Enhancing IBM's Analog-NAS with a Reinforcement Learning NAS Agent

Sujeeth Bhavanam
Columbia University
New York, NY
sb4839@columbia.edu

Tawab Safi
Columbia University
New York, NY
as7092@columbia.edu

***Abstract*—This project introduces an innovative approach to neural architecture search (NAS) for analog in-memory computing (IMC) devices using a Reinforcement Learning (RL) based agent. By integrating a supervised Vector Quantized-Variational AutoEncoder (S-VQVAE), we develop a novel latent search space that is both hardware-aware and optimized for efficient exploration. The RL agent navigates this space, effectively selecting architectures that balance computational efficiency with high performance on the CIFAR-10 dataset under analog constraints. Our method significantly reduces the computational resources and search time by over three times compared to traditional evolutionary algorithms while achieving comparable accuracy, thus demonstrating a scalable and efficient approach to NAS in constrained hardware environments.**

Acknowledgement: Thank you Professor Kaoutar El Maghraoui for the opportunity and Dr Hadjer Benmeziane for your mentorship through the project.

1. Introduction

Recent advancements in neural architecture search (NAS) have primarily focused on optimizing neural network configurations to meet specific hardware constraints, crucial for edge computing environments where power efficiency and low latency are paramount. Analog in-memory computing (IMC) presents a promising solution by leveraging the physical properties of memory devices for computation and storage, significantly enhancing speed and reducing power consumption compared to traditional von Neumann architectures [1].

In this project, we integrate Reinforcement Learning (RL) into NAS to effectively design neural architectures tailored for IBM's analog in-memory computing devices. We develop a methodology that incorporates a supervised Vector Quantized-Variational AutoEncoder (S-VQVAE) to create a novel latent search space that is both hardware-aware and conducive to efficient exploration by an RL agent. This agent navigates this space to identify optimal architectures, utilizing a tailored reward function that evaluates architecture performance on the CIFAR-10 dataset under analog constraints. Our proposed RL agent was able to design neural architectures that were able to achieve a 95 % 1-day accuracy similar to the evolutionary search algorithm used in [1] while utilizing significantly less computational resources, reducing search time by a factor of more than 3.

2. Related Work

The pursuit of optimizing neural architecture search (NAS) using reinforcement learning has been extensively explored in the literature. Zoph and Le [2] initially demonstrated the use of reinforcement learning for designing neural network architectures, setting a foundational approach for subsequent research in this field. Further advancing the capabilities of NAS, Cai et al. [3] introduced an efficient method by leveraging network transformations, allowing for iterative improvements on existing architectures. Concurrently, the incorporation of graph-based methods as seen in the work by Wen et al. [4], where operation embeddings were utilized, signifies a shift towards more complex representations within NAS, enabling a nuanced understanding of architectural components.

Additionally, block-wise generation of neural architectures has also gained traction due to its practical implications. Zhong et al. [5] provided methodologies for block-wise construction, which promote modular and scalable approaches to architecture design. This concept of modularity is further exemplified in the progressive search strategies proposed by Liu et al. [6], which aim to refine the search process incrementally, thereby enhancing efficiency.

Significant to the context of our work is the application of NAS in hardware-specific scenarios. Notably, the use of IBM's analog in-memory hardware acceleration kit, as discussed by Benmezian et al. [7], underscores the practical challenges and solutions associated with deploying NAS in constrained hardware environments. Moreover, the exploration of multi-objective NAS by Hsu et al. [8] introduces an additional layer of complexity by balancing multiple performance metrics, a critical consideration for real-world applications.

Lastly, the comprehensive review by Elsken et al. [9] on the use of reinforcement learning within NAS encapsulates the diverse strategies and outcomes achieved in recent years, offering a critical analysis of the progression and effectiveness of various approaches in this evolving field.

