Neural Architecture Search for Policy Learning Models

Ryan King
Computer Science
Texas A&M University
College Station, Texas 77843
Email: kingrc15@tamu.edu

Aditya Thagarthi Arun
Computer Science
Texas A&M University
College Station, Texas 77840
Email: adityata@tamu.edu

***Abstract*—Household robots have become more widely available and popular due to their compact size and low cost. To continue to drive down the cost of personal robots, software that guides these robots needs to be more efficient and powerful. Deep Reinforcement learning has been used to learn optimal policies for robots that need to navigate through complex environments. However, these models usually require a large number of parameters. Recent methods in Neural Architecture Search have been able to automatically find efficient model architectures that achieve near state-of-the-art results with a fraction of the parameters of hand crafted models. In this paper we explore two different methods of search for neural architectures: evolutionary search and splitting steepest descent. We train these models in the Deepmind Lab environment and evaluate their ability to obtain rewards within their environment. We report the ability of a model to obtain rewards as the average reward accumulated over 1000 iterations through it's environment along with the number of parameters in the models resulting from each method.**

I. Introduction

Many real world agents need to learn to explore on their own without rewards. For example, a Roomba vacuum cleaner could be given the task of getting a house as clean as possible in as little time as possible. A policy that could be used would be to clean the areas with the highest density of dirt first but the Roomba has no prior knowledge about the location of the highest density of dirt in a house without first exploring it. In curiosity driven exploration (1) real world external rewards can be infrequent or absent. So in place of real world rewards, Reinforcement Learning attempts to learn an optimal policy with the use of a utility function. Reinforcement learning techniques have been used in autonomous driving (2) have used reinforcement learning to train deep neural networks to learn optimal policy for autonomous vehicles. Similar policy learning methods exist in the development of AI (3)(1) where an agent learns to navigate their environment by optimizing a utility function. While all of these methods leverage the ability of neural networks to learn complex tasks, the size of these models and the computational cost to use them can be large. This becomes a bigger issue when a policy making model is embedded in a system. What is needed is the smallest possible architecture to develop a policy.

Neural Architecture Search (NAS) is a field of Automated Machine Learning (AutoML) that attempts to automatically find the best neural architecture for some data. Recent methods in NAS (4) have shown that NAS can outperform handcrafted neural architectures with smaller model sizes. This has proven to be very successful in embedded systems where memory storage and computational resources are limited. One method of learning neural architectures is through Neural Evolution (NE). NE has been proposed in (5) where an agent is trained to use the controls of a vehicle to move around a track. Training is conducted with a deep neural network whose architecture is determined based on an evolutionary algorithm. In an evolutionary algorithm, the architecture of the network is embedded as a string which is called a gene. At the beginning of the algorithm, a set of candidate genes and architectures are created and trained on a specified task. The best performing genes are kept while the rest are discarded. New genes are added to the pool through a combination of splicing and mutating successful genes.

Additionally, architecture splitting (6) has shown a method of continually developing neural architectures through an iterative process of training and splitting. They show that a network that has converged to a local or global minimum may be at a higher dimensional saddle point. They escape these saddle points by creating a network morphic structure (7) of their network that allows each node to be represented by two nodes while producing the same output for all inputs. This allows the forward pass through the network to remain the same while representing the network in a higher dimension. With this network morphism, they are able to calculate a splitting indicator of the new network by computing the minimum eigenvalue of a semi-Hessian matrix they call the splitting matrix. The minimum eigenvalue represents the concavity at each point in the new network where a minimum eigenvalue below 0 represents a saddle point and an eigenvalue greater than 0 represents a minimum. This iterative process of training and splitting provides a method of learning the optimal structure.

Following work done in reinforcement learning and NAS, we used NAS to find optimal neural architectures for policy learning. Each model was trained to collect rewards in a maze in the shortest amount of time. We made use of Deepmind's Lab (8) to create an environment for our machine learning models, this game-like program is made specifically for agent based AI research. We used an the Actor-Critic model from the Asynchronous Advantage Actor-Critic (A3C) (9) research

to learn policies. We explored two different approaches to developing a neural architecture in a reinforcement setting: Evolutionary and progressive splitting. We evaluated each model's performance and size as the number of parameters in the model.

