weight relative to reconstruction and perceptual losses,
- increasing the weight on L1 and perceptual terms to preserve structure,
- asymmetric learning rates: generator LR 3×10^{-6} and discriminator LR 2×10^{-9}.

With this configuration, the final CVAE-GAN attains a CLIP text–image similarity of 0.2751. It effectively mitigates classical VAE blurriness and maintains strong structural coherence. Nonetheless, qualitative inspection reveals that many samples remain abstract or texture-like rather than fully photorealistic, highlighting architectural capacity limits at this resolution and scale.

Table 3 summarises the semantic alignment of the three model families on the car-focused dataset.

4. Mixture-of-Experts Details

4.1. Integration into LEMUR and Experimental Setup

Unless stated otherwise, all experiments use the standard CIFAR-10 protocol:

- **Dataset:** CIFAR-10 with 50,000 training and 10,000 test images.
- **Preprocessing:** Per-channel normalization using training-set statistics.
- **Data augmentation:** Random horizontal flips and random crops with 4-pixel padding for all MoE families; mixup with $\alpha = 0.2$ is additionally used for the heterogeneous MoE.
- **Training duration:** Between 50 and 200 epochs, depending on configuration.

The overarching goal of the MoE integration is two-fold: (i) to investigate whether sparse expert routing can outperform existing LEMUR baselines by 2–10% absolute accuracy, and (ii) to add architecturally diverse, expert-based models to the LEMUR corpus for downstream AutoML.

4.2. MoE Architectures and Gating Mechanisms

We consider three MoE families:

1. homogeneous MoEs with identical CNN-based experts (Experiment 1),
2. MoEs with AlexNet experts (Experiment 2),
3. a heterogeneous MoE combining four different backbones (Experiment 3).

In all cases, the gating network maps an input image to a vector of routing logits over experts; routing is either sparse (top-2) or soft (weighted combination).

4.2.1. Experiment 1: Homogeneous MoEs

The first family (models MoE-0707 through MoEv8) employs homogeneous experts and sparse routing:

- **Experts:** Each expert consists of a convolutional feature extractor with six convolutional blocks, interleaved with batch normalization and dropout, followed by a three-layer fully-connected head.
- **Gate:** A two-layer feedforward network with batch normalization produces routing logits. During training, Gaussian noise is added to the logits to encourage exploration and avoid early collapse.
- **Routing:** There are $N = 8$ experts. For each input, the gate selects the top-2 experts by routing weight; only these experts are evaluated, and their logits are averaged (or summed) for the final prediction.
- **Load-balancing loss:** A small auxiliary term encourages uniform expert usage:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{CE}} + \lambda_b \mathcal{L}_{\text{balance}} \tag{3}$$

where $\mathcal{L}_{\text{balance}}$ measures the deviation of empirical routing frequencies from the uniform distribution over experts.

MoEv7 is the best-performing homogeneous configuration and serves as the representative model for this family.

4.2.2. Experiment 2: AlexNet-Expert MoEs

The second family (MoEv9-AlexNet through MoEv9-AlexNetv4) replaces the generic experts with AlexNet-based CNNs:

- **Experts:** Each expert uses the standard AlexNet convolutional backbone as a feature extractor, followed by a task-specific classifier head. Custom initialization schemes were explored to promote specialization across classes.
- **Gate (ImprovedGate):** The gating network combines convolutional and fully connected layers and uses a temperature-controlled softmax. A learned noise scaling factor controls the magnitude of injected noise during training, stabilizing routing decisions.
- **Routing:** As in Experiment 1, routing is top-2: only the two experts with the highest gate scores are activated for each input.
- **Load balance and diversity:** In addition to the load-balance loss, AlexNet-based MoEs include an inter-expert diversity term:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{CE}} + \lambda_b \mathcal{L}_{\text{balance}} + \lambda_d \mathcal{L}_{\text{diversity}} \tag{4}$$

Table 3. Summary of text-to-image models and CLIP-based semantic alignment on the car-focused COCO subset.

Model Family	CLIP (val)	Characteristics
UNet-D (Diffusion)	0.17–0.24	Robust denoising, moderate detail, overfits after $\sim$100 epochs
GAN (LSTM+CNN)	≈ 0.214	Stable after 150 epochs with 2:1 D:G updates, clear structural evolution
CVAE-GAN	0.2751 (peak)	Best semantic alignment; artifact-free but mostly abstract textures

where $\mathcal{L}_{\text{diversity}}$ penalizes highly correlated expert predictions, encouraging more diverse decision boundaries.

Among these models, MoEv9-AlexNetv3 provides the best trade-off between accuracy, expert specialization, and computational cost.

4.2.3. Experiment 3: Heterogeneous MoE

The final configuration, `MoE-hetero4-Alex-Dense-Air-Bag`, introduces architectural diversity by combining four distinct CNN backbones:

- **Experts:** AlexNet, DenseNet, AirNet, and BagNet serve as four separate experts, each with its own convolutional feature extractor and classifier head.
- **Gate:** A lightweight convolutional gating network produces soft routing weights $g_i(x)$ over the four experts for each input x.
- **Soft routing:** Unlike the sparse top-2 routing used in the homogeneous and AlexNet MoEs, the heterogeneous MoE uses soft routing:

$$y(x) \;=\; \sum_{i=1}^{4} g_i(x)\, f_i(x) \tag{5}$$

where $f_i(x)$ are the expert logits and $g_i(x)$ are normalized routing weights. Training-time noise on the gate outputs encourages exploration and prevents premature specialization collapse.

To analyse routing behaviour, the training logs track expert utilization entropy, specialization, and per-class performance, confirming that different experts are preferentially used for different subsets of classes.

4.3. Training Configuration and Optimization

4.3.1. Common settings

Across all experiments:

- **Batch size:** 128 (unless explicitly overridden).
- **Epochs:** 50 for homogeneous and AlexNet MoEs; up to 200 for the heterogeneous MoE.
- **Gradient clipping:** Max-norm of 1 or 3 depending on the model to stabilize training.
- **Metrics:** Top-1 CIFAR-10 test accuracy, expert utilization entropy, and simple diversity indicators for routing behaviour.
- **Seeds:** Each configuration is trained with three random seeds; mean accuracy is reported, and the best single-seed model is highlighted.

4.3.2. Optimizers and schedules

Optimizer and scheduling choices differ slightly across MoE families:

- **Homogeneous MoEs (Exp. 1):** Stochastic Gradient Descent with momentum 0.9 and weight decay 1×10^{-4}. The learning rate starts at 0.01 and is decayed according to a standard schedule (step-based or cosine, depending on variant).
- **AlexNet MoEs (Exp. 2):** Also trained with SGD (momentum 0.9, weight decay 1×10^{-4}), reusing the same base schedule as the homogeneous family. The diversity regularizer is tuned to avoid both expert collapse and excessive randomness.
- **Heterogeneous MoE (Exp. 3):** AdamW is used with differential learning rates for gate and experts. An initial learning rate of 0.01 is warmed up and then decayed via cosine annealing. The heterogeneous model additionally uses mixup, label smoothing with $\epsilon = 0.1$, and slightly stronger gradient clipping to stabilize long-horizon training up to 200 epochs.

4.4. Extended Results on CIFAR-10

Table 4 summarizes the key MoE models and single-backbone baselines on CIFAR-10, including both final and peak accuracies and the epoch at which the peak is reached.

The best homogeneous MoE (MoEv7) attains test accuracy of approximately 93.9%, while the heterogeneous MoE peaks at about 93.1% and surpasses each individual expert backbone. At the same time, the AlexNet-based MoE family provides a controlled setting to study expert specialization and the effect of additional diversity regularization.

4.5. Routing Behaviour and Architectural Insights

The diagnostics collected during training provide qualitative insights that complement the quantitative CIFAR-10 results:

- **Homogeneous MoEs:** Careful tuning of the load-balancing term and gating noise is crucial. Under-regularized gates collapse to one or two dominant experts, while over-regularization harms accuracy; MoEv7 finds a sweet spot, exhibiting both high accuracy and relatively balanced expert utilization.

Table 4. Representative Mixture-of-Experts models and single-backbone baselines on CIFAR-10. "Final" refers to accuracy after the last training epoch; "Peak" is the best accuracy observed during training, with the corresponding epoch.

Model	**Final acc.**	**Peak acc.**	**Best epoch**
Homogeneous MoE (Exp. 1)			
MoEv7	0.9379	0.9390	49
AlexNet Experts (Exp. 2)			
MoEv9-AlexNetv3	0.8652	0.8656	49
Heterogeneous MoE (Exp. 3)			
MoE-hetero4-Alex-Dense-Air-Bag	0.9026	0.9313	153
Single-model baselines			
AlexNet	0.8654	0.8675	44
DenseNet	0.8684	0.8792	45
AirNext	0.7720	0.7769	16
BagNet	0.7575	0.8273	19

Table 5. Experiment 1 (Homogeneous MoE): final and peak accuracies.

Model	**Final**	**Ep.**	**Max**	**Best Ep.**
MoE-0707	0.4915	50	0.4999	27
MoEv2	0.6954	15	0.7697	5
MoEv3	0.2358	3	0.2358	3
MoEv4	0.9237	50	0.9281	39
MoEv5	0.8608	50	0.9207	46
MoEv6	0.8750	50	0.8750	50
MoEv7	**0.9379**	**50**	**0.9390**	**49**
MoEv8	0.8118	50	0.8166	49

Table 6. Experiment 2 (AlexNet experts): final and peak accuracies.