3. Problem Statement

In the quest to optimize neural network architectures on hardware-constrained platforms, particularly simulated analog devices, a significant challenge is to design models that not only perform with high accuracy but also adhere to specific hardware requirements. This report investigates the feasibility of using Reinforcement Learning (RL) to autonomously design neural network architectures, akin to ResNet, that excel on the CIFAR-10 dataset within such constraints. The overarching problem addressed here is: *Can an RL-based agent be constructed to design a ResNet-like architecture optimized for performance on simulated analog devices, achieving high accuracy on the CIFAR-10 dataset?*

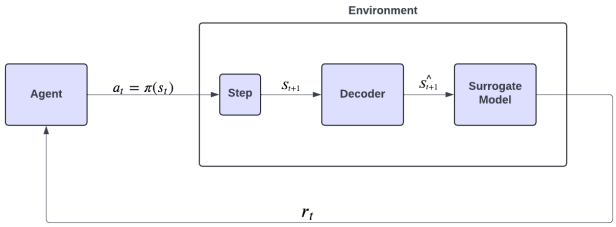

Figure 1: RL-based NAS MDP.

As delineated in Figure 1, our approach employs an RL-based methodology wherein the agent begins with a random network architecture represented as a latent state. Guided by its policy, denoted as $\pi(s_t)$, the agent selects actions (a_t). These actions are inputs to an environment that simulates the deployment of the network architecture on analog hardware. The environment responds with a new state (s_{t+1}) and a reward (r_t), which reflects the change in accuracy due to the modifications proposed by the agent.

The environment consists of three integral components:

- **Step Block**: This component interprets the agent's action, utilizing a learned codebook from a Vector Quantized-Variational AutoEncoder (VQ-VAE) to adjust the network architecture accordingly and generate the next latent state representation (s_{t+1}).
- **Decoder Block**: It decodes the updated latent state into a modified network architecture, providing a tangible representation for evaluation ($\hat{s_{t+1}}$).
- **Surrogate Model Block**: It assesses the performance of the updated architecture by estimating its accuracy on the CIFAR-10 dataset.

4. Methodology

4.1. Architecture Search Space

In this section, we introduce a novel latent search space for the RL agent to explore. This search space helps in reducing the number of neural architecture configurations to search over by searching in an effective search space that was created using data about previously trained architecture representation.

4.1.1. Dataset. The dataset used consists of 1000 neural architecture representations used in [1] and its corresponding 1-day accuracy. It is designed for a neural architecture search (NAS) tailored to analog devices, expanding upon the foundational work of [1]. The primary objective is to establish a novel latent search space that leverages previously trained architectures from the [1] research. To achieve this, a supervised Vector Quantized Variational Autoencoder (S-VQVAE) model is employed, which processes the architectural data to encode and subsequently decode it from a latent representation.

Each architecture in the dataset is defined by a combination of multiple structural elements:

- Main Blocks (M): Each architecture contains several distinct main blocks.
- Residual Blocks (R1, R2, R3, R4, R5): Each main block is composed of multiple residual blocks, which may incorporate skip connections with optional downsampling. Downsampling is accomplished using 1x1 convolution layers to ensure matching input and output sizes.
- Branches (B1, B2, B3, B4, B5): Each residual block can contain several branches, each utilizing a specific type of convolution block. The dataset includes a variety of convolution blocks to cover a broad spectrum of standard architectures like ResNets, ResNext, and Wide ResNets.
- Convolution Blocks: These are denoted by labels (A, B, C, D), where A and B correspond to common ResNet blocks (BottleNeckBlock and BasicBlock, respectively), and C and D represent these blocks with altered order of ReLU and Batch normalization operations and we replace these labels with 1,2,3,4 in the vector representaion respectively .
- Widening Factor (widenfact1, widenfact2, widenfact3, widenfact4, widenfact5): This factor scales the width of the convolution blocks within the residual blocks.

Fig. 2 shows the ranges of the structural elements in the dataset.

