Enhancing Vision and Language Interplay through Active Contextual Modulation

Rashmi Kapu
University of Maryland
College Park, MD
rashmik@umd.edu

Chaitanya Kulkarni
University of Maryland
College Park, MD
chaitkul@umd.edu

Eadom Dessalene
University of Maryland
College Park, MD
edessale@umd.edu

***Abstract*—This project aims to enhance the understanding of vision and language interaction by developing an approach that dynamically adjusts visual representations based on language provided contextual cues. Our technique makes vision a context driven, dynamic process by actively influencing visual representations through language, in contrast to standard systems that passively aggregate features or apply established algorithms. Existing approaches result in constrained systems because they do not effectively exchange context between large language models and low level visual modules. Our solution introduces a feedback interface between LLMs and vision networks, allowing context modulated control. This enables the alignment of vision modules with complex, combined contexts, represented through programs that facilitate sequential feedback to address relevant contexts for tasks. Our method addresses the limitations of existing systems like VISPROG and VIPER-GPT, which lack a mechanism for LLMs to infuse context into the visual processing pipeline. We propose a feedback loop where EvoPrompt [1] identifies the most relevant contextual cues and their optimal application points within the vision network, utilizing MidVision feedback (MVF) [2] for context-based affine transformations.**

I. Introduction

The project aims to revolutionize the understanding of the interplay between vision and language by proposing an innovative approach that actively modulates visual representations based on contextual cues provided by language. Unlike traditional methods that simply combine language and vision features or employ predefined processes, our approach involves using language to dynamically influence visual representations, thus making vision an active process where context plays a crucial role; it has to be actively inferred and used to sequentially sequential application of affines belonging to different contexts over the visual input (static images).

A defining trend in approaches like VISPROG or VIPER-GPT is that there is little exchange of context information between the low-level visual modules and the LLMs calling them - this is because they lack an interface between vision networks and LLMs that allows for the imbuing of context over visual features. The programs produced by the LLM can only operate explicitly over the high-level outputs of these visual modules, but have no access to intermediate context information, making these systems inexpressive and brittle. The scope of what the programs can achieve is very limited. The interface between LLMs and vision networks is feedback, enabling context-modulated control of vision networks by an LLM. In MVF, context was binary (animate/inanimate, big/small, etc), and only 1 context at a time was used. Thanks to LLMs, vision modules can be made in alignment with a complex combination of contexts (visually derived contexts, or knowledge-based context) represented through a program, which will capture a sequence of feedback calls based on which contexts are relevant in solving a task.

II. Related Work

A. EvoPrompting

This paper has demonstrated the impressive accomplishments of language models in code generation. The authors have explored the use of language models as general adaptive mutation and crossover operators for an evolutionary neural architecture search (NAS) algorithm. Although language models struggle with NAS tasks, the integration of prompt engineering with soft prompt tuning, EVOPROMPTING, consistently produces diverse and high performing models. This methodology has proved effective on the MNIST-1D dataset, which created convolutional architecture variants that surpass the architectures designed by human experts and naive few shot-prompting in terms of accuracy and model size. The authors have also applied EVOPROMPTING to the search of Graph Neural Networks on the CLRS Algorithmic Reasoning Benchmark, which resulted in architectures that outperform existing state-of-the-art models on numerous tasks while maintaining similar model sizes. This technique has designed precise and efficient neural network architectures across various machine learning tasks.

B. Mid Vision Feedback

In this paper, the authors present a novel mechanism called Mid-Vision Feedback (MVF), which plays a crucial role in modulating perception in biological vision. MVF operates by aligning perception with high-level categorical expectations, linking these expectations with linear transformations applied to feature vectors within a neural network's mid-level. This alignment towards high-level contexts significantly enhances both overall accuracy and contextual consistency. During the training process, MVF incorporates a loss term that encourages a greater distinction between feature vectors associated with different contexts, which strengthens context-specific learning. MVF demonstrates flexibility in accommodating various sources of contextual expectations and enables the seamless integration of symbolic systems with deep vision architectures

in a top-down fashion. Through empirical evaluations, the authors illustrate MVF's superior performance compared to post-hoc filtering methods for incorporating contextual knowledge. Configurations utilizing predicted context, even in the absence of prior contextual knowledge, exhibit superior performance over those lacking context awareness.