Model	**Final**	**Ep.**	**Max**	**Best Ep.**
MoEv9-AlexNet	0.7595	50	0.8210	26
MoEv9-AlexNetv2	0.8138	50	0.8138	50
MoEv9-AlexNetv3	**0.8652**	**50**	**0.8656**	**49**
MoEv9-AlexNetv4	0.8421	50	0.8467	49

- **AlexNet experts:** Adding the diversity term encourages different AlexNet experts to specialize on different subsets of classes or input modes, which improves robustness and leads to the strongest AlexNet-based configuration (MoEv9-AlexNetv3).

Table 7. Experiment 3 (heterogeneous experts and single-model baselines).

Model	**Final**	**Ep.**	**Max**	**Best Ep.**
MoE-hetero4-Alex-Dense-Air-Bag	**0.9026**	**200**	**0.9313**	**153**
AlexNet	0.8654	50	0.8675	44
DenseNet	0.8684	50	0.8792	45
AirNext	0.7720	50	0.7769	16
BagNet	0.7575	50	0.8273	19

- **Heterogeneous MoE:** Soft routing over AlexNet, DenseNet, AirNet, and BagNet yields complementary behaviour: stronger backbones (AlexNet, DenseNet) dominate overall accuracy, while weaker experts (AirNet, BagNet) still contribute on specific classes or image structures. This explains why the heterogeneous MoE exceeds each single backbone while retaining stable long-horizon training.

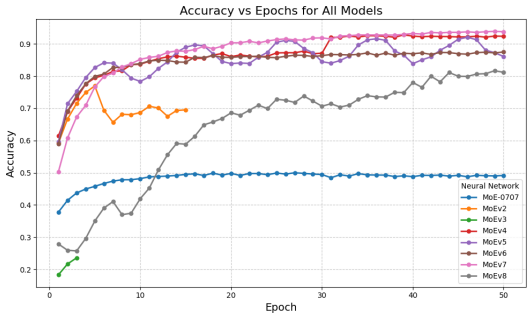

Figure 3. Experiment 1 training curves (accuracy vs. epochs).

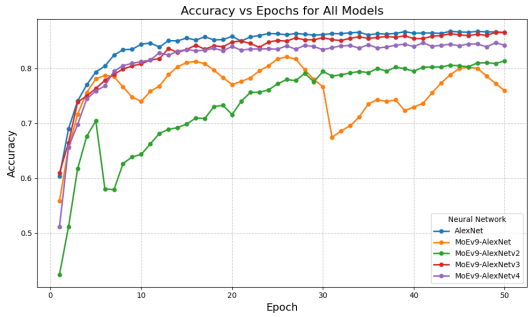

Figure 4. Experiment 2 training curves (accuracy vs. epochs).

5. AST Pipeline Details

5.1. Stage 1: Static Graph Construction via Symbolic Tracing

The foundation of our analysis is a high-fidelity computation graph of the target neural network. This is achieved not by parsing the Python source text directly, but by analyzing a live, instantiated model object using the `torch.fx` symbolic tracing library.

- **Symbolic Execution:** The process begins by feeding the model instance into a symbolic tracer. The tracer executes the model's `forward` method abstractly; instead of performing numerical tensor computations, it records the sequence of operations as a directed acyclic graph (`fx.Graph`). Each node in the graph represents a specific operation (e.g., a module call, a function call, or an I/O operation), and the edges represent the flow of data (tensors) between them.
- **Custom Introspective Tracer:** A key challenge with standard `torch.fx` is its tendency to treat container modules like `nn.Sequential` as opaque, "leaf" nodes, hiding their internal structure. To overcome this, we implemented a custom `LeafTracer` (from `mutator/utils/fx_graph_utils.py`). This tracer is specifically configured to introspect `nn.Sequential` containers, recursively tracing their sub-modules. The result is a fully "flattened" and detailed graph where every individual layer, even those nested deep within containers, is represented as an explicit `call_module` node. This level of granularity is essential for accurately tracking dependencies and propagating dimensional changes.

5.2. Stage 2: Bridging the Graph to Source Code

The `fx.Graph` is an abstract representation, but it is disconnected from the editable source code. To bridge this gap, we developed the `ModuleSourceTracer` (`mutator/utils/source_tracer.py`), a critical component that maps abstract graph nodes to their concrete definitions in the source file.

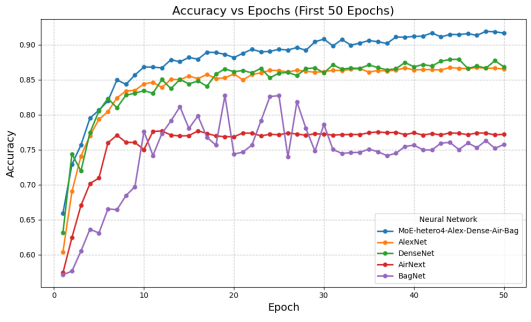

Figure 5. Experiment 3 training curves (first 50 epochs shown).

- **Dynamic `__init__` Patching:** The tracer operates by dynamically patching the `__init__` constructor of core PyTorch modules (`nn.Conv2d`, `nn.Linear`, etc.) before the model itself is instantiated.
- **Runtime Stack Inspection:** When the model's constructor is executed, every line of code that instantiates a patched layer (e.g., `self.conv1 = nn.Conv2d(...)`) triggers the custom logic. This logic uses Python's native `inspect` module to perform a stack walk. By examining the call stack frames, it precisely identifies the filename, line number, and column offset of the code that initiated the instantiation.
- **Source Map Generation:** This location information is captured and stored in a dictionary we term the *source_map*. This map provides a one-to-one correspondence between a module's unique, fully-qualified name in the `fx.Graph` (e.g., `'layer1.0.conv1'`) and its physical location in the source code file. This map perform targeted code modifications in later stage.

5.3. Stage 3: Mutation Planning and Propagation

With the graph and source map, the `ModelPlanner` (`mutator/planning/base_planner.py`) can devise structurally-sound mutations. For channel dimension edits, which are the focus of our static mutations, the process is as follows:

- **Target Selection:** A mutable layer (e.g., a `nn.Conv2d` or `nn.Linear` module) is selected as the mutation target.
- **Dependency Analysis:** The planner uses the `fx.Graph` to identify all direct and indirect dependencies. For a forward channel propagation, it finds all consumer nodes that take the target layer's output as input. For layers that are part of a residual connection (identified by tracking consumers of `torch.add` operations), they are grouped together to ensure their output dimensions are mutated consistently.
- **Plan Generation:** A mutation plan is generated. This is a dictionary detailing all required code changes. For instance, if the `out_channels` of `conv1` is changed from 64 to 128, the plan will also include a corresponding change to the `in_channels` of its consumer, `conv2`, from 64 to 128. The plan also includes the source locations for each required edit, retrieved from the *source_map*.
- **Symbolic Expressions:** The planner can generate mutations as either fixed integer values or as symbolic expressions (e.g., changing a dimension to `'in_channels * 2'`). This allows for the creation of more flexible and scalable architectural motifs.

5.4. Stage 4: Automated Code Refactoring via AST Manipulation

The execution of the mutation plan is handled by the `CodeMutator` (`mutator/execution/code_mutator.py`), which performs modifications on the source code using Abstract Syntax Trees.

- **Parsing:** The original Python source code string is first parsed into an AST using Python's built-in `ast` library. This tree is a structured representation of the code.
- **Targeted Traversal:** The mutator traverses this tree. Using the precise line and column numbers from the mutation plan, it locates the exact `ast.Call` node that corresponds to the instantiation of the target layer.
- **Node Modification:** Once the node is found, its arguments are modified. For a keyword argument like `out_channels=64`, the mutator finds the corresponding `ast.keyword` object and replaces its value (an `ast.Constant` node) with a new node representing the new integer value. For symbolic mutations, it replaces the value with a parsed AST snippet of the expression (e.g., an `ast.BinOp` node).
- **Unparsing:** After all modifications in the plan have been applied to the AST, the `ast.unparse()` function is called. This converts the modified tree back into a syntactically correct Python code string, which now contains the mutated architecture.

5.5. Stage 5: Post-Mutation Verification

To ensure the integrity of the mutated architecture, a final verification step is performed, as seen in the `run_single_mutation` function of `orchestrator.py`.

- **Instantiation and Forward Pass:** The newly generated source code string is dynamically loaded as a module, and the model is instantiated. A dummy input tensor, typically of shape `(2, 3, 224, 224)` to emulate a batch of standard vision inputs, is created and passed through the model's `forward` method. A successful forward pass confirms that all layer dimensions are consistent and there are no shape-related runtime errors.
- **Backward Pass:** To further validate trainability, a simple backward pass is also conducted. A loss is computed from the output (e.g., `output.sum()`), and `loss.backward()` is called. This ensures that gradients can flow through the entire network, a prerequisite for successful training. Any model that fails this verification step is discarded.