Hyper-parameter	Definition	Range
OC_0	First layer's output channel	Discrete Uniform [8, 128]
KS_0	First layer's kernel size	Discrete Uniform [3, 7]
M	Number of main blocks	Discrete Uniform [1, 5]
R*	Number of residual block per main block	Discrete Uniform [1, 16]
B*	Number of branches per main block	Discrete Uniform [1, 12]
CT*	Convolution block type per main block	Uniform Choice [A; B; C; D]
WF*	Widening factor per main block	Uniform [1, 4]

*The hyper-parameter is repeated for each main block. ConvBlock refers to different Conv-Relu-BN blocks.

Figure 2: Searchable hyperparameters and their ranges

In the construction of the dataset for the neural architecture search, each architecture was categorized into one of fourteen accuracy bins ranging from 0.6- to 0.94, based on its 1-day accuracy as seen in Table. 1. This stratification was particularly refined within the 0.9 to 0.95 accuracy range, where approximately 70% of architectures were concentrated, thereby allowing for a more balanced distribution across bins.

TABLE 1: Accuracy Bins for Neural Architecture Search

Bin Label	Accuracy Range
0	(0.6 - 0.65)
1	(0.65 - 0.7)
2	(0.7 - 0.75)
3	(0.75 - 0.8)
4	(0.8 - 0.85)
5	(0.85 - 0.9)
6	(0.9 - 0.905)
7	(0.905 - 0.91)
8	(0.91 - 0.915)
9	(0.915 - 0.92)
10	(0.92 - 0.925)
11	(0.925 - 0.93)
12	(0.93 - 0.935)
13	(0.935 - 0.94)

Finally, we have a 22 dimensional vector representation and a class label based on its 1-day accuracy for each neural archictecture, which we call $\hat{S}$.

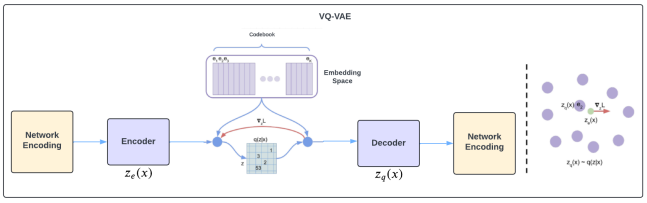

Figure 3: VQ-VAE Architecture.

4.1.2. VQ-VAE. The dataset was employed to train a Vector Quantized Variational Autoencoder (VQ-VAE), which utilized 14 distinct codebook vectors of dimension 8 corresponding to each accuracy bin. VQ-VAEs are a type of generative model that quantizes the continuous latent space to discrete representations using a predefined codebook, facilitating the learning of a categorical latent distribution. We refer [10] for the implementation and the architecture is shown in Fig. 3. The loss function is as follows:

$$\text{MSE}(z_q(x), x) + ||\text{sg}[z_e(x)] - e||^2 + \beta||z_e(x) - \text{sg}[e]||^2 \quad (1)$$

It comprises of three terms: the reconstruction loss, which encourages the decoded representation to match the input; a quantization loss that minimizes the distance between the latent vector and the nearest codebook vector; and a commitment loss, ensuring that the encoded latent vector is close to its assigned codebook vector. However, this standard loss function does not incorporate label information, which is crucial for ensuring that the learned codebooks act as cluster centers for architectures with similar accuracies. As

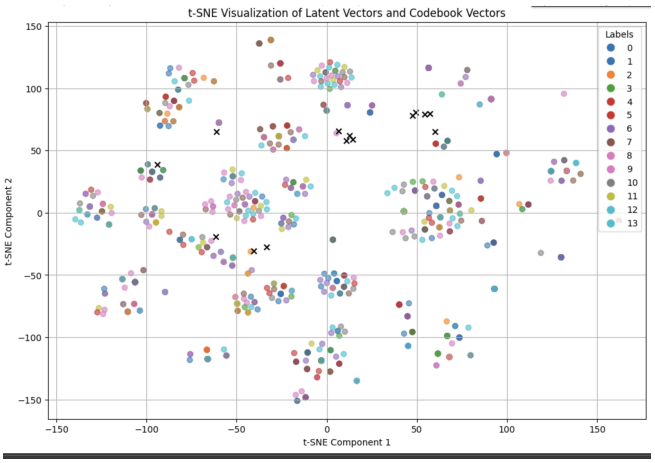

Figure 4: VQ-VAE latent search space.

