# Enhancing LLM-Based Neural Network Generation: Few-Shot Prompting and Efficient Validation for Automated Architecture Design

Supplementary Material

## S1. Detailed Algorithms

### S1.1. Few-Shot Architecture Prompting (FSAP)

The FSAP procedure formalises the prompt construction strategy described in Section 3.2 of the main paper. Given a target dataset $\mathcal{D}$ and desired context size $n$, the algorithm queries LEMUR for high-performing architectures, designates one as the reference model, and draws $n$ supporting examples via uniform random sampling to isolate the effect of context size from example-quality effects. The generated prompt is passed to DeepSeek Coder 7B at temperature $0.6$ to produce a complete PyTorch architecture.

**Algorithm 1** Few-Shot Architecture Prompting (FSAP)

**Require:** Dataset $\mathcal{D}$, example count $n \in \{1, \ldots, 6\}$
**Require:** LEMUR database $\mathcal{L}$
**Ensure:** Generated architecture code $\mathcal{C}$
$\mathcal{M} \leftarrow \text{Query}(\mathcal{L}, \mathcal{D}, \text{top-accuracy})$
$\mathcal{M}_{\text{ref}} \leftarrow \text{Sample}(\mathcal{M}, 1)$
$\mathcal{M}_{\text{cand}} \leftarrow \mathcal{M} \setminus \mathcal{M}_{\text{ref}}$
**if** $|\mathcal{M}_{\text{cand}}| \geq n$ **then**
    $\mathcal{M}_{\text{supp}} \leftarrow \text{RandomSample}(\mathcal{M}_{\text{cand}}, n)$
**else**
    $\mathcal{M}_{\text{supp}} \leftarrow \mathcal{M}_{\text{cand}}$
**end if**
$\mathcal{P} \leftarrow \text{TaskDescription}(\mathcal{D})$
$\mathcal{P} \mathrel{+}= \text{DatasetSpec}(\mathcal{D})$
$\mathcal{P} \mathrel{+}= \text{FormatModel}(\mathcal{M}_{\text{ref}})$
**for** $m \in \mathcal{M}_{\text{supp}}$ **do**
    $\mathcal{P} \mathrel{+}= \text{FormatSupporting}(m)$
**end for**
$\mathcal{P} \mathrel{+}= \text{GenerationRules}()$
$\mathcal{C} \leftarrow \text{DeepSeekCoder}(\mathcal{P}, T{=}0.6)$
**return** $\mathcal{C}$

### S1.2. Whitespace-Normalized Hash Validation

The hash validation procedure safely eliminates formatting-level duplicate architectures prior to the computationally demanding training phase, effectively saving 2–3 GPU hours per rejected instance. The automated three-step process — comprehensive whitespace removal, MD5 hashing, and B-tree indexed database lookup — runs in $O(|\mathcal{C}| + \log N)$ time. This is heavily dominated by the linear code-length term, consistently achieving sub-millisecond latency in practical applications.

**Algorithm 2** Whitespace-Normalized Hash Validation

**Require:** Code string $\mathcal{C}$, LEMUR database $\mathcal{L}$
**Ensure:** Decision: {ACCEPT, REJECT}
$\mathcal{C}' \leftarrow \text{RemoveWhitespace}(\mathcal{C})$ ▷ $O(|\mathcal{C}|)$
$h \leftarrow \text{MD5}(\mathcal{C}')$ ▷ $O(|\mathcal{C}|)$, hardware-accel.
$\mathcal{H} \leftarrow \text{Query}(\mathcal{L}, \texttt{SELECT nn\_id FROM lemur})$ ▷ $O(\log N)$
**if** $h \in \mathcal{H}$ **then**
    **return** REJECT
**else**
    **return** ACCEPT
**end if**

The total asymptotic complexity of $O(|\mathcal{C}| + \log N) \approx O(|\mathcal{C}|)$ makes this hash validation significantly faster than standard AST parsing ($O(|\mathcal{C}|^{1.5})$ in practice) or GraphCodeBERT inference (50–200 ms per sample). Furthermore, this lightweight approach directly targets the most dominant duplication pattern observed in LLM pipelines: superficial formatting variations arising from inconsistent indentation and redundant whitespace within raw code generation outputs.

## S2. Complete Prompt Template

The full prompt template utilized across all FSAP experiments is detailed below. The construction of this prompt acts as the vital control interface for the LLM-driven generation pipeline. The key design choices driving this template are:

(1) **Accuracy labels alongside each model:** By providing explicit performance metrics mapped to each code block, we establish a quantifiable, performance-based guidance signal. This context encourages the LLM to correlate specific architectural motifs with high empirical success, biasing its output toward proven structural patterns rather than arbitrary mutations.