6. Reinforcement Learning Pipeline Details

6.1. Dataset Construction and Masking Pipeline

We construct a reinforcement-learning dataset using the LEMUR API, which provides diverse PyTorch convolutional neural network implementations for image classification. For each retrieved model, we programmatically isolate the layer-definition block (e.g., convolutional layers, pooling layers, classifier layers) and replace it with a placeholder mask. Given an original model:

```
class Net(nn.Module):
    def __init__(self, ...):
        super().__init__()
        self.conv1 = nn.Conv2d(...)
        self.conv2 = nn.Conv2d(...)
        self.fc = nn.Linear(...)
```

The architecture region is removed:

```
class Net(nn.Module):
    def __init__(self, ...):
        super().__init__()
        # [MASKED LAYER]
```

The resulting {masked model, original model} pairs serve the purposes: Reinforcement-learning prompts where the agent must recover the architecture

Each masked model is embedded into a completion prompt:

```
Below is a partial PyTorch neural network
    working on cifar-10 with masked layers:

{masked_code}

Please generate the complete PyTorch neural
     network code by replacing the [MASKED
    LAYER] placeholders.

Requirements:

1. Output the complete Python/PyTorch code
    wrapped in ```python``` code blocks

2. Keep the same class structure (Net class
    ) and method signatures exactly

3. Replace each '# [MASKED LAYER]' with
    appropriate PyTorch layer definitions

4. Keep train_setup, learn, forward methods
     exactly the same
```

```
5. supported_hyperparameters function must
    remain exactly the same outside the Net
    class, only have ['lr', 'momentum', '
    dropout']

6. Make sure the Net.__init__ method
    signature matches: __init__(self,
    in_shape: tuple, out_shape: tuple, prm:
    dict, device: torch.device)

7. prm only have ['lr', 'momentum', 'batch'
    , 'transform', 'epoch'], use other prm
    key will be strictly penalized

8. Try to generate more different network
    structures, try to use different layers
    provided by torch.

Now generate the complete working code
    wrapped in ```python``` blocks:
```

6.2. Model and Training Framework

We fine-tune the **DeepSeek-Coder-6.7B-Instruct** model due to its strong instruction following and code synthesis capabilities. Reinforcement learning is performed using `GRPOTrainer` in `HuggingFace TRL` lib, which avoids the need for a separate critic network, thereby increasing the efficiency of the structure generation task.

The training configuration includes:

- 4-bit quantization and LoRA adapters for memory-efficient fine-tuning
- Automatic token-length filtering to maintain model context limits
- Multi-step RL fine-tuning over masked CNN architecture samples
- Periodic evaluation and logging for reproducibility

6.3. Reward Function Design

Generated completions are parsed to extract the pure code block. We adopt a hierarchical reward mechanism combining syntax correctness, runtime feasibility, and downstream task performance.

1. **Static & runtime validation**
   - Code compiles without syntax error
   - Class instantiates successfully
   - Forward pass on dummy input completes
2. **Neural performance evaluation**
   - Train on CIFAR-10 for one mini-epoch
   - Compute accuracy using `check_nn` API

Reward scoring:

If execution fails (excluding accuracy exceptions), reward $= -1$. This encourages models that are both syntactically correct and empirically performant. It can also accept a baseline metric, improvements above this baseline will earn more rewards.

Criterion	Reward
Successful compilation / model builds	+0.2
Forward execution passes	+0.2
Invalid model	-1.0
Accuracy improvement over baseline	$+1.0 \times \Delta\text{Acc}$
Training loop runs	+0.2

6.4. Reinforcement Learning Loop

The RL loop proceeds as follows:

1. Sample a masked CNN skeleton from the dataset
2. Generate the completion using DeepSeek-Coder
3. Extract executable code and evaluate performance
4. Compute reward from compilation, execution, and accuracy
5. Update policy using `GRPOTrainer`

This creates a closed learning loop that improves model-generated architectures autonomously.

6.5. System Integration and Logging

- All generations and evaluation metrics are logged via a custom `SimpleCodeLogger`
- Generated models are stored in `out/nngpt/llm/epoch`
- Evaluation traces and failures are recorded for debugging and replay
- Training outputs are serialized for reproducibility and ablation study

Overall, the system forms an iterative improvement pipeline linking

Code Generation $\rightarrow$ Execution $\rightarrow$ Reward $\rightarrow$ Model Update.

This design enables the model to discover and optimize novel CNN architectures guided directly by performance feedback.

7. Genetic Algorithm (NAS-GA) Implementation Details

7.1. Search Spaces and Architectural Encoding

The genetic algorithm operates over a parameterized representation of AlexNet-style CNNs that includes both architectural and training hyperparameters. Two search spaces are considered:

Basic Architecture search space. The Basic Architecture space keeps the overall macro-structure fixed and restricts search to a limited set of hyperparameters:

- convolutional filters, kernel sizes, and strides,
- learning rate, momentum, and dropout rate.

Architectures are encoded as vectors of these parameters. The convolutional stack and fully connected topology follow an AlexNet-style template, with the GA discovering, for example, when to use large initial kernels or very wide fully connected layers.

Block-type Variation search space. The Block-type Variation space extends the Basic search by also allowing block-level structural choices:

- pooling type (e.g., max, average, adaptive pooling),
- activation function,
- inclusion of batch normalization,
- optional skip layers,
- plus all hyperparameters from the Basic search: filters, kernels, strides, learning rate, momentum, and dropout rate.

This larger, more expressive search space enables the GA to modify not only numeric hyperparameters but also the fundamental building blocks and connectivity pattern of the network.

7.2. Genetic Algorithm Pipeline and Fitness Evaluation

The NAS-GA pipeline follows the standard evolutionary pattern as:

- **Initialization.** The population is initialized around a baseline AlexNet configuration. Each individual corresponds to a specific parameter vector in one of the two search spaces.
- **Selection, crossover, mutation.** At each generation, a GA engine applies selection, crossover, and mutation operators on the parameter vectors. Crossover mixes architectural traits (e.g., kernel sizes, pooling types), while mutation perturbs individual parameters (e.g., changing dropout rate or toggling batch normalization).
- **Limited-epoch fitness evaluation.** To maintain computational feasibility while generating thousands of architectures, each candidate's fitness is estimated via limited-epoch training on CIFAR-10. Validation accuracy after this short training is used as the fitness score during evolution.
- **Duplicate avoidance via checksum.** A checksum is computed over each architecture's description (parameter vector and macro-structure). Any candidate whose checksum has already been evaluated is discarded without retraining. This prevents duplicate testing and helps maintain diversity in the evolving population.
- **Logging and checkpointing.** The system logs every evaluated architecture, its fitness, and GA state, enabling robust reproducibility and allowing the evolutionary process to be resumed from checkpoints.

The champion architectures from both search spaces are subsequently re-evaluated under the standardized LEMUR 2 protocol, yielding the CIFAR-10 test accuracies.

7.3. Extended Results: Evolution Dynamics and Champion Architectures

7.3.1. Basic Architecture search

In the Basic Architecture search, the GA is run for 25 generations on CIFAR-10 with the constrained hyperparameter-only space.

- The initial best fitness is 61.00% in Generation 1 and improves steadily to 62.57% by Generation 5.
- The GA reaches peak fitness of **62.76%** in Generation 9, maintaining this level until the end of the run, indicating convergence.

The champion Basic Architecture model—reported attains **62.76%** CIFAR-10 test accuracy after one epoch of training under the LEMUR 2 protocol and exhibits the following evolved traits:

- a large 11×11 kernel in the first convolutional layer,
- very wide fully connected layers (e.g., 4096 / 2048 units),
- a high dropout rate of 0.6 to control overfitting.

These characteristics reflect how, under a restricted search space, the GA exploits large receptive fields and heavy regularization rather than structural diversification.

7.3.2. Block-type Variation search

In the Block-type Variation search, the GA runs for 50 generations with the richer space including pooling, activation, batch normalization, and skip connections, in addition to the hyperparameters above.

- The best fitness jumps from 68.60% in Generation 1 to 74.94% in Generation 2, indicating rapid early gains.
- Fitness then improves more gradually, reaching a peak of **80.04%** in Generation 38.
- The final 12 generations show stable fitness, suggesting convergence to a highly competitive architecture.

The champion Block-type Variation model achieves **80.04%** CIFAR-10 test accuracy after one epoch in the LEMUR 2 evaluation. Structurally, it differs markedly from the Basic champion:

- it omits the large 11×11 initial convolution, instead relying on efficient 3×3 kernels,
- it introduces **batch normalization**, improving optimization stability and generalization,
- it evolves a **mixed-pooling strategy**, combining adaptive, max, and average pooling,
- it may skip or modify early convolutional blocks (e.g., skipping `conv1`) to reallocate capacity downstream.

Allowing block-level structural choices provides a substantial performance gain (80.04% vs. 62.76%) over a purely hyperparameter-focused search, even under the same limited-epoch fitness protocol.

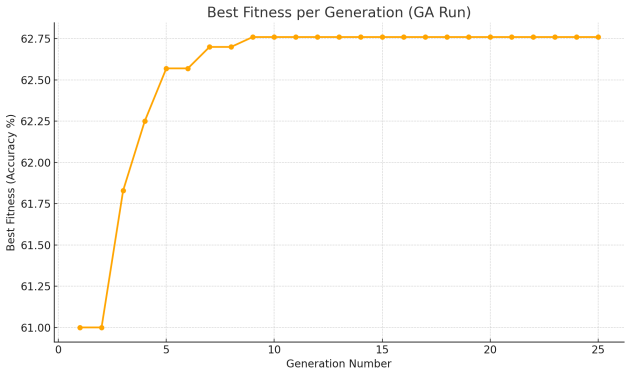

Figure 6. Evolutionary performance based on Basic Architecture

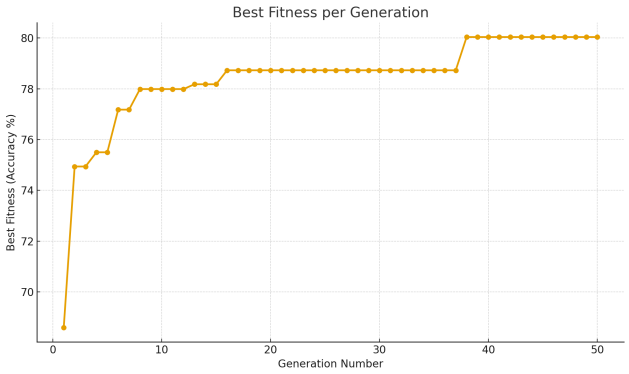

Figure 7. Evolutionary performance based on Block-type Variation

8. Fractal-Inspired Computational Architectures Implementation Details

8.1. System Overview and Goals

The FractalNet framework is designed to:

- automatically generate a large number of structurally diverse CNNs via fractal templates;
- train and evaluate these models under a unified protocol on CIFAR-10;
- export the resulting models and statistics into LEMUR 2 as an additional family of image-classification architectures.