4.1.3. Supervised VQ-VAE. To ensure the clustering is done properly we modified the loss function to ensure latent vectors with a particular label are pushed to the corresponding codebook and every other point is pushed away. The modifed loss function is as follows:

$$\text{MSE}(z_q(x), x) + ||\text{sg}[z_e(x)] - e||^2 + \beta||z_e(x) - \text{sg}[e]||^2 - I(k \neq y)(||\text{sg}[z_e(x)] - e||^2 + \gamma||z_e(x) - \text{sg}[e]||^2)$$

where, first three terms are same as before, and additional terms are introduced to penalize mismatch between the labels and the closest codebook vectors. The inclusion of these terms ensures that codebook vectors act as effective cluster centers for architectures with similar performance, promoting the aggregation of similar architectures within each defined accuracy bin. This approach not only enhances the model's ability to navigate the architecture space efficiently but also aligns the latent space organization directly with the performance characteristics of the architectures, fostering a more targeted and effective search process.

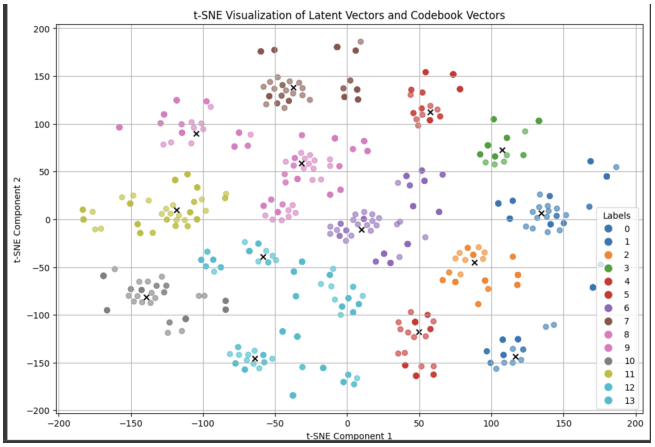

Figure 5: Supervised VQ-VAE latent search space.

4.1.4. Training. To train the supervised VQVAE model, we use a bayesian hyperparameter search to pick the best

hyperparameters for the model to ensure least loss as seen in Fig. 6. Once the training was done for 1000 epochs on T4 GPU for 10 min, we observe a latent search space as seen in Fig. 5 and here we can clearly see that architectures with same labels are clustered together in the latent space, hence creating a very effective latent search space for the RL agent to explore.

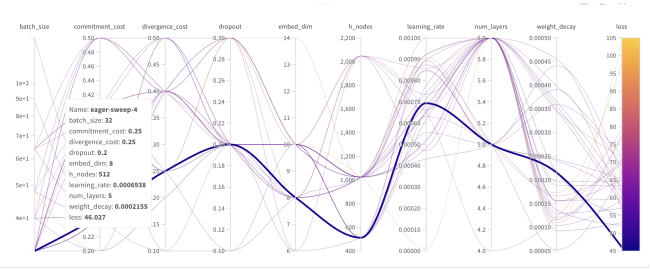

Figure 6: Hyperparameter Search.

4.1.5. Codebook and Decoder. Once the training is done we saved the codebook and decoder weights to use in the RL training.