(2) **Explicit synthesis instructions:** The directive to "combine best elements" deliberately shifts the model's generative priority away from simply duplicating the reference architecture or making shallow alterations. Instead, it actively promotes structural crossover, challenging the LLM to intelligently synthesize novel, hybrid architectures by merging advantageous features from the diverse supporting examples.

(3) **Strict output constraints:** By strictly enforcing a fixed class name (`Net`), rigid method signatures, and PyTorch-only dependencies, we provide a reliable scaffold

that guarantees every generated architecture remains syntactically valid. This ensures zero friction when integrating models into our automated training pipeline, preserving the LLM's architectural freedom in designing internal topologies.

```
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:
```

## S3. Complete Architecture Code Examples

This section provides the full PyTorch source code for the four representative architectures discussed qualitatively in Section 5.2.1 of the main paper. The first three (alt-nn3 variants) illustrate the architectural synthesis enabled by $n$=3 prompting; the fourth (alt-nn1 baseline) demonstrates the shallow variation characteristic of single-example prompting for direct comparison.

### S3.1. Example 1: ResNet + AlexNet Hybrid

**Model ID:** *alt-nn3-34df74344dd63a558c4b6413b809f6ed*

This architecture exemplifies successful cross-paradigm synthesis, seamlessly integrating modern representation learning techniques with classical, high-capacity classification heads. Specifically, it synthesises ResNet-style residual units (`AirUnit` blocks featuring identity skip connections and batch normalisation to mitigate vanishing gradients) with AlexNet's signature large-scale, two-stage fully-connected classifier (utilizing 4096-unit dense layers regularized by aggressive dropout).

This structural hybrid leverages the deep, stable feature extraction capabilities of residual networks while preserving the massive parameterization of early deep learning classifiers. Crucially, such sophisticated topological combinations are qualitatively absent in all $n$=1 generated variants, which typically default to shallow, sequential template alterations. This strongly corroborates that providing three supporting examples uniquely enables the LLM to perform genuine architectural pattern merging across disparate network families.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class AirUnit(nn.Module):
    """ResNet-style residual block"""
    def __init__(self, in_ch, out_ch, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, 3, stride,
     1, bias=False)
        self.bn1   = nn.BatchNorm2d(out_ch)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, 1, 1,
    bias=False)
        self.bn2   = nn.BatchNorm2d(out_ch)
        self.down  = None
        if stride != 1 or in_ch != out_ch:
            self.down = nn.Sequential(
                nn.Conv2d(in_ch, out_ch, 1, stride, bias
    =False),
                nn.BatchNorm2d(out_ch))

    def forward(self, x):
        identity = x
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        if self.down:
            identity = self.down(x)
        return F.relu(out + identity)

class Net(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.l1  = AirUnit(3,   64,  stride=2)
        self.l2  = AirUnit(64,  128, stride=2)
        self.l3  = AirUnit(128, 256, stride=2)
        self.pool = nn.AdaptiveAvgPool2d((1,1))
        self.fc1  = nn.Linear(256, 4096) # AlexNet
        self.drop = nn.Dropout(0.5)
        self.fc2  = nn.Linear(4096, 4096)
        self.out  = nn.Linear(4096, num_classes)

    def forward(self, x):
        x = self.l3(self.l2(self.l1(x)))
        x = torch.flatten(self.pool(x), 1)
        x = self.drop(F.relu(self.fc1(x)))
        x = self.drop(F.relu(self.fc2(x)))
        return self.out(x)