The system ultimately produces and trains **over 1,200** unique convolutional models, each controlled by a small set of fractal parameters: the *fractal depth* N (recursion level) and the number of parallel *columns* (`num_columns`). Architectures and metrics are stored with a standardized naming convention, using the prefix: `FractalNet-`

for all CIFAR-10 FractalNet entries.

8.2. Template-Driven Fractal Generation

The FractalNet subsystem follows the generator–template–runner pattern already adopted elsewhere in the LEMUR ecosystem, but specialized to fractal architectures.

8.2.1. Generator

The **Generator** module (see `AlterNNFN.py`) enumerates candidate configurations in a structured fashion:

- It samples discrete values for fractal depth N and column width `num_columns` within predefined ranges.
- For each configuration, it permutes building blocks consisting of convolution, normalization, activation, and dropout layers, yielding a blueprint that specifies the ordering of these layers within each fractal unit.
- It writes out architecture descriptors that can be consumed by the template module.

This combinatorial but template-constrained sampling defines the effective search space: recursive deep/wide CNNs whose internal blocks differ in layer ordering and presence of normalization / dropout.

8.2.2. Fractal Template

The **Template** module (`fractal_template.py`) implements the core fractal construction:

- It instantiates self-similar, recursively defined units that can branch into multiple columns and recombine later, following the principles of FractalNet-style architectures.
- Within each unit, convolutional, batch normalization, activation, and dropout layers are assembled according to the blueprint produced by the Generator.
- The recursion level N determines how many times the pattern is expanded; `num_columns` controls the number of parallel pathways combined at each fractal level.

This approach yields multi-column networks that grow in both depth and width in a balanced way: deeper models are obtained by increasing N, while wider ensembles arise from increasing `num_columns`, all under a single template.

8.2.3. Runner and Integration with LEMUR

The **Runner** (`NNAlterFractalNet.py`) and associated evaluation script `ab/gpt/NNEval.py` manage the end-to-end experiment loop:

- For each configuration, the Runner instantiates the PyTorch model from the fractal template, sets up data loaders, and applies standardized training parameters.
- It performs automatic checkpointing, metric logging, and error handling for each architecture.
- All trained models, logs, and statistics are written to a dedicated `new_lemur/` directory, from which entries are imported into the LEMUR 2 database with the `FractalNet-` prefix.

The overall pipeline is designed as a closed loop of generation, construction, training, and assessment, enabling

large-scale exploration with minimal manual intervention.

8.3. LLM Configuration and Control

The fractal framework is coupled with LLM-based configuration modules to maintain consistency with the broader LEMUR stack:

- **LLM backend:** A configuration under `conf/llm` integrates the `DeepSeek-R1-Distill-Qwen-7B` model as the primary LLM used in this component.
- **Prompt loading:** The prompt for any LLM-driven control or meta-configuration is managed by `conf/prompt`.
- **Evaluation script:** `ab/gpt/NNEval.py` is responsible for training and evaluating models under the same logging and configuration regime used by other LEMUR 2 generators.

In the current experiments, the fractal architecture search itself is driven by the template-based generator, while the LLM integration provides a uniform entry point for future LLM-controlled configuration and orchestration.

8.4. Training and Evaluation Protocol

8.4.1. Dataset and Preprocessing

All FractalNet variants are trained and evaluated on **CIFAR-10**, using the standard split:

- 50,000 training images and 10,000 held-out test images (used as validation in this context).
- All images are RGB, 32×32 pixels.

Preprocessing and augmentation are deliberately simple and fixed across all models:

- Per-channel normalization using dataset statistics.
- Random horizontal flip as the primary augmentation (`norm + flip`).

8.4.2. Training Hyperparameters

Each generated architecture is trained for a short, uniform schedule:

- **Optimizer:** Stochastic Gradient Descent (SGD).
- **Learning rate:** 0.01.
- **Batch size:** 16.
- **Momentum:** 0.9.
- **Dropout:** 0.2 (applied in the fractal blocks according to the sampled blueprint).
- **Epochs:** 5 per model.

Training is implemented in PyTorch with two key efficiency techniques:

- **Automatic Mixed Precision (AMP):** exploits half-precision arithmetic to reduce computation time.
- **Gradient checkpointing:** selectively recomputes intermediate activations to reduce GPU memory consumption.

Together, these techniques allow the system to handle large fractal configurations and still train **more than 1,200** models within a reasonable memory budget.

8.4.3. Evaluation Stages

The evaluation protocol is standardized and identical for every FractalNet variant:

1. **Model verification:** The automatically generated Python class is imported and instantiated. This checks syntactic correctness and compatibility with the PyTorch training framework.
2. **Training and checkpointing:** The model is trained for 5 epochs using SGD, AMP, and gradient checkpointing. Checkpoints and intermediate metrics are recorded.
3. **Performance logging:** After each epoch, the Runner logs validation accuracy, loss, GPU memory usage, and training time. These metrics are stored alongside configuration metadata in the `new_lemur/` directory.

Architectures that fail instantiation or training are marked as unsuccessful and excluded from downstream analysis, but their error states are still recorded.

8.5. Extended Results and Fractal Hyperparameter Effects

8.5.1. Aggregate statistics

Across all 1,200+ runs, the framework provides the following aggregate statistics:

- **Average validation accuracy:** approximately 83%.
- **Top validation accuracy:** approximately 89–90%.
- **Mean training time:** about 5 minutes per epoch.
- **Typical GPU memory usage:** 4–5 GB.
- **Successful training rate:** around 97%.

These numbers indicate that the vast majority of fractal architectures converge reliably under the 5-epoch protocol and that the pipeline remains computationally tractable.

8.5.2. Baseline vs. best FractalNet configuration

A baseline FractalNet configuration—representing an early manual design—achieves a validation accuracy of **72.2%** on CIFAR-10 under the same 5-epoch training budget. After large-scale automated generation and evaluation of 1,200 variants, the best-performing configuration reaches **80.18%** validation accuracy, an improvement of roughly 8 percentage points.

This gain is achieved by systematically adjusting:

- fractal depth N, to balance depth and gradient flow;
- column width (`num_columns`), to exploit multi-path feature reuse;
- the ordering and presence of normalization, activation, and dropout in each fractal unit.

Overall, the search discovers architectures that reuse features more effectively, distribute parameters more evenly across depth and width, and converge faster than the baseline.

8.5.3. Effect of fractal depth and column width

Analysis of the final metrics reveals clear trends:

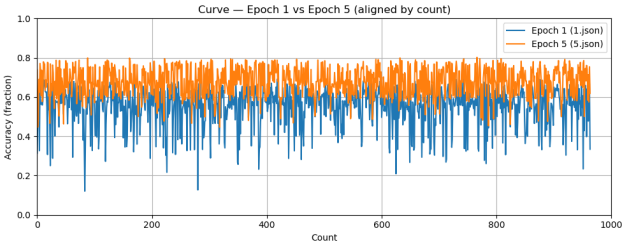

Figure 8. Validation accuracies of all FractalNet models over the first and fifth epochs, showing both initial learning dynamics and final convergence stability.

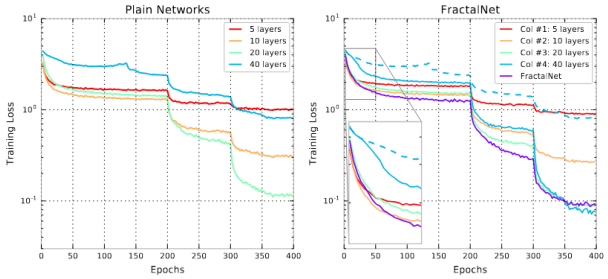

Figure 9. Comparison of training loss progression across epochs between Plain Networks and FractalNet.

- **Moderate configurations work best.** Networks with **fractal depth** $N = 3$–4 and **column width** **`num_columns`** $= 3$–4 consistently achieve the highest validation accuracies, while maintaining stable training.
- **Extreme configurations are unstable.** Architectures with very high depth or width (e.g., $N \geq 5$ or `num_columns` ≥ 7) often fail to train due to GPU memory exhaustion or gradient instability and are frequently aborted.

Validation accuracy distributions after epoch 1 and epoch 5 further show:

- strong early learning signals for many architectures;
- consistent improvement between epochs 1 and 5, corroborating stable convergence under the shared protocol.

8.5.4. Comparison with plain networks and NAS-style baselines

The FractalNet variants are compared to:

- a standard CNN baseline;
- representative NAS-generated models (outside the fractal family).

Plots of training loss versus epochs for FractalNet and plain networks show that:

- FractalNet exhibits a smoother, more consistent loss decrease;
- it converges faster and to lower training loss even in the early epochs;
- the full fractal model outperforms its individual columns considered as standalone networks, consistent with the ensemble-like effect of fractal connectivity.