II. Related Works

A. Reinforcement Learning

An actor-critic model is a reinforcement learning technique, it consists of two networks an actor and a critic(9). The Advantage function calculates the agent's prediction error. The actor chooses an action at each time step and the critic estimates the action quality or state value. This allows the critic network to learn what states are better or worse, the actor uses this information to teach the agent(by updating the policy distribution) to seek better quality states. An extension of this is the asynchronous advantage actor-critic model(A3C). The A3C model leverages parallel environments, it parallel trains multiple workers, these workers work together to update a global policy distribution.

B. Neural Architecture Search

1) Evolutionary: Evolutionary models are genetic algorithms, they are very effective at finding and evolving unique artificial neural architectures but they require retraining models from scratch across the configuration space(10). Using this method the researchers were able to outperform the best fixed topology neural networks on challenging reinforcement learning tasks. Principles of biology are heavily mirrored through the use of methods such as selection(of well performing architectures), crossover and mutations used on successful model to potentially make even better models. Mutations for instance can either mutate existing connections or add entirely new structures to a network.

2) Splitting: While evolutionary algorithms are very successful at finding unique solutions, the search for those solutions requires massive amounts of computational resource both in the form of compute power and time. Methods in the field of NAS have proposed other methods (6; 7) of finding optimal neural architectures without the need to retrain networks from scratch. In (6), it is shown that as models train, their parameters converge to local minimum which may be high dimensional saddle points. In order to escape these saddle points, first a network morphic model G must be created for model M such that

$$M : x \to z, G : x \to z, \forall x$$

.

This network morphism is created so that the low dimensional model parameters can be represented in a higher dimensional space. In (7), they show that linear layers with an equal input and output size can be added to a model to create a network morphism when the weights of the linear layers are the identity matrix and the biases are 0.

III. Methods

A. Base Model

We use the Actor-Critic model from (11). The pipeline for this model consists of a set of convolutional layers that are used to process the input images, an LSTM that takes the output of the convolutional layers and finally two linear layer which task the output of the LSTM as an input. The first linear layer is called the actor which predicts the next action that the agent should take given the current input. Outputs of this layer are passed into a softmax function which transforms the logits into probabilities of each action the agent can take. The second linear layer is called the critic. This layer attempts to predict the reward that will be gained after the next action.

The agents world is initialized along with the state of the agent. The agent then has a fixed number of actions that are determined by the current learned policy model. The entropy of the probability outputs from the actor, the actual reward and the perceived award are all used to calculate the loss at each step. The gradients from these loss are calculated and accumulated to update the model after a fixed period of time.

B. Evolutionary

1) Crossover: We implemented crossover in a similar manner to (10) or any genetic algorithm for that matter. We take the top two best performing models from our pool, the configurations of these two models are swapped and recombined to produce a child model with a high probability. We then adopted this technique to a reinforcement learning setting.

2) Mutation: In mutation we are are only concerned with making changes to a given network. With a lower probability than crossover we pick an aspect of a network's configuration and we randomly change it, for instance, we may randomly change the activation functions of the network. Mutation is only done on randomly configured new models. In contrast to (10) we are only changing the existing structures of our network, we are not adding new structures due to the lack of computational resources.

C. Network Morphism

As an alternative to iteratively testing new configurations from scratch, we attempt to add new layers to our model similar to methods discussed in our related works and in (7). We decide to add a fully connected identity layer to our model between the output of our LSTM and the actor and critic layers. We initialize this layer with the weights set as the identity matrix and the biases as 0.

IV. Experiments

A. Experiment Setup

We use the Deepmind Lab (8) to create our environment. We use a static maze map with random rewards spread throughout the map. Each reward in the map is worth 1 point and a random number of rewards can be found in each iteration. An example of the maze environment can be found in Fig. reffig:maze.

Method	Average Reward	Params (M)
Baseline	0.85	0.846152
Splitting	0.826	1.10932
Evolutionary	0.69	0.083656

TABLE I

WE REPORT THE RESULTS OF EACH METHOD AS THE AVERAGE REWARD FROM THE LAST 1000 ITERATIONS AND THE NUMBER OF PARAMETERS IN THE MODEL. THE PARAMETERS ARE COUNTED IN MILLIONS.