III. Methodology

A. EVOPROMPTING on MNIST-1D Dataset

Our aim is to replicate the process of finding a set of neural network architectures that maximize a reward function for arbitrary pairs of dataset and task. While previous studies have utilized evolutionary methods for architecture search, we seek to replicate and modify their approach by integrating a language model for crossover and mutation operations.

This methodology offers several advantages. Unlike previous approaches that required defining a specific search space, this approach allows for exploration of any neural network architecture represented in Python. This provides greater flexibility and diversity in the generated architectures while minimizing the need for manual design. Modern pre-trained LMs, trained on extensive datasets offer valuable insights into code structure and functionality. Also, LMs can adapt their crossover operations over time to generate more rewarding architectures, which enhance the replication process.

To reproduce Evoprompt on MNIST-1D dataset (40*1 features), we propose the following approach :

*1) **Initialization**:*

- Defines key parameters like the number of evolutionary rounds (10), number of prompts per round (1), number of model samples per prompt (3), and evaluation settings.
- Specifies the task (MNIST-1D digit classification) and the folder containing initial seed architectures.

*2) **Population Seeding**:*

- Loads the seed architectures from the specified folder.Evaluates each seed architecture if not pre-evaluated (accuracy and model size).
- Creates the initial population by storing each seed's code, its evaluation metrics, and a calculated fitness score (combining accuracy and model size).

*3) **Evolutionary Loop**:* In each round:

- Crossover and Mutation: Generates few-shot prompts: Creates prompts that guide the search towards better models. These prompts are based on: Randomly selected parent architectures (code and performance) from the current population. A target accuracy improvement and model size reduction factor. Uses a large language model (LLM) to generate new child model architectures (code snippets) based on each prompt using API (OpenAI API for GPT-4.0 in our case).
- Evaluation: Evaluates each generated child architecture on the MNIST-1D dataset (accuracy and model size). Calculates the fitness score for each child (combining accuracy and model size).
- Selection: Filters out children with accuracy below a certain threshold. Combines the evaluated children with the existing population. Selects the top-performing models based on their fitness scores to form the next generation's population.

B. Final Selection:

After a set number of rounds, the algorithm selects the top models from the final population based on their fitness scores. These are the most efficient and accurate models discovered during the evolution process.

C. EVOPROMPTING on CIFAR-10 Dataset

To apply our methodology to the CIFAR-10 dataset, we will use the same process of finding optimal neural network architectures with the help of a large language model for crossover and mutation operations. We will modify our evaluation metrics and fitness functions to suit the more complex, high dimensional images in the CIFAR-10 dataset. This includes creating and testing new architectures tailored for CIFAR-10 classification, with the goal of enhancing model accuracy while preserving the flexibility of our evolutionary approach.

D. Integration of Feedback

Currently, the search space for our evolutionary prompting method is performed over a set of PyTorch neural network operations, including but not limited to the following: $\{Conv2D, Linear, Flatten, MaxPool2D, ReLu\}$ where Conv2D corresponds to convolution operations, Linear corresponds to fully connected operations, flatten corresponds to tensor reshaping, MaxPool2D corresponds to spatial averaging, and ReLu is a non-linear activation function. We argue the success of the LLM across the experiments thus far has little to do with the expressiveness of the aforementioned operations, and far more to do with the plethora of architectures within the training set of GPT-4 that were hand-designed for the task of character classification. We expect this approach implemented thus far to fail for more complex datasets that have received less attention in the vision community and require significant domain knowledge.

We propose an alternative to EvoPrompting that involves the tight integration of the following two concepts: *program synthesis* and *feedback*. Rather than prompt GPT-4 to produce a sequence of PyTorch operations as done so far, we propose that GPT-4 performs *program synthesis*, where programs represent a sequence of preconditions (based on the context predictions) and operations (representing feedback modulation). This program serves as a stand-in replacement for the architecture of the vision network. Below we offer two qualitative examples.