9. Retrieval-Augmented Block Extraction Details

9.1. High-Level Design

Architecturally, NN-RAG comprises five co-operating components: **BlockDiscovery**, **BlockExtractor**, **FileIndexStore**, **BlockValidator**, and **RepoCache**. Together, these provide a reproducible path from raw repositories to an indexed library of executable blocks.

9.2. Corpus Construction and Block Discovery

NN-RAG operates on a curated corpus of PyTorch repositories that includes large hubs such as `timm`, `torchvision`, and `transformers`, as well as a long tail of smaller, specialized projects. The discovery process proceeds as follows:

- **Repository caching (RepoCache).** Repositories are cloned or downloaded once and stored in a local cache. Subsequent runs reuse this cache, enabling incremental updates as upstream code changes.
- **Block discovery (BlockDiscovery).** The system scans Python files within each cached repository, searching for non-abstract classes that:
  1. inherit (directly or indirectly) from `torch.nn.Module`, and
  2. define a `forward()` method.

  Each such class is treated as a candidate "block" (e.g., `SwinTransformerBlock`, `DropPath`, `PatchEmbed`).
- **Concrete-syntax parsing.** Source files are round-tripped through LibCST, which preserves concrete syntax, formatting, and comments. This ensures that extracted modules remain faithful to the original implementations, including stylistic details that may aid comprehension.

Parse artifacts, symbol tables, and import relations are recorded for each file and stored in an index for later use by the extractor and resolver.

9.3. Indexing and Dependency Resolution

9.3.1. FileIndexStore: Content-Addressed Indexing

The **FileIndexStore** manages an SQLite-backed index that records:

- file paths and repository origins,
- LibCST parse trees or derived structures,
- symbol definitions and their scopes,
- import statements and their targets.

Each file is identified via a SHA-1 hash of its contents, enabling content-addressed storage. This design allows O(1) incremental rebuilds: when code changes upstream,

only files whose hash differs need to be re-parsed, keeping NN-RAG responsive even for large corpora.

9.3.2. Scope-Sensitive Dependency Closure

For each discovered block, NN-RAG computes the minimal transitive closure of its dependencies without executing repository code:

- **Scope modelling.** A resolver simulates Python's LEGB (Local–Enclosing–Global–Builtins) scoping rules to determine where names are defined. This includes:
  - local definitions inside the block's file,
  - imported names from other modules,
  - module-level constants and helper functions,
  - class-level attributes required by `forward()`.
- **Transitive closure.** Starting from the block's class definition and its `forward()` body, the resolver follows symbol references to collect all required auxiliary definitions (e.g., helper layers, utility functions). Only definitions actually referenced by the block are included, avoiding unnecessary bloat.
- **Topological ordering.** The collected definitions are organized into a directed graph based on their dependencies. A topological sort yields an order that satisfies definition-before-use, ensuring that the final assembled module respects Python's execution model.

The result is a minimal, scope-closed set of definitions and imports sufficient to execute the block in isolation.

9.4. Module Assembly and Validation

9.4.1. BlockExtractor: Concrete-Syntax Assembly

The **BlockExtractor** uses LibCST to assemble the final module:

- Original import statements (including aliases) are preserved where possible, so that the extracted module closely mirrors its source context (e.g., `import torch.nn as nn` remains intact).
- The topologically ordered definitions are inserted into a fresh module, with comments and formatting retained from the original files.
- The block class itself is placed at the end of the definition list, after all its dependencies, to honor definition-before-use constraints.

By performing a LibCST round-trip rather than simple string slicing, NN-RAG maintains concrete-syntax fidelity and reproducible formatting.

9.4.2. BlockValidator: Multi-Stage Safety Checks

Each assembled module passes through a conservative validation pipeline:

1. **AST parsing.** Python's `ast` module parses the generated file to ensure syntactic correctness.
2. **Bytecode compilation.** The module is compiled into bytecode using `compile()`, which surfaces syntax errors that may not appear during parsing alone.
3. **Sandboxed execution.** The module is imported and executed in a sandboxed environment. This stage captures:
   - unresolved symbol references,
   - import-time errors (e.g., missing third-party libraries),
   - gross side effects that prevent successful import.

Only modules that pass all stages are considered valid and are integrated into LEMUR 2. Failures are logged with the corresponding block name and error type to support future diagnostics and corpus refinement.

9.5. Block Library Statistics

9.5.1. Coverage and Success Rate

NN-RAG's evaluation uses a curated set of **1,289** target neural-network block names drawn from widely used PyTorch implementations. After extraction and validation, **941** modules (**73.0%**) are admitted as fully executable, self-contained components. The remaining blocks typically fail due to unresolved external dependencies or import-time issues that cannot be resolved safely without manual intervention.

These 941 modules form the NN-RAG block library, and architectures that incorporate them are tagged with the prefix `rag-` in the LEMUR 2 database.

9.5.2. Category Distribution

The target blocks span a broad range of modelling primitives across vision and language. Table 8 summarizes the distribution over semantic categories.

At the repository level, NN-RAG's index is deliberately not restricted to a few marquee libraries: while large hubs (`timm`, `torchvision`, `transformers`) contribute many blocks, a sizable long tail of smaller projects collectively supplies a substantial fraction of reusable components. This breadth is important for assembling hybrid architectures that draw on diverse design motifs.

9.6. Integration and Downstream Use

9.6.1. Integration into the LEMUR Corpus

Each validated NN-RAG module is registered in LEMUR 2 with:

- a unique identifier and its original fully qualified name (module path + class name),
- a pointer to its source repository and commit hash,
- category labels (e.g., attention, convolution, loss),
- a tag indicating its origin (`rag-`).

These blocks can then be imported by other LEMUR components, including LLM-driven architecture generators, AST mutation pipelines, and hand-written architectures. When a generated architecture uses one or more NN-RAG modules, the corresponding runs are also tagged with

Table 8. Distribution of NN-RAG target blocks across categories, with representative examples.

Category	Count	Examples
Attention Mechanisms	180	`MultiHeadAttention`, `SelfAttention`
Convolutional Layers	220	`Conv2d`, `DeformConv`
Transformer Blocks	150	`BertLayer`, `SwinTransformerBlock`
Pooling Operations	110	`MaxPool2d`, `AdaptiveAvgPool2d`
Normalization Techniques	100	`BatchNorm2d`, `LayerNorm`
Loss Functions	90	`FocalLoss`, `DiceLoss`
Network Architectures	150	`ResNet`, `VGG`, `EfficientNet`
Utility Modules	289	`DropPath`, `PatchEmbed`
Total	**1,289**	

`rag-`, enabling downstream analysis of how often retrieved blocks appear in high-performing models.

9.6.2. Retrieval-Augmented Design Workflows

In addition to building the static library, NN-RAG supports retrieval-augmented workflows:

- When an LLM is asked to synthesize a new architecture, NN-RAG's index can be queried (e.g., via code search or semantic search over block names and docstrings) to retrieve candidate blocks (such as attention layers, specialized convolutions, or loss functions).
- These retrieved blocks are then treated as high-quality exemplars that the LLM can assemble or adapt, reducing the need for fully free-form code generation and grounding designs in verified implementations.

This mechanism was used to derive and reuse architectural motifs from over 900 PyTorch modules during LLM-guided architecture generation, enriching LEMUR 2 with deep models that explicitly incorporate NN-RAG components.

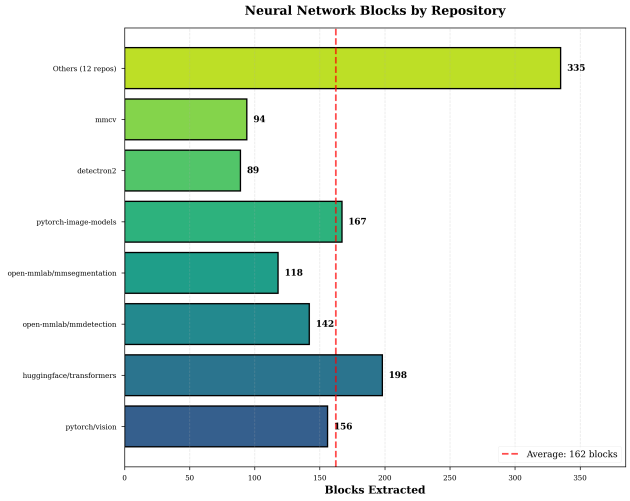

Figure 10. Repository-level distribution of extracted blocks: prominent hubs complemented by a sizable long tail of smaller projects

10. Few-Shot Architecture Prompting and Hash Validation Details

10.1. LLM Configuration

All `alt-nn` architectures are generated with LLM-driven code synthesis:

- **Base model.** We use **DeepSeek Coder 7B**, a code-focused LLM trained on approximately 2T tokens with strong coverage of Python scientific libraries (including PyTorch), as the backbone generator.
- **LoRA fine-tuning.** The model is adapted to LEMUR using Low-Rank Adaptation (LoRA) with rank $r = 32$ and $\alpha = 32$, yielding $\sim$ 35M trainable parameters ($\sim 0.5\%$ of the base model). Fine-tuning is performed on LEMUR's PyTorch models and associated metadata to specialize the LLM for neural-network code generation.
- **Decoding hyperparameters.** For all FSAP runs we use temperature $= 0.6$, top-$k = 50$, top-$p = 0.95$, and a maximum generation length of 65,536 tokens. This configuration was chosen empirically to balance diversity with syntactic and API correctness.