    def supported_hyperparameters(self):
        return ['lr', 'momentum']
```

### S3.2. Example 2: DPN-Inspired Hybrid

**Model ID:** *alt-nn3-57d770565afc5e0d651cb0938fc8f942*

This specific generated architecture successfully merges standard Dual Path Network (DPN) bottleneck feature-extraction blocks ($1\times1 \rightarrow 3\times3 \rightarrow 1\times1$) with a highly progressive and customized convolutional backbone design. The unusually asymmetric mid-network channel progression (64→128→**440**→384→192) serves as strong empirical evidence of the model's creative architectural synthesis rather than mere direct template copying. Crucially, this

exact dimensional configuration appears in absolutely none of the provided LEMUR reference models utilized in the initial prompt context. Furthermore, the final classification head effectively incorporates batch normalization directly within the fully-connected dense layers—a distinctly modern architectural regularization pattern that clearly demonstrates robust, cross-architecture feature pattern transfer during the generation phase.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class DPNBlock(nn.Module):
    """DPN-inspired bottleneck"""
    def __init__(self, in_ch, out_ch, stride=1):
        super().__init__()
        self.c1 = nn.Conv2d(in_ch, out_ch, 1, bias=False
    )
        self.b1 = nn.BatchNorm2d(out_ch)
        self.c2 = nn.Conv2d(out_ch, out_ch, 3, stride,
    1, bias=False)
        self.b2 = nn.BatchNorm2d(out_ch)
        self.c3 = nn.Conv2d(out_ch, out_ch, 1, bias=
    False)
        self.b3 = nn.BatchNorm2d(out_ch)
        self.sc = nn.Sequential()
        if stride != 1 or in_ch != out_ch:
            self.sc = nn.Sequential(
                nn.Conv2d(in_ch, out_ch, 1, stride, bias
    =False),
                nn.BatchNorm2d(out_ch))

    def forward(self, x):
        out = F.relu(self.b1(self.c1(x)))
        out = F.relu(self.b2(self.c2(out)))
        out = self.b3(self.c3(out))
        return F.relu(out + self.sc(x))

class Net(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        # Unusual 440-channel mid-layer: creative
    synthesis
        self.l1   = DPNBlock(3,   64,  1)
        self.l2   = DPNBlock(64,  128, 2)
        self.l3   = DPNBlock(128, 440, 2)
        self.l4   = DPNBlock(440, 384, 1)
        self.l5   = DPNBlock(384, 192, 2)
        self.pool = nn.AdaptiveAvgPool2d((1,1))
        self.fc1  = nn.Linear(192, 512)
        self.bn   = nn.BatchNorm1d(512)
        self.fc2  = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.l5(self.l4(self.l3(self.l2(self.l1(x)))
    ))
        x = torch.flatten(self.pool(x), 1)
        return self.fc2(F.relu(self.bn(self.fc1(x))))

    def supported_hyperparameters(self):
        return ['lr', 'momentum']
```

### S3.3. Example 3: Hierarchical Residual Units + Multi-Scale Features

**Model ID:** *alt-nn3-dcb59f747eb3b27c0552d2aea356e33a*

This architecture synthesizes ResNet-style `AirUnit` blocks with three-layer residual paths into a hierarchical feature extraction pipeline combining varying spatial scales. The model begins with large kernel convolutions (7×7, stride 3) for rapid downsampling before cascading into sophisticated residual units integrating pooling steps implicitly through strided convolutions. This design merges classical and modern approaches: aggressive early spatial reduction from AlexNet with deep residual learning from ResNet, creating a compact yet expressive architecture suitable for CIFAR-scale tasks.

```
class AirInitBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels, out_channels,
     kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
    def forward(self, x): return self.layers(x)

class AirUnit(nn.Module):
    def __init__(self, in_channels, out_channels, stride
    ):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels, out_channels,
     kernel_size=3, stride=stride, padding=1),
            nn.BatchNorm2d(out_channels), nn.ReLU(
     inplace=True),
            nn.Conv2d(out_channels, out_channels,
     kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(out_channels), nn.ReLU(
     inplace=True),
            nn.Conv2d(out_channels, out_channels,
     kernel_size=3, stride=1, padding=1),
        )
        self.downsample = (
            nn.Sequential(
                nn.Conv2d(in_channels, out_channels,
     kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            ) if stride != 1 or in_channels !=
     out_channels else nn.Identity()
        )
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        return self.relu(self.layers(x) + self.
     downsample(x))

class Net(nn.Module):
    def __init__(self, in_shape: tuple, out_shape: tuple
     , prm: dict, device: torch.device) -> None:
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_shape[1], 96, kernel_size=7,
     stride=3, padding=2),
            nn.BatchNorm2d(96), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            AirInitBlock(96, 192),
            AirUnit(192, 384, stride=2),
            AirUnit(384, 256, stride=1),
            AirUnit(256, 256, stride=2),
            nn.AdaptiveAvgPool2d((6, 6))
        )
        self.classifier = nn.Sequential(
            nn.Dropout(p=prm['dropout']),
            nn.Linear(256 * 6 * 6, 4096), nn.ReLU(
     inplace=True),
            nn.Dropout(p=prm['dropout']),
            nn.Linear(4096, out_shape[0])
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.classifier(torch.flatten(self.
     features(x), 1))
```