(Example 1): Subset of actual GPT-4 output, ignoring prompt instructions.

```
#Task: classify image that contains either a
    dog or a cat
Output:
  if prob('whiskers') > 0.4:
```

```
    mid_feedback('whiskers')
    late_feedback('whiskers')
  else if prob('long_nose') > 0.9:
    mid_feedback('long_nose')
  if prob('stripes') > 0.5:
    early_feedback('stripes')
    mid_feedback('stripes')
  if prob('claws') > 0.8:
    mid_feedback('claws')
```

(Example 2): Subset of actual GPT-4 output, ignoring prompt instructions.

```
Task: classify image that contains either a bed
      or a table.
Output:
  if prob('food') > 0.1:
    mid_feedback('food')
    late_feedback('food')
  else if prob('sheets') > 0.7:
    late_feedback('sheets')
  if prob('chair') > 0.7:
    late_feedback('chair')
```

Rather than have the LLM directly produce a PyTorch model architecture, we propose the LLM produces a program, which in turn directly maps onto a sequence of feedback operations. We note in the examples the *early feedback*, *mid feedback*, and *late feedback* calls correspond to the chosen injection layers (feedback injection early in the network, feedback injection at the mid-point of the network, and feedback injection late in the network). The proposed approach is as follows:

- A context extraction module is run over the input image using a pre-trained CLIP model, producing thousands of contexts c, each with probability score p.
- A description of the classification task (e.g. *classify image that contains a dog or a cat*) is fed to an LLM, which performs an evolutionary search to arrive at generated child programs p_i, for $\forall i \in n$. Two example programs for different tasks are shown above.
- The contexts c along with their associated probability scores p are fed to program p_i, producing a sequence of feedback calls to execute. For example, say in Example 1 the extracted contexts are $whiskers, longnose, stripes, claws$ with probabilities $0.6, 0.0, 0.8, 0.0$, respectively. Then the returned feedback calls would be mid feedback('whiskers'), late feedback('whiskers'), early feedback('stripes'), mid feedback('stripes').
- These feedback calls would in turn be used to bias the visual features of our network in accordance with context, exactly as was done in the MVF paper.
- The program accuracy for all programs p are evaluated and sent back to the LLM for improvement through crossover and mutation.

There are many benefits to centering the search space of the LLM around context and feedback. One promising benefit is that the classification task is represented as a sequence of sub-tasks over low-level, mid-level, and high-level features o, bringing hierarchy and compositionality into the problem. Another promising benefit is that by operating over context, the LLM can infuse domain knowledge into the programs. In Example 1, in discriminating between a dog and a cat, the LLM quickly ascertains the primary contextual differences between a dog and cat - that cats have whiskers, stripes, and claws, and dogs may or may not have long noses. The program relies primarily on these contexts in differentiating between the two classes.

IV. Experiments and Results

A. MNIST-1D Dataset

In the MNIST-1D dataset, each example is one-dimensional and represented by a 40-dimensional vector. The dataset consists of 5000 images. 80% of the dataset is used for training the model, while the remaining 20% is used for validation. While the images used for validation change with each model run, the split percentage stays the same. Hence, the model is run five times before its performance is evaluated.

- Number of rounds = 10
- Number of few-shot prompts per round = 1
- Number of samples to generate per prompt = 3
- Number of in-context examples per prompt = 3
- Number of survivors to select per generation = 3
- Number of times to run each model = 5

Fig 1, 2 and 3 depict the newly generated architectures (best model #1 and #2) using 3 in-context examples in few shot prompt. These models are run 5 times (5 simultaneous thread executions) and the average accuracy of all the epochs averaged over 5 runs is shown in the table II.

If the provided seed yields better results than the generated child architectures, the code retains the seed and tends to produce more and more child architectures similar to the particular seed in the next rounds. We can see from the figures that all of the best architectures are mostly similar to each other with very slight variations, for the same reason.

Average accuracy for 5 runs (15 epochs), no.of parameters of the given seeds are shown in I.

Seed	Accuracy	Model Parameters
Seed 1	82.875%	173002
Seed 2	75.88%	12710
Seed 3	57.75%	15210

TABLE I

Table II depicting the accuracy averaged over 5 runs (5 threads) and model parameters of the best child architectures generated by the LLM. The no.of epochs for each of these architectures may vary, since it is generated by the language model.