Generated architectures are integrated into LEMUR as standard `Net` classes and tagged with the prefix `alt-nn`, with suffixes `alt-nn1` through `alt-nn6` denoting the number of supporting examples, n, used in the prompt.

10.2. Training and Evaluation Protocol

10.2.1. Training configuration

All `alt-nn` models are trained under a rapid validation regime to maximize coverage over architectures:

- **Epochs:** 1 epoch per model.
- **Optimizer:** SGD with momentum.
- **Batch size:** dataset-dependent, typically between 64 and 4096 examples per batch.
- **Metric:** top-1 accuracy on the held-out validation/test split.

This short training schedule is sufficient to obtain a consistent ranking of architectures under matched budgets and is aligned with the first-epoch analyses.

10.3. Extended Generation Statistics

Across all datasets and $n \in \{1, \ldots, 6\}$, FSAP generates **4,033** unique architectures that compile and train successfully (after hash-based deduplication). Table 9 shows the distribution of models per variant and evidences the context-overflow phenomenon at high n.

Table 9. Number of generated `alt-nn` architectures per FSAP setting. "Success rate" is relative to initial generation attempts for each n; the sharp drop at $n = 6$ reflects severe context overflow.

Variant	# Examples n	# Models	Success Rate
alt-nn1	1	3,394	100% (baseline)
alt-nn2	2	306	9.0%
alt-nn3	3	103	3.0%
alt-nn4	4	102	3.0%
alt-nn5	5	121	3.6%
alt-nn6	6	7	0.2%
Total	–	**4,033**	–

Generation success decreases dramatically as n increases: while $n = 1$ yields thousands of models, $n = 6$ produces only 7 valid architectures, corresponding to a 99.8% failure rate due to non-compiling or non-training code.

The prompt structure follows this template:

````
CREATE an IMPROVED neural network by
    combining the best elements
from these models:
MAIN MODEL
(current accuracy: {accuracy}%):
```python
{reference_architecture_code}
```

SUPPORTING MODEL 1
(accuracy: {addon_accuracy_1}%):
```python
{supporting_architecture_1_code}
```

SUPPORTING MODEL 2
(accuracy: {addon_accuracy_2}%):
```python
{supporting_architecture_2_code}
```

[... up to n supporting models ...]

IMPROVEMENT RULES - FOLLOW EXACTLY:
1. Class name: 'Net' (unchanged)
2. Methods: __init__, forward, train_setup(
    device),
   learn(data,target,device) - keep
      signatures
3. Include: supported_hyperparameters() ->
    ['lr', 'momentum']
4. Only standard PyTorch (no torchvision)
5. Works with 32x32 RGB images -> [
    num_classes] classes
IMPROVE by combining best features from all
    models above.
Provide COMPLETE improved model code:
````

10.4. Discussion and Practical Guidelines

- **Context overflow.** Very large prompts ($n \geq 4$) increase the probability of syntactic and semantic errors during generation, resulting in both lower accuracy and dramatically higher failure rates, with $n = 6$ effectively unusable under our settings.
- **Efficiency gains.** Whitespace-normalized hash validation provides an $\approx 100\times$ speedup over AST-based deduplication and prevents training of around 100 exact-duplicate models, saving hundreds of GPU hours with negligible computational overhead.

These observations yield concrete guidelines for LLM-based neural architecture generation in LEMUR 2: use $n = 3$ exemplars for most tasks, avoid $n > 3$ unless context length is carefully managed, and always employ lightweight hash-based deduplication ahead of training.

11. Data Transformation Details

11.1. Prompt based code generation

We carried out code generation using Olympic Coder 7B through various prompting approaches, including Zero-shot, Role Prompting, Step-Back Prompting, and Chain-of-thought Prompting methods. However, we soon noticed issues such as identical transform functions were often generated, and many outputs had syntax errors.

11.2. Brute Force Approach

After initial approaches yielded limited improvements, we adopted a manual method. First, we designed a script to automatically generate image transformation functions using the torchvision package from PyTorch. The available transforms were organized into a dictionary. Given a set of parameters, the script generated multiple files, each containing one, two, or three randomly selected transforms from the dictionary, in addition to fixed transforms: resize, to tensor, and normalization. The script permuted and cycled through various transform combinations to generate the files, assigning random parameters to each selected transform function. In total, 6,000 transforms were generated and evaluated-

2,000 for each case of using one, two, or three variable transforms.

11.3. Training and Testing

All data transformation functions were evaluated for the image classification task using a ResNet model with the CIFAR-10 dataset. The model was trained for one epoch with a batch size of 64, a learning rate of 0.01, a momentum of 0.9, and a dropout rate of 0.2.

11.4. Evaluation Metrics

The chain-of-thought method demonstrated the highest performance among prompt-based generation techniques. In this method, the large language model (LLM) was instructed to modify a given transform. Of the 100 generated transforms, 47 were found to be syntactically incorrect. The best accuracy achieved was 57.28%, comprising of RandomResizedCrop, ColorJitter, RandomHorizontalFlip, GaussianBlur, ToTensor, and Normalize transforms. In comparison, the brute-force approach generated 6,000 transforms. Using this method, the best accuracy achieved was 61.24% with the RandomPosterize, Resize, ToTensor, and Normalize transforms. Additionally, the data transformations with a single selected transform generally performed better than those with two or three transforms. The transforms with better accuracy than the current best-performing transform (accuracy greater than 56.11%) are included in the LEMUR dataset. All transform functions and their corresponding evaluations will serve as metadata and may be used to fine-tune large language models for generating more efficient transforms in future research.

Table 10. Comparison of evaluation metrics across different transform configurations.

Configuration	Mean Accuracy	Best Accuracy	95% Confidence Interval
Prompt generated	0.1040	0.5728	[0.0644, 0.1436]
1 transform selected	0.5256	**0.6124**	[0.5234, 0.5279]
2 transforms selected	0.4832	0.6071	[0.4801, 0.4863]
3 transforms selected	0.4401	0.5983	[0.4363, 0.4439]

```
import torchvision.transforms as transforms

def transform(norm):
    return transforms.Compose([
        transforms.RandomResizedCrop(256,
            scale=(0.8, 1.0), ratio=(0.9,
            1.1)),
        transforms.ColorJitter(brightness
            =0.2, contrast=0.2, saturation
            =0.2, hue=0.1),
        transforms.RandomHorizontalFlip(),
        transforms.GaussianBlur(kernel_size
            =3, sigma=(0.5, 1.5)),
        transforms.ToTensor(),
        transforms.Normalize(*norm)
    ])
```

Listing 1. Prompt generated transform with highest accuracy (57.28%)

```
import torch
import torchvision.transforms as transforms

def transform(norm):
    return transforms.Compose([
    transforms.RandomPosterize(bits=4, p
        =0.68),
    transforms.Resize((64,64)),
    transforms.ToTensor(),
    transforms.Normalize(*norm)
])
```

Listing 2. Brute force generated transform with highest accuracy (61.24%)

12. NN-VR: VR-Ready Neural Network Verifier Implementation Details

12.1. System Architecture and Workflow

12.1.1. From LEMUR to ONNX

On the LEMUR side, NN-VR operates on models that have already been trained and recorded in the LEMUR 2 database. For each selected model:

1. The model's PyTorch checkpoint, task metadata (e.g., dataset, input shape), and configuration are retrieved via the LEMUR API.
2. A Python-based export script converts the model to ONNX, attaching minimal metadata (input tensor shape, task name, model identifier) so that the Unity side can reconstruct the correct input pipeline.
3. Exported ONNX files are placed into a designated directory watched by NN-VR, together with a small JSON descriptor that links back to the original LEMUR run.

This stage prepares models in a framework-agnostic interchange format and ensures that each Unity run can be traced back to a specific LEMUR configuration.

12.1.2. Neural Network Parser

The Neural Network Parser is responsible for ingesting the ONNX exports and preparing them for Unity:

- It scans the export directory, loads ONNX graphs, and extracts structural information (layers, tensor shapes, and operator types) needed to configure Unity's Barracuda runtime.
- It checks for the presence of unsupported operators, records them in a pre-verification log, and marks models that are likely to fail import.

- For models that pass this static check, it generates a per-model Unity configuration asset (e.g., mapping ONNX inputs to Unity textures or compute buffers) so that inference can be triggered from a standard test scene.

12.1.3. Compatibility Verifier

The Compatibility Verifier runs inside Unity and performs runtime checks under VR-like conditions. It imports the ONNX file via Barracuda and executes a configurable battery of tests:

- **Operator and shader coverage.** It verifies that all ONNX operators are mapped to supported Barracuda kernels and that the underlying shader programs compile successfully on the target graphics API.
- **GPU memory profiling.** It runs inference at VR-relevant resolutions and records peak GPU memory usage, flagging models that exceed a configurable budget.
- **Unity version coverage.** The same checks can be repeated across multiple Unity versions (e.g., including 6000.0.42f1) to detect regressions or changed operator implementations.

Failures at this stage are logged with precise error messages (e.g., unsupported operator, shader compilation error, out-of-memory), which are later consolidated on the Python side.