- **Codebook:** Serves as a compact representation of the latent search space, where each vector acts as a discrete cluster center, encapsulating distinct sets of architectural characteristics. These vectors guide the reinforcement learning (RL) agent's decisions on modifying specific dimensions of the latent state, enabling quantized exploration of the latent architecture space.
- **Decoder:** Reconstructs the modified latent vector back into a higher-dimensional (22) vector $\hat{S}$ that represents a new architectural configuration. This process translates quantized explorations in the latent space into tangible neural network architectures, which are then evaluated by a surrogate model for their performance potential.

4.2. Surrogate Model

Surrogate models are utilized in neural architecture search (NAS) to expedite the evaluation process by approximating the performance of candidate architectures, mitigating the computational expense of training each architecture fully.

In our approach, a surrogate model was trained using an XGBoost algorithm on the dataset defined in the previous section. The model underwent 150 training iterations, achieving a Root Mean Square Error (RMSE) of 0.05. This level of accuracy in prediction mirrors the results reported in the [1], demonstrating the efficiency of our method in predicting the 1-day accuracy of proposed architectures denoted by $\hat{S}$.

4.3. Training the RL Agent

4.3.1. Overview. The Reinforcement Learning (RL) agent is trained to navigate the neural architecture search space represented by a latent state vector. This vector is learned via a Supervised Vector Quantized-Variational AutoEncoder (S-VQVAE) [11]. The agent learns to select optimal codebook vector values that lead to high-performing neural architectures. The following figure represents an overview of the RL agent's training loop:

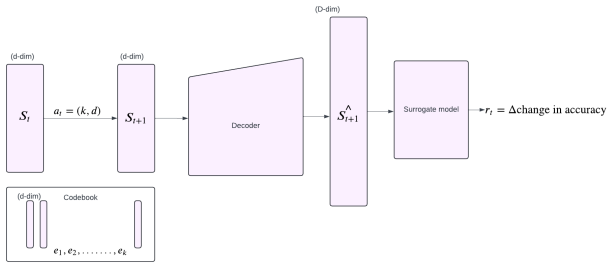

Figure 7: RL agent training flow.

4.3.2. State Space. The state space is a `d_dim` vector, where `d` represents the number of hidden dimensions in the latent representation learned by the Supervised VQVAE. The codebook vectors and latent state vectors share the same dimensionality.

4.3.3. Action Space. The action space consists of a total of `(number of codebooks * d_dim + 1)` actions. Each action is a single value between `0` and `(number of codebooks * d_dim)`, where the last index action represents the terminal action. If the action is not terminal:

- The action is represented as $a_t = (k, t)$, where k is the codebook vector number, and t is the index of the codebook vector that is copied to the t-th index of the state vector.
- The step block in the environment replaces the t-th index of the state vector with the k-th codebook vector's t-th index value to form the next latent state s_{t+1}.

4.3.4. Reward. The reward for an action is the difference in the 1-Day Accuracy between the new state s_{t+1} and the previous state s_t. It reflects the improvement in model accuracy caused by a specific action.

4.3.5. Training Loop. Following is the training loop flow:

1) **Initialization:** Start with a randomly initialized `d_dim` latent state vector s_0.
2) **Action Decision:** Based on the current state s_t, the RL agent selects an action a_t.
   - If the action is terminal, the same state vector is returned, and the trajectory ends.
   - Otherwise, the proposed codebook vector and its index value are identified and swapped with the corresponding value in the latent state vector to produce a new state s_{t+1}.

3) **Decoding:** Pass the new latent state vector through the decoder to produce a high-dimensional `D_dim` representation.
4) **Evaluation:** Use the surrogate model to predict the 1-Day Accuracy for the architecture, and calculate the change in accuracy from the previous state.
5) **Reward Assignment:** Pass the change in accuracy as a reward to the RL agent.
6) **Loop Continuation:** Repeat the process until:
    - The maximum allowed steps in a trajectory are reached, or
    - The agent selects the terminal action.

Note: In our implementation, there are 14 learned codebooks, each represented as an 8-dimensional vector. The Decoder converts the 8-dimensional latent state vector to a 22-dimensional state vector that the surrogate model can understand.