Child Architecture	Accuracy	Model Parameters
Best model 1	90.50 %	125060
Best model 2	88.65 %	212234
Best model 3	84.37 %	291178

TABLE II

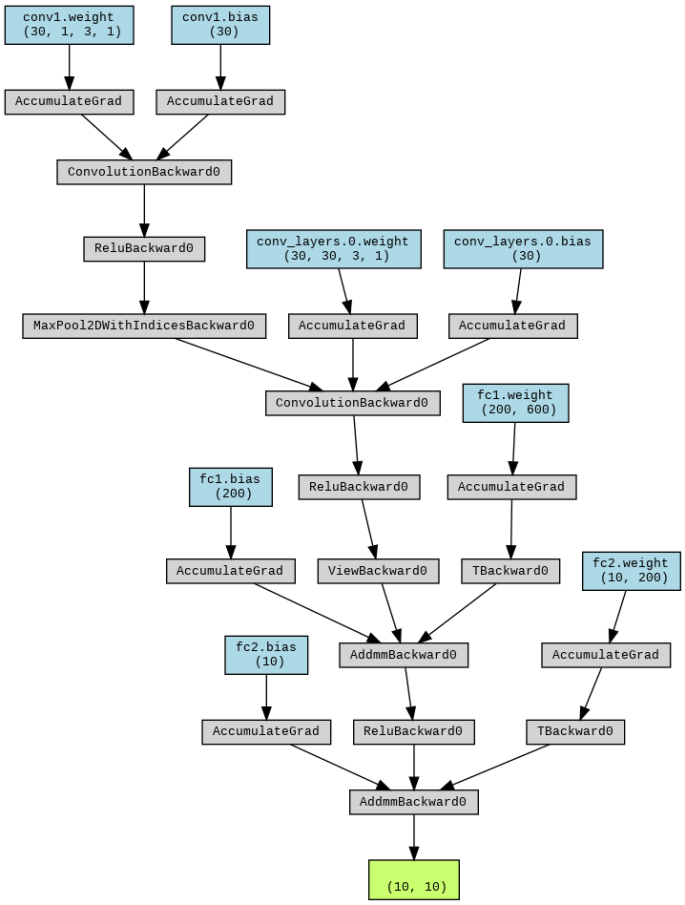

Fig. 1: Visualization of the generated new architecture- Best model #1 after 10 evolutionary rounds

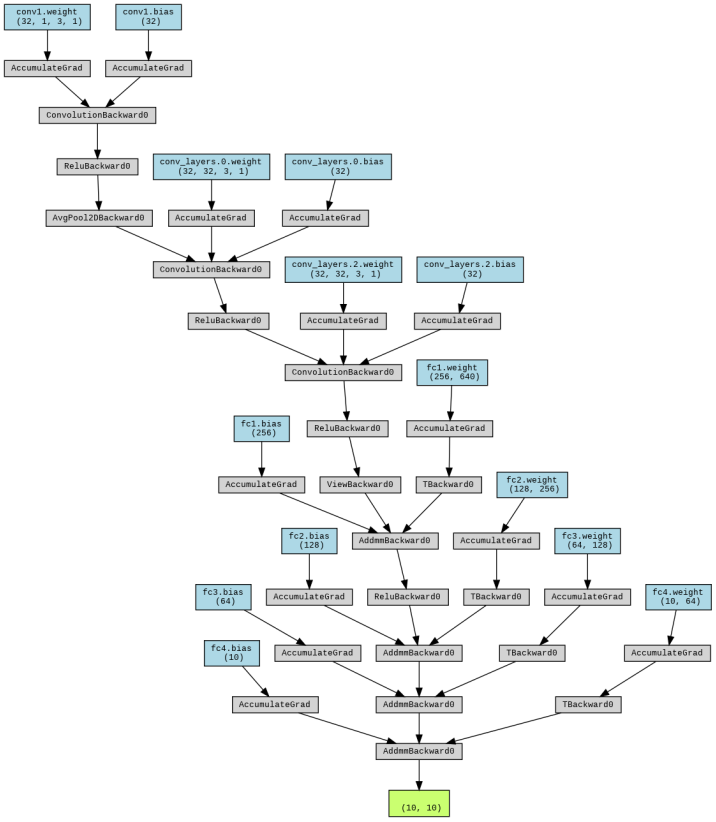

Fig. 2: Visualization of the generated new architecture- Best model #2 after 10 evolutionary rounds

As we can see from the Table I, the first seed is better in terms of performance over the others, and hence is retained as a parent for most of the runs while generating child codes. This is exactly why we can see from Figures 1 , 2 and 3 that the newly generated architectures are similar to the in-context examples generated from Seed 1. The training curves for the same can be seen in Appendix A.