### S3.4. Example 4: Baseline alt-nn1 (Single Example)

**Model ID:** *alt-nn1-0e40be6fbc3426f57a305bfd8b8148fa*

This representative $n$=1 baseline architecture illustrates the shallow sequential variation pattern typical in single-prompt experiments. Lacking modular abstraction, residual shortcuts, or normalization, the model relies on a dense linear progression of GELU-activated pooling operations characteristic of straightforward, non-creative structural mutation relative to references.

```
class Net(nn.Module):
    def __init__(self, in_shape: tuple, out_shape: tuple
    , prm: dict, device: torch.device) -> None:
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_shape[1], 64, kernel_size=11,
    stride=4, padding=2),
            nn.GELU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2)
    ,
            nn.GELU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding
    =1),
            nn.GELU(),
            nn.Conv2d(384, 256, kernel_size=3, padding
    =1),
            nn.GELU(),
            nn.Conv2d(256, 256, kernel_size=3, padding
    =1),
            nn.GELU(),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(p=prm['dropout']),
            nn.Linear(256 * 6 * 6, 4096),
            nn.GELU(),
            nn.Dropout(p=prm['dropout']),
            nn.Linear(4096, 643),
            nn.GELU(),
            nn.Linear(643, out_shape[0])
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.classifier(torch.flatten(self.
    avgpool(self.features(x)), 1))
```

## S4. Extended Per-Dataset Results

This section expands upon the dataset-balanced evaluation discussed in the main paper by providing the raw, per-dataset quantitative performance and statistical significance matrices.

### S4.1. Per-Dataset Performance Analysis

Table **??** presents detailed per-dataset results with statistical significance markers.

**Observations**:

- **Task-dependent variation:** The relative performance of prompt variants differs across datasets. In particular, $n$=3 achieves the highest mean accuracy on CIFAR-100, while other datasets exhibit different preferences.

Table S1. Per-Dataset Performance (Mean Accuracy %) on 1-epoch. Asterisks denote significance vs. baseline: * $p < 0.05$, ** $p < 0.01$. Best per dataset in **bold**.

| Dataset | alt-nn1 | alt-nn2 | alt-nn3 | alt-nn4 | alt-nn5 |
|---|---|---|---|---|---|
| MNIST | 96.5 | 93.8* | **97.1** | 93.9 | 94.7 |
| CelebA-Gender | 75.8 | **82.3*** | 74.4 | 80.6 | 72.1 |
| CIFAR-10 | **38.7** | 36.1 | 38.3 | 30.3* | 34.0 |
| CIFAR-100 | 14.5 | 7.4* | **26.1**** | 10.8 | 10.5 |
| ImageNette | **44.2** | 36.4* | 42.5 | 29.8** | 18.2** |
| SVHN | 39.2 | **43.1** | 40.0 | 38.3 | 28.5 |
| **Bal. Mean** | 51.5 | 49.8 | **53.1** | 47.3 | 43.0 |

- **Simple vs. complex tasks:** On simpler benchmarks such as MNIST, performance remains high across all variants ($> 93\%$), suggesting limited sensitivity to prompt configuration under low task complexity.
- **Context scaling effects:** Variants with $n$>3 exhibit lower performance on several datasets (e.g., ImageNette), indicating that larger prompt contexts do not consistently translate into improved early performance.

### S4.2. Statistical Significance Analysis

Table **??** presents statistical validation focusing on key findings.

Table S2. Statistical Significance Tests (per-dataset comparison vs. alt-nn1 baseline). Only results with $p < 0.05$ shown.

| Dataset | Comparison | $\Delta$ | p | d |
|---|---|---|---|---|
| **CIFAR-100** | **alt-nn3 vs alt-nn1** | **+11.6%** | **0.001** | **0.73** |
| MNIST | alt-nn2 vs alt-nn1 | -2.7% | 0.029 | -0.19 |
| CelebA | alt-nn2 vs alt-nn1 | +6.5% | 0.038 | 0.41 |
| CIFAR-10 | alt-nn4 vs alt-nn1 | -8.4% | 0.016 | -0.54 |
| ImageNette | alt-nn4 vs alt-nn1 | -14.5% | 0.010 | -1.20 |

**Key Statistical Result:** On CIFAR-100, the $n$=3 configuration achieves higher mean accuracy than $n$=1 ($p$=0.001, Cohen's $d$=0.73), suggesting that moderate context enrichment may be particularly beneficial in fine-grained classification settings under the rapid screening protocol.

In contrast, configurations with $n$>3 show statistically significant lower performance on certain datasets (e.g., ImageNette), indicating that larger prompt contexts can introduce instability or diminishing returns in early training performance. We emphasise that these comparisons reflect early-epoch behavior under a rapid screening protocol and are intended to capture relative trends across prompt variants rather than final converged performance.