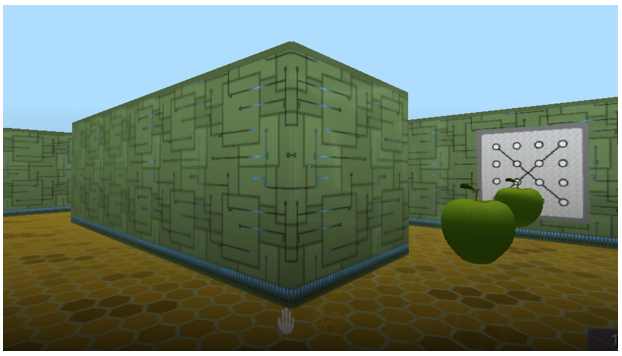

Fig. 1. Maze with rewards(apples) and goal

We create our actor-critic model using PyTorch (12) utilizing only a CPU. We use the actor-critic model from (11) with the addition of a 3 filter 3x3 convolutional layer with a stride of 2 and a 3x3 max pooling layer with stride of one to reduce the computational cost of the large input images. Though Deepmind Lab provides many different inputs including range between the agent and it's obstacles, we choose to give our model only the visual input pictured in Fig 1. Since the agent is only receiving information about what is directly in front of it, we allow our actor model to choose between 3 options: look left, look right and move forward.

To score models, we measure the average reward over the previous 1000 iterations through the maze after training for 10000 iterations. We additionally record the size of the model as the number of parameters in the model. We report the results of each method in Table I. In the evolutionary setting, we select the top performing configuration for evaluation. We compare these methods to a baseline actor-critic model that follows (9) with the addition of our initial convolutional and max pooling layers. In following section we discuss the implementation and results of the splitting and evolutionary methods.

B. Splitting

We added new layers to our model as discusses in our methods section. New layers are added every 2000 iterations through the environment. The results of this method can be found in Table I. We notice that the results from this method seem to perform the worst. This could indicate that the model needs further fine tuning of the parameters that it currently had before new parameters could be added to the model. This would mean that the model had not completely converged before the new layer was added. However, the losses in this setting vary by large amounts due to the random setup of the maze.

Parameter	Default Configuration	Search Space
LSTM Hidden Size	256	32, 64, 128, 256
# Convolutional Filters	32	32, 64, 128
Activations	ELU	ReLU, ELU, Softplus

TABLE II

THE CONFIGURATIONS USED FOR OUR EXPERIMENTS. THE DEFAULT CONFIGURATION IS USED FOR THE BASELINE AND SPLITTING SETTING. THE EVOLUTIONARY SETTING USES THE SEARCH SPACE IN THE FAR RIGHT COLUMN TO GERENATE MODELS.

C. Evolutionary

We using a standard tournament style evolutionary algorithm to different configurations of our model. We start with a pool of 10 random configurations sampled from our configuration space found in in II. Each model is trained for 2000 iterations through our static maze. The average reward gathered from the last 1000 iterations is used to score each of the models. We select the top-k configurations to keep in the model pool after each evolutionary iteration. For these experiments we use the top-3 model configurations. We create the remain configurations using the top-2 model configurations with a combination of configuration mutation and crossovers. We set the probability of mutation at 25% and the probability of crossover at 90%.

This algorithm can be run indefinitely. However, due to time restrictions, we choose to allow this method to run for a week. We take the top configuration at the end of the search and train it for 10,000 iterations. We record the models average reward accumulated over the last 1,000 iterations and report the results along with the model size as the number of parameters in Table I. From this experiment we obtained the following top configuration:

- LSTM Hidden Size: 32
- Number of Convolutional Filters: 16
- Activations: ReLU

We notice that both the LSTM hidden size and the number of convolutional filters is lower than the default configuration. However, this configuration performance worse than the default configuration when it comes to the average reward gathered. This may indicate that a smaller model will train faster resulting in a higher reward in a smaller amount of time such