V. Problems Encountered

- Code with Textual Inclusions: LLMs sometimes generate code that includes comments or explanations alongside the actual functional code. This can make it difficult to directly execute the generated code without separating the commentary from the core functionality.
- Miscalculation of Input Features: In some cases, the LLM might incorrectly determine the number of input features for a Fully-Connected (FCN) layer in the architecture. This issue was addressed by dynamically calculating the input shape based on the actual data fed into the model during training. Over time, the LLM learned to adopt this approach and started dynamically calculating input features for the FCN layer. Additionally, using an API for a more advanced LLM (e.g., GPT-4.0) yielded slightly better results in this regard.
- DataLoader Concurrency Issues: When running multiple models concurrently on separate threads, it's not feasible to use the same DataLoader object to feed data to all models simultaneously. This necessitates the LLM to generate code for a new DataLoader instance whenever it creates a fresh architecture.
- Stagnant Evolution: In some scenarios, the LLM might get stuck in a loop, repeatedly generating the same child architecture due to an over-performing predecessor. This stagnation hinders the exploration of diverse architectures.

VI. Conclusion

From the experiments, we can conclude that the performance of the child architectures generated by the LLM, GPT 4 is quite impressive. It has successfully produced diverse and effective neural network architectures. These child architectures have achieved high accuracy on the MNIST dataset, which demonstrates that leveraging LLMs for evolutionary operations such as crossover and mutation, can significantly improve neural network architectures.

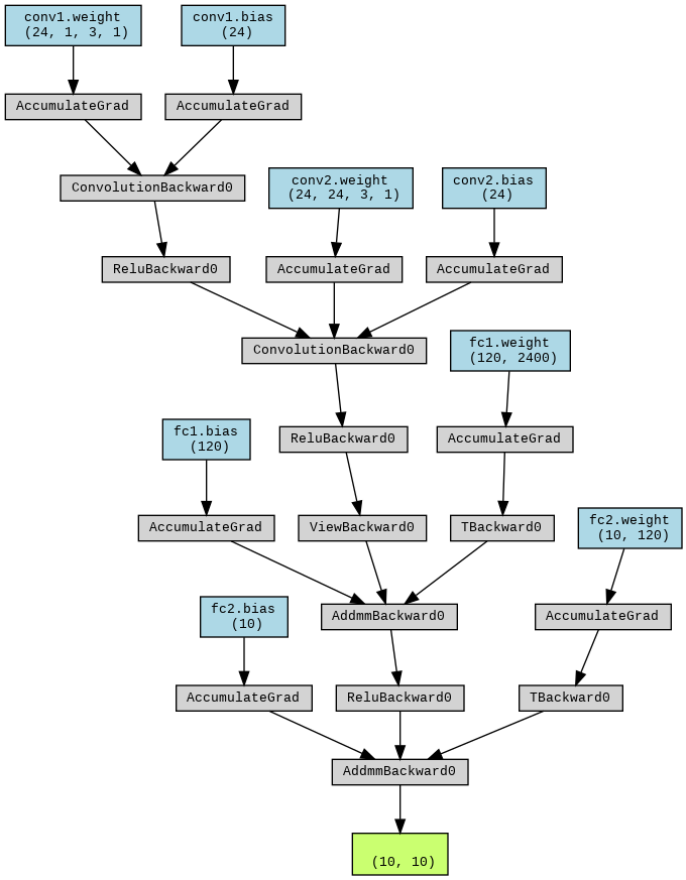

Fig. 3: Visualization of the generated new architecture- Best model #3 after 10 evolutionary rounds

VII. Future Work

Our goal for the next phase is to apply our methodology to the CIFAR-10 dataset. This involves finding the optimal neural network architectures by utilizing a large language model such as GPT-4. To improve model accuracy, we will modify fitness functions and evaluation measures to account for the complexity of CIFAR-10.

We propose integrating feedback methods with program synthesis. Instead of directly generating PyTorch operations, GPT-4 will create programs based on expected scenarios, guiding feedback into neural network layers. This approach will enhance classification accuracy and context awareness by integrating domain knowledge and offering hierarchical task representation.