12.1.4. Automated Porting System

The Automated Porting System orchestrates the end-to-end path from LEMUR to VR:

1. **Model selection.** It selects candidate models from the LEMUR corpus (possibly filtered by task or architecture family) and triggers the ONNX export stage.
2. **Unity project configuration.** It injects the ONNX files and associated configuration assets into a Unity project that contains a reusable verification scene. This scene includes a Barracuda-based inference pipeline and simple VR-compatible camera / rendering setup.
3. **Verification run.** The Unity project is launched in batch mode and runs through all selected models, executing the compatibility and performance checks.
4. **Result logging.** For each model, Unity writes a structured log file (e.g., JSON) summarizing import success, memory usage, latency, and numerical consistency. These logs are collected back on the Python side and attached to the corresponding LEMUR models as deployment metadata.

In effect, NN-VR turns VR deployment into a reproducible experiment: given a model identifier in LEMUR, a single command exports it, runs Unity-based verification, and records device-level metrics.

12.2. Verification Criteria and Metrics

Latency. For each model that successfully imports into Unity, NN-VR measures the per-frame inference time under a standard VR configuration. The critical threshold is an 11 ms budget per frame, corresponding to 90 Hz rendering. NN-VR records whether this budget is satisfied, along with the mean and variance of the measured inference times over multiple runs.

Memory overhead. NN-VR compares the memory footprint of a model running inside the Unity VR scene to its standalone (non-VR) inference footprint, focusing on GPU allocations. The verification log includes:

- baseline memory usage for an empty scene,
- incremental memory usage after loading and executing the model,
- the ratio between VR and standalone memory consumption.

For AirNet, the final configuration achieves $\leq 15\%$ additional memory overhead compared to standalone inference on the test hardware.

Numerical consistency. To ensure that deployment does not silently degrade model behaviour, NN-VR compares Unity/Barracuda outputs to reference PyTorch predictions for a fixed set of test inputs. The models are considered consistent if Unity outputs deviate by at most 1% from PyTorch outputs; NN-VR encodes this as a pass/fail flag for each verified model, together with the maximum observed relative deviation.

12.3. Evaluation

NN-VR has been applied to over 10,000 pre-trained models exported from LEMUR. For standard convolutional architectures:

- approximately 95% of models that reach the Unity import stage achieve full compatibility, including ONNX import, operator support under Barracuda, and adherence to memory constraints;
- for models that pass compatibility checks, NN-VR records whether they meet the 11 ms latency target and whether numerical discrepancies stay within the 1% tolerance.

These deployment-oriented measurements complement the training metrics already stored in LEMUR and enable systematic analysis of accuracy–latency–memory trade-offs for VR applications.

13. Statistics

The individual statistics figures can be found under here. Please note that due to automatic insertion of the statistics into the database with varying hyperparameters, the statistics given here are subject to change.

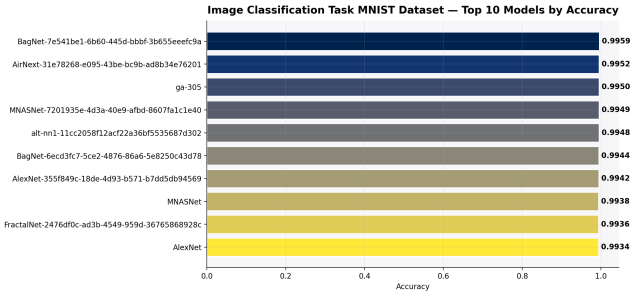

Figure 11. Top 10 image classification models ranked by their best recorded accuracy for the MNIST dataset.

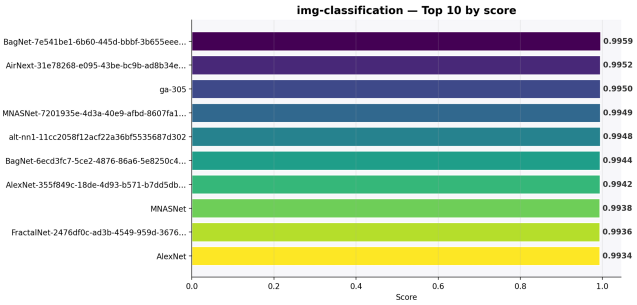

Figure 12. The top 10 models for image classification tasks ranked by best achieved accuracy.

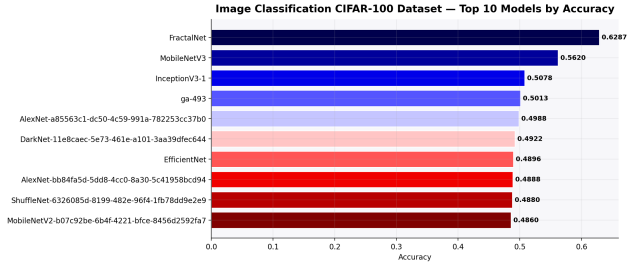

Figure 13. Top 10 image classification models ranked by their best recorded accuracy for the CIFAR-100 dataset.

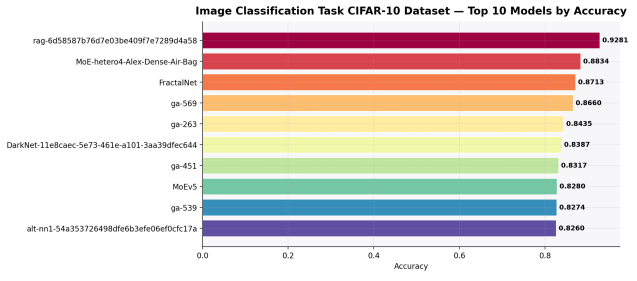

Figure 14. Top 10 image classification models ranked by their best recorded accuracy for the CIFAR-10 dataset.

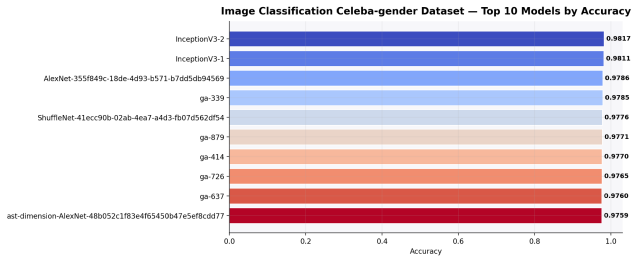

Figure 15. Top 10 image classification models ranked by their best recorded accuracy for the Celeba-gender dataset.

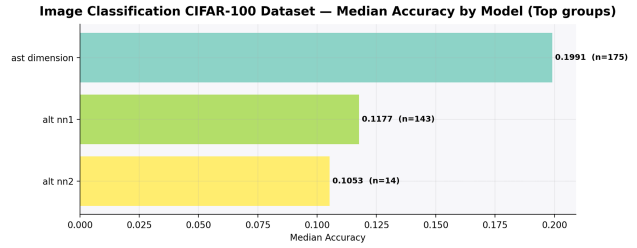

Figure 16. Median accuracy per model group for image classification tasks on the CIFAR-100 dataset.

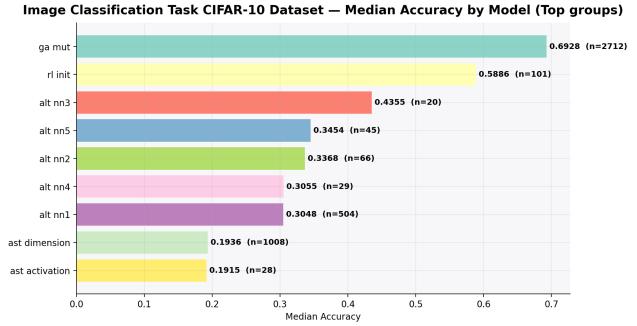

Figure 17. Median accuracy per model group for image classification task on the CIFAR-10 dataset.

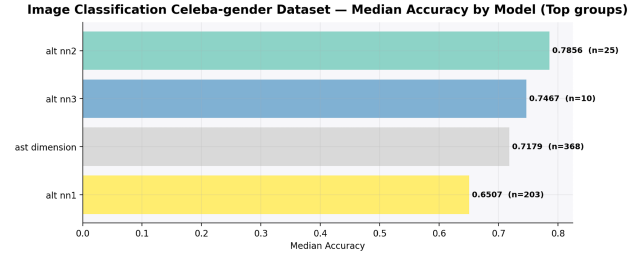

Figure 18. Median accuracy per model group for image classification tasks on the Celeba-gender dataset.

14. Detailed Results

14.1. Image Captioning

NN-Caption produces 357 distinct MS COCO captioning architectures that compile and train successfully. Vary-

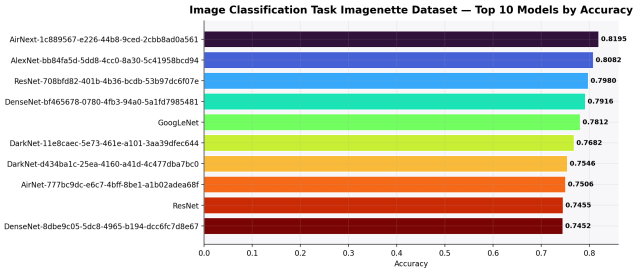

Figure 19. Top 10 image classification models ranked by their best recorded accuracy for the Imagenette dataset.

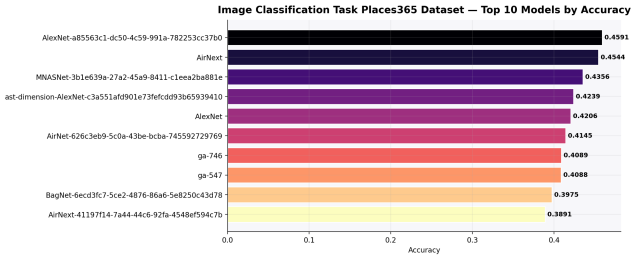

Figure 20. Top 10 image classification models ranked by their best recorded accuracy for the Places365 dataset.