4.3.6. Models Trained. We trained the following RL agents:

1) **PPO MLP Policy:**
    - The agent uses Proximal Policy Optimization (PPO)[12], an actor-critic network that clips the updates done to the actor network for stability and faster convergence.
    - The actor predicts an action given a state, while the critic predicts the value (total expected return) of a state.
    - Both the actor and critic networks use a separate underlying Multi-Layer Perceptron (MLP) for their predictions.
    - Only the latent state vector is given as input to the RL agent.
2) **PPO Multi-Input Policy (with memory of 6 previous actions):**
    - The PPO agent uses the same underlying architecture as the MLP Policy, with an additional input: an action buffer containing the past 6 actions taken.
    - The state passed to the model consists of:
        a) The state latent vector, and
        b) The action buffer.
    - This additional memory is expected to help the RL agent learn better strategies by understanding the previous actions that led to the current state.

5. Experimental Results

The RL agents were implemented using Stable Baselines 3 [13]. The following parameters were used for training our agents:

- All agents were trained for a total of 1,500,000 steps.
- At each actor and critic update step, the updated policy is used to create a rollout of 2048 steps. The collected trajectories in 2048 steps are used to update the RL agent for 12 epochs with a batch size of 32. This process is continued until the total of 1,500,000 steps is reached, which is roughly 732 rollouts.

5.1. Initial Exploration of Search Space

We visualize the exploration behavior of the RL agents using t-SNE plots. The render graphs for PPO `MlpPolicy` and PPO `MultiInputPolicy` show how the agent navigates the architecture search space in the initial phase of training for 256 steps. Each color in the graph indicates a different trajectory, the start of a trajectory is indicated by a green dot, and the end of a trajectory is indicated by a red cross.

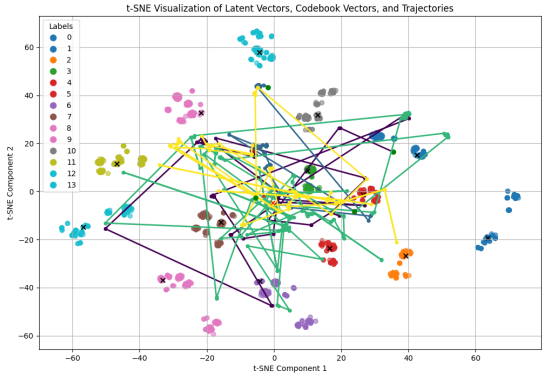

Figure 8: PPO MlpPolicy Initial Exploration.

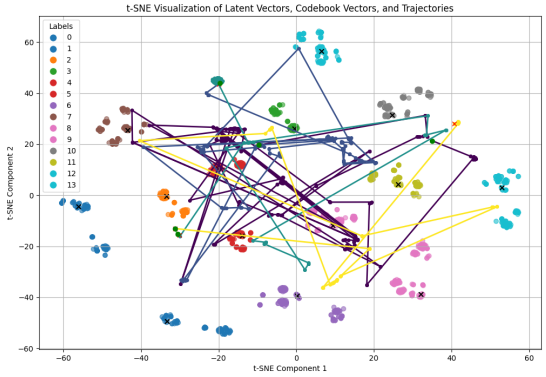

Figure 9: PPO MultiInputPolicy Initial Exploration.

Both models appear to effectively explore the architecture search space during the initial phase of training. This exploration is crucial for the model to learn how to navigate the search space to design architectures with high accuracy.

5.2. Performance During Training

The graphs below show the mean episode reward per rollout (`rollout/ep_rew_mean`) and the mean episode length per rollout (`rollout/ep_len_mean`) for the PPO `MlpPolicy` and PPO `MultiInputPolicy` during the training phase.