Appendix

A. *Seeds Used for MNIST-1 Dataset:*

Given below are the seed models (python, pytorch) used :
Seed 1 :

```
class Seed(nn.Module):
    features: int = 32
    nlayer: int = 3

    def __init__(self):
        super(Seed, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=(3, 1))
        self.relu1 = nn.ReLU()
        self.pool1 = nn.AvgPool2d(kernel_size=(2, 1), padding=(1, 0))

        self.conv_layers = nn.ModuleList()
        for _ in range(self.nlayer - 1):
            self.conv_layers.append(nn.Conv2d(in_channels=self.features, out_channels=self.features, kernel_size=(3, 1), stride=1, padding='same'))
            self.conv_layers.append(nn.ReLU())

        self.flatten = nn.Flatten()

        # Calculate output size dynamically after convolutions
        self.conv_out_size = self._get_conv_out_size()
        self.fc1 = nn.Linear(in_features=self.get_prev_out_size[1], out_features=256)  # Dynamic input features
        self.relu2 = nn.ReLU()
        self.fc2 = nn.Linear(in_features=256, out_features=10)


    def __call__(self, x):
        x = x.reshape(10, 1, 40)
        x = x[..., None]  # Add a channel dimension for 2D convolution
        x = self.conv1(x)
        x = self.pool1(x)

        for conv, relu in zip(self.conv_layers[::2], self.conv_layers[1::2]):
            x = conv(x)
            x = relu(x)

        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu2(x)
        x = self.fc2(x)
        return x
```

Seed 2 :

```
class Seed(nn.Module):
    features: int = 25
    # nlayer: int = 3
    def __init__(self):
        super(Seed, self).__init__()
        # Initialize model parameters
        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = self.features, kernel_size=(5,1), stride=(2,1), padding=(1,))
        self.relu1 = nn.ReLU()


        self.conv_layers = nn.ModuleList()
        for _ in range(2):
            self.conv_layers.append(nn.Conv2d(in_channels=self.features, out_channels = self.features,
```

```
    kernel_size=(3,1), stride =(2,1),
    padding = (1,)))
                self.conv_layers.append(nn.
    ReLU())

          self.flatten = nn.Flatten()

         # Calculate output size after
    convolutions dynamically
          self.conv_out_size = self.
    _get_conv_out_size()
          self.fc1 = nn.Linear(
    in_features=self.get_prev_out_size
    [1], out_features=10)


    def __call__(self, x):

        x = x.reshape(10, 1, 40)
        x = x[..., None]  # Add a channel
     dimension for 2D convolution
        x = self.conv1(x)
        x = self.relu1(x)

        for conv, relu in zip(self.
    conv_layers[::2], self.conv_layers
    [1::2]):
            x = conv(x)
            x = relu(x)

        x = self.flatten(x)
        x = self.fc1(x)

        return x
```

Seed 3 :

```

class Seed(nn.Module):
    # features: int = 40
    hidden_size: int = 100
    def __init__(self):
          super(Seed, self).__init__()
          # Initialize model parameters
    here
          self.flatten = nn.Flatten()
          self.fc1 = nn.Linear(
    in_features=40, out_features=self.
    hidden_size)
          self.relu1 = nn.ReLU()
          self.fc2 = nn.Linear(
    in_features=self.hidden_size,
    out_features=self.hidden_size)
          self.fc3 = nn.Linear(
    in_features=self.hidden_size,
    out_features=10)

    def __call__(self, x):

        x = x.reshape(10, 1, 40)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu1(x)
        x = x + self.relu1(self.fc2(x))
        x = self.fc3(x)

        return x
```