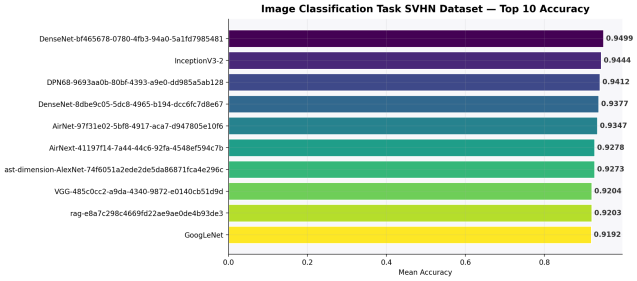

Figure 21. Top 10 image classification models ranked by their best recorded accuracy for the SVHN dataset.

ing prompt complexity between 5 and 10 exemplar model snippets changes the fraction of runnable generations from 80% to 50%, respectively, indicating reduced code validity with longer prompts. The LLM explores diverse encoder–decoder combinations, including ResNet encoders with Squeeze-and-Excitation, ConvNeXt and EfficientNet backbones, and LSTM/GRU/Transformer decoders. Under the shared 3-epoch screening protocol, the best generated model, a modified ResNet-50 encoder with a Transformer decoder (hidden size 768, 8 attention heads [1]), attains BLEU-4 = 0.1192, compared to 0.3246 for the engineered ResNet+LSTM baseline. In a separate 50-epoch follow-up run, the strongest generated ResNet-50 + Transformer configuration reaches BLEU-4 = 0.317. This reflects a trade-off between accuracy and architectural diversity in the corpus.

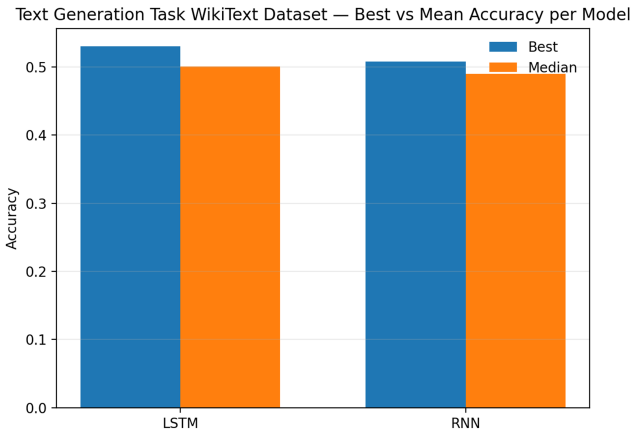

Figure 22. Best and median model accuracy or text generation Task on the Wikitext dataset.

14.2. Text-to-Image

Across paired text–image evaluations, the three text-to-image model families exhibit distinct behaviour. The diffusion model (UNet-D) achieves CLIP text–image similarity scores of 0.17–0.24; an earlier cross-attention, classifier-free-guidance variant failed to converge and was replaced by this simpler architecture. The GAN, using an LSTM text encoder with a convolutional generator–discriminator, reaches CLIP ≈ 0.214 after 150 training epochs once stabilized via a discriminator-favoured update schedule, reduced learning rate, and modified optimizer parameters. CVAE-GAN variants evolve from a DistilBERT-based 128×128 model (CLIP 0.21–0.22) through a 256×256 configuration with pronounced checkerboard artefacts to a final design with a CLIP text encoder, an upsampling-plus-convolution decoder, and a PatchGAN-style discriminator trained with reconstruction, perceptual, Kullback–Leibler, and adversarial losses; this model attains a peak CLIP score of 0.2751, yielding the strongest semantic alignment but predominantly abstract, non-photorealistic textures.

14.3. Mixture-of-Experts

Eight Mixture-of-Experts variants are evaluated on CIFAR-10. Among homogeneous configurations, a tuned model (MoEv7) achieves the highest test accuracy of 93.9%. The heterogeneous MoE, combining AlexNet, AirNet, DenseNet, and BagNet experts under soft routing, reaches 93.13% test accuracy and outperforms each constituent backbone trained individually, demonstrating gains from expert diversity.

14.4. Genetic Algorithm

On CIFAR-10, the two genetic algorithm search spaces produce distinct champion architectures. In the hyperparameter-restricted *Basic Architecture* search,

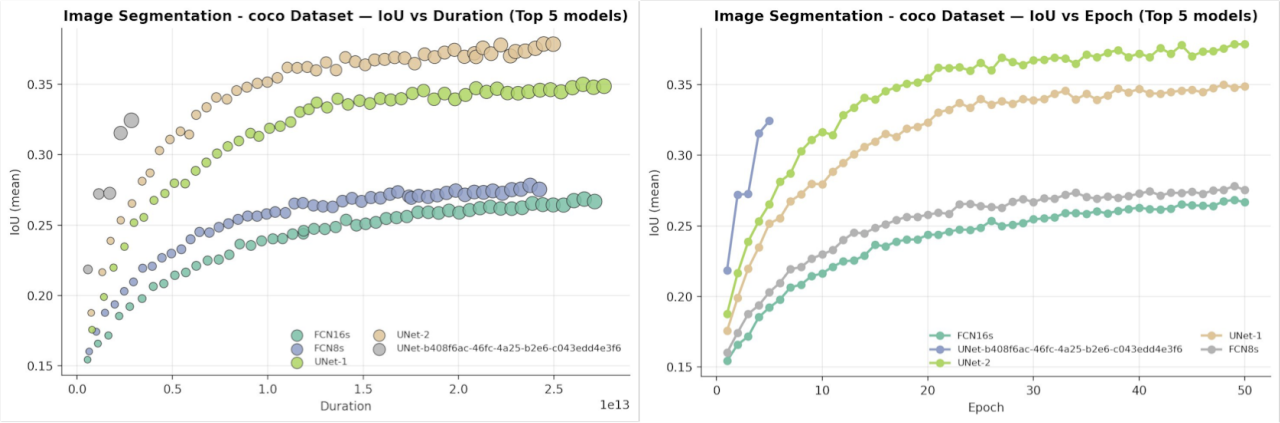

Figure 23. IoU progression over training epochs and relationship between model IoU and training duration for the top-performing image segmentation models.

the best model attains 62.76% test accuracy after first epoch of training and retains an 11×11 initial convolutional kernel with a dropout rate of 0.6. In the *Block-type Variation* search, where pooling type, activation function, and batch normalization are also evolved, the champion improves to 80.04% test accuracy, omits the large initial kernel, incorporates batch normalization, primarily uses 3×3 convolutions, and adopts mixed pooling, indicating substantial benefits from allowing block-level structural modifications.

14.5. NN-RAG

NN-RAG processes 1,289 candidate neural network blocks collected from the `timm`, `torchvision`, and `transformers` libraries; 941 modules (73.0%) pass syntax, compilation, and sandbox-execution checks and are retained as fully executable components. The resulting library spans attention mechanisms, convolutional and transformer blocks, normalization layers, and loss functions, and architectures that incorporate these modules are tagged with the prefix "rag-" in LEMUR 2.

14.6. Few-Shot Prompting

Across seven vision datasets, few-shot prompting yields 4,033 unique architectures tagged "alt-". Varying the number of exemplar architectures n in the prompt induces a non-monotonic effect: $n = 3$ achieves the highest dataset-balanced mean accuracy of 53.1% and improves CIFAR-100 accuracy by 11.6 percentage points ($p = 0.001$), whereas $n > 3$ reduces average accuracy and $n = 6$ produces a 99.8% failure rate, with most generations failing to compile or train, consistent with context overflow. Whitespace-normalized hash-based deduplication prevents training of approximately 100 exact-duplicate architectures, saving an estimated 200–300 GPU hours under the LEMUR 2 training budget. Dataset-balanced averaging is further found to be critical: naive aggregation without balancing can inflate mean accuracy by up to 15.7 percentage points due to uneven architecture counts per dataset.

14.7. Data Transformation

In the LLM-based setting, 100 augmentation pipelines are generated; 53 pass syntactic validation, implying that approximately 47% contain syntax or compatibility errors. Among the valid pipelines, the best configuration—`RandomResizedCrop` + `ColorJitter` + `RandomHorizontalFlip` + `GaussianBlur` followed by the fixed suffix—achieves 57.28% validation accuracy after one training epoch. The combinatorial search enumerates 6,000 pipelines with one, two, or three variable transforms followed by `Resize`, `ToTensor`, and `Normalization`; its top-performing pipeline uses a single `RandomPosterize` transform and reaches 61.24% validation accuracy, outperforming the best LLM-generated pipeline under the same training protocol.

14.8. NN-VR

NN-VR is applied to more than 10,000 pre-trained models exported from the LEMUR corpus. For standard convolutional architectures, approximately 95% of models that reach the Unity import stage achieve full compatibility, including ONNX import into Unity, operator support under Barracuda, and adherence to memory constraints. A case study on an AirNet-based model illustrates typical behaviour: initial deployments revealed memory allocation conflicts that were subsequently fixed by the automated optimization stage via adjusted shader compilation and resource allocation, yielding a configuration that meets all validation criteria. For models that pass compatibility checks, NN-VR records whether the 11 ms per-frame latency budget required for 90 Hz VR render-

ing is satisfied and verifies that Unity/Barracuda predictions deviate by at most 1% from the original PyTorch outputs. These deployment-oriented measurements complement LEMUR 2 training metrics and enable analysis of accuracy–latency–memory trade-offs for VR applications.

References

[1] Vaswani Ashish. Attention is all you need. *Advances in neural information processing systems*, 30:I, 2017. 22