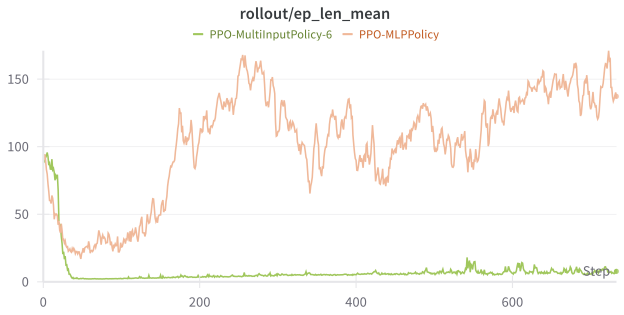

Figure 10: Average Episode Length per Rollout

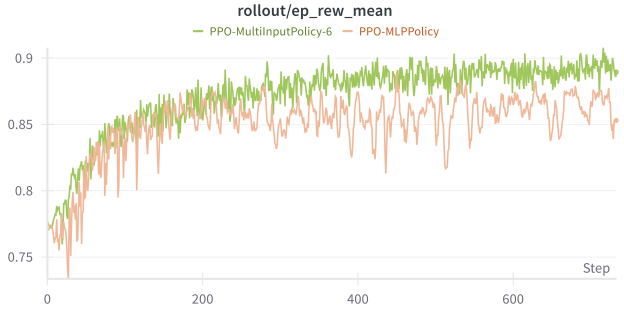

Figure 11: Average Episode Reward per Rollout

- Both models achieve an average episode reward per rollout of over 85% for the architectures they design towards the end of training. The PPO `MultiInputPolicy` performs better than PPO `MlpPolicy`, achieving a higher average episode reward of around 90% per rollout.
- The PPO `MultiInputPolicy` designs architectures with higher accuracy than the PPO `MlpPolicy` while using significantly fewer steps. Towards the end of training, the `MlpPolicy` has an average episode length of 130 per rollout, while the `MultiInputPolicy` only takes an average of around 5 steps per episode while designing architectures with higher accuracy. This demonstrates that having a memory of previous actions helps the model learn efficiently and design better architectures in fewer steps.
- Taking fewer steps to generate a trajectory (episode) during inference time allows the model to search faster, as it can roll out more trajectories to find the optimal network in the same amount of time.

5.3. Performance During Inference

Both PPO `MultiInputPolicy` and PPO `MlpPolicy` agents find models with an accuracy of around 95% (based on the Surrogate Model) after 10 minutes of search. Our RL agent finds architectures with similar accuracy as the evolutionary search algorithm used in the IBM's Analog NAS paper while using significantly fewer resources. Both RL agents were run on a local CPU for 10 minutes each, which is faster than the evolutionary algorithm that took around 30 minutes on a GPU.

The experimental results demonstrate the effectiveness of using Reinforcement Learning for Neural Architecture Search on analog devices. Both PPO `MlpPolicy` and PPO `MultiInputPolicy` agents are able to find architectures with high accuracy, with significantly less computational resources than evolutionary search. The `MultiInputPolicy` agent outperforms the `MlpPolicy` agent due to its ability to retain memory of previous actions.

6. Conclusion

In this study, we presented an innovative approach to neural architecture search (NAS) for analog in-memory computing devices, utilizing a reinforcement learning (RL) agent along with a supervised Vector Quantized-Variational Autoencoder (S-VQVAE). Our methodology capitalized on the novel latent search space generated through the S-VQVAE, which effectively captures the meaningful representaions of neural architectures tailored for analog hardware constraints. The RL agent, trained within this structured latent space, demonstrated a significant reduction in search time and computational resource usage while maintaining high accuracy in architecture selection.

Key contributions of our work include the integration of S-VQVAE to encode and decode neural architectures into a latent space, ensuring that similar architectures cluster around specific codebooks. This setup facilitated a more directed and efficient search by the RL agent, showcasing the potential of combining VQVAE and RL for hardware-aware NAS tasks. The agent's performance, achieving architectures with up to 95% accuracy on the CIFAR-10 dataset within minimized computational budgets, underscores the practicality of our approach against traditional methods.

Moreover, our results affirm the feasibility of using RL to automate the design of efficient and effective neural network architectures for specialized hardware, offering a robust alternative to evolutionary and other manual search techniques. This research not only advances the field of NAS but also opens new avenues for future work focusing on the scalability and adaptability of the proposed methods to various other domains and hardware configurations.

7. Future Work

In collaboration with our research mentor, Dr. Hadjer Benmezian at IBM, we plan to continue improving and

expanding our project over the summer, 2024. Our future work will focus on the following aspects:

1) **Benchmark Evaluation:** To thoroughly assess the performance of our VQ-VAE-based RL agent approach, we intend to test it against NAS-Bench-101 and other widely used NAS benchmark datasets. This evaluation will offer a comprehensive comparison with existing NAS techniques and highlight the efficacy of our proposed method.
2) **Graph-Based Representation:** Instead of relying on naive D dimensional architecture representation, which may not fully capture the structural intricacies of neural networks, we plan to represent neural architectures using graph-based encodings. By training the VQ-VAE on these graph representations, we aim to achieve a more expressive latent space, providing the RL agent with richer architectural features to explore.
3) **Enhanced Surrogate Model and Analog Simulation Accuracy:** The current Surrogate Model offers a preliminary estimate of the neural architecture's performance on simulated analog devices. Moving forward, we will refine and improve this model to provide more accurate predictions. Additionally, we will rigorously test the accuracy of the models designed by our RL agent on IBM's analog device simulation to validate their real-world applicability.

Through these enhancements and evaluations, we hope to further establish the effectiveness of our VQ-VAE-based RL agent approach for neural architecture search on analog devices.

8. Contributions

UNI	as7092	sb4839
Name	Tawab	Sujeeth
Contribution fraction	50 %	50 %
What I did 1	Custom Environment	VQ-VAE
What I did 2	RL agents	RL agents
What I did 3	Presentation	Presentation
What I did 4	Report	Report

9. References

[1] H. Benmezian *et al.*, "Analognas: A neural network design framework for accurate inference with analog in-memory computing," *arXiv preprint arXiv:2305.10459*, 2023.

[2] B. Zoph and Q. V. Le, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[3] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," *arXiv preprint arXiv:1707.04873*, 2017.

[4] H. Wen, S. Han, A. Liu, S. Singhal, and T. Arodz, "Graph-based neural architecture search with operation embeddings," *arXiv preprint arXiv:2105.04885*, 2021.

[5] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Practical block-wise neural network architecture generation," *arXiv preprint arXiv:1708.05552*, 2017.

[6] C. Liu, B. Zoph, M. Neumann, *et al.*, "Progressive neural architecture search," *arXiv preprint arXiv:1712.00559*, 2017.

[7] H. Benmezian *et al.*, "Using the ibm analog in-memory hardware acceleration kit for neural network training and inference," *arXiv preprint arXiv:2307.09357*, 2023.

[8] C.-H. Hsu, S.-H. Chang, D.-C. Juan, *et al.*, "Monas: Multi-objective neural architecture search," *arXiv preprint arXiv:1806.10332*, 2018.

[9] T. Elsken, J. H. Metzen, and F. Hutter, "Reinforcement learning for neural architecture search: A review," *Robotics and Autonomous Systems*, 2019. DOI: https://doi.org/10.1016/j.robot.2019.05.013.

[10] A. van den Oord, O. Vinyals, *et al.*, "Neural discrete representation learning," *arXiv preprint arXiv:1711.00937v2*, 2017.

[11] A. Razavi, A. van den Oord, and O. Vinyals, "Supervised vector quantized variational autoencoder," *arXiv preprint arXiv:1909.11124*, 2019.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[13] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, pp. 1–8, 2021.