B. Training and Validation Accuracy curves for the Seeds:

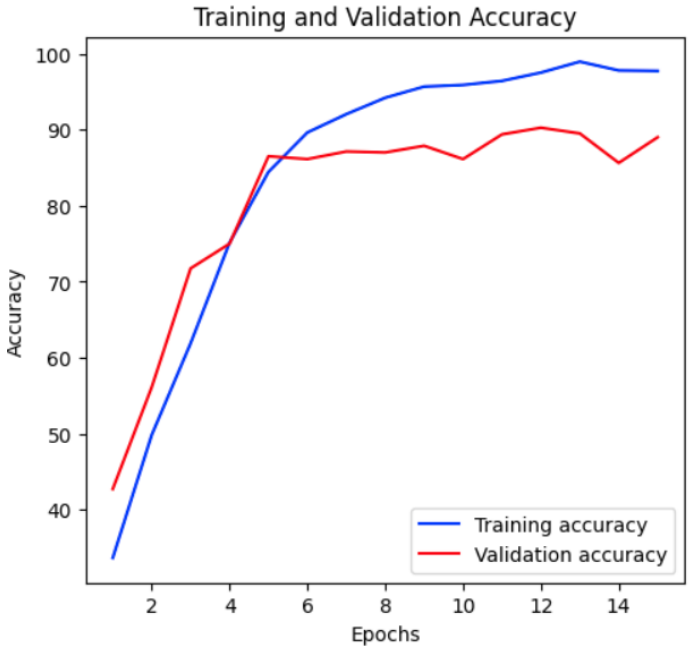

Fig. 4: Training and Validation accuracy curve for Seed 1

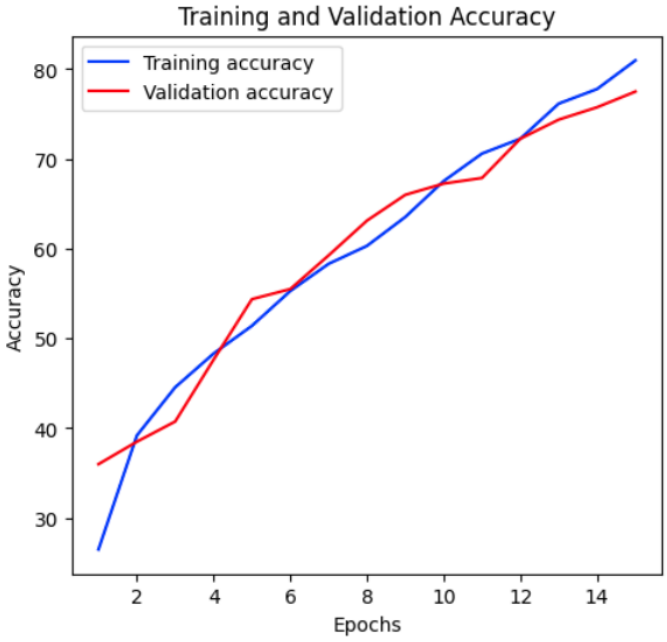

Fig. 5: Training and Validation accuracy curve for Seed 2

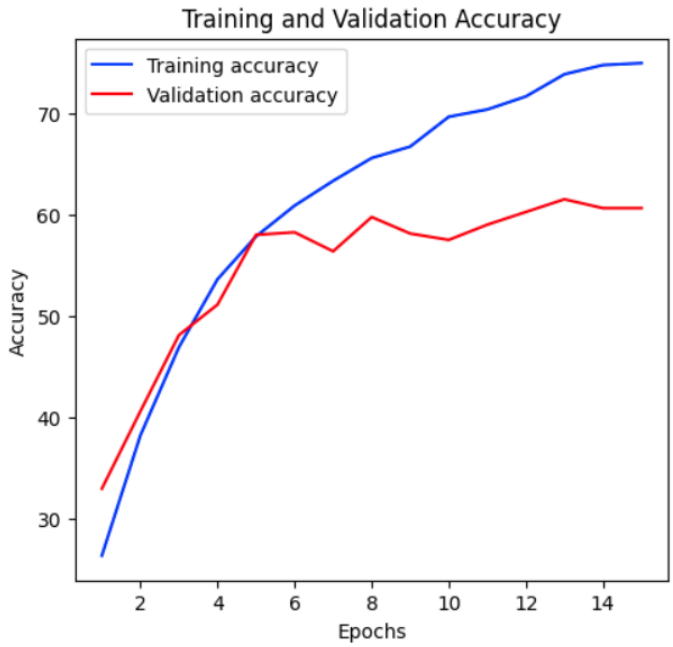

Fig. 6: Training and Validation accuracy curve for Seed 3

C. Training and Validation Accuracy curves for the Child Architectures:

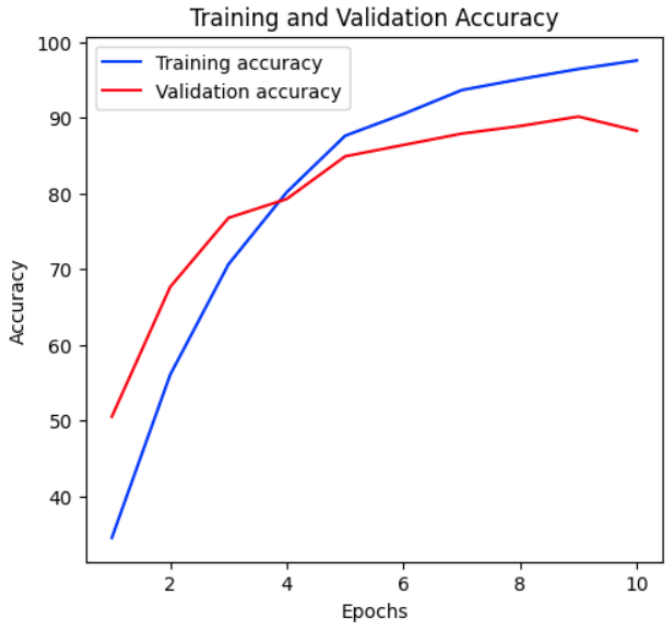

Fig. 7: Training and Validation accuracy curve for Best model #1

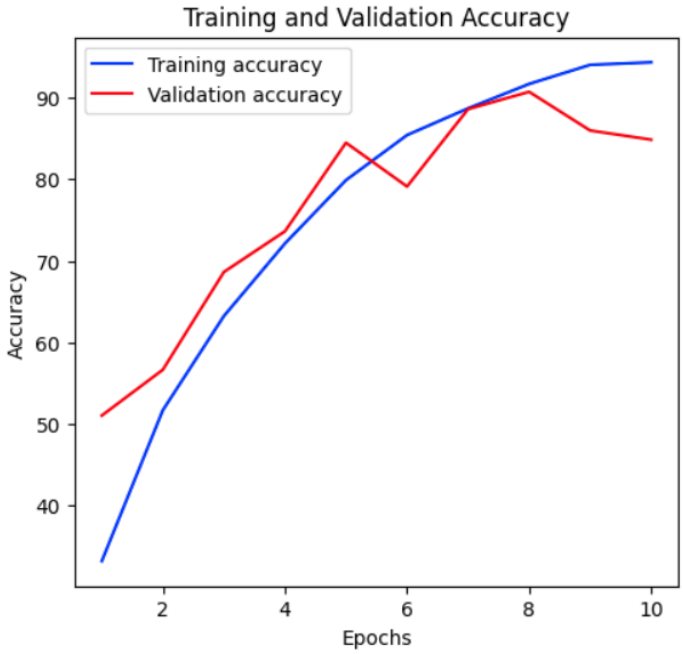

Fig. 8: Training and Validation accuracy curve for Best model #2

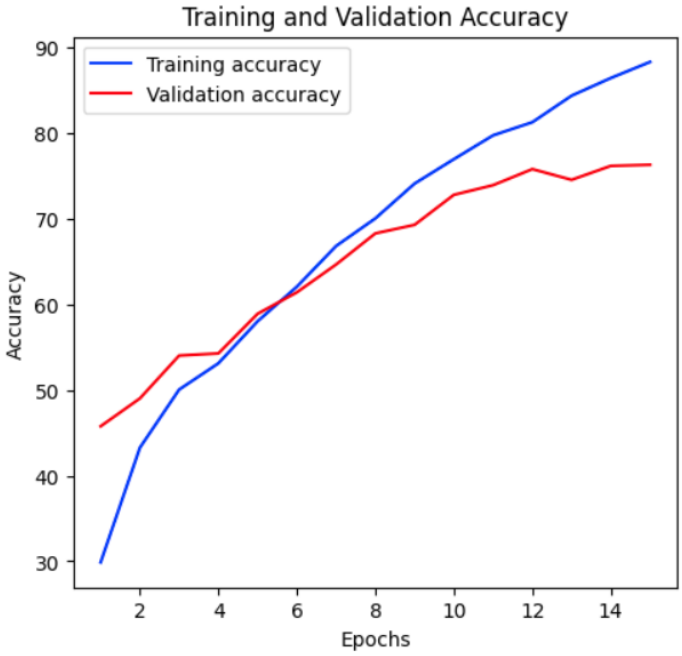

Fig. 9: Training and Validation accuracy curve for Best model #3

REFERENCES

[1] Chen, A., Dohan, D. and So, D., 2024. Evoprompting: Language models for code-level neural architecture search. Advances in Neural Information Processing Systems, 36.

[2] Maynord, M., Dessalene, E.T., Fermuller, C. and Aloimonos, Y., 2022, September. Mid-Vision Feedback. In The Eleventh International Conference on Learning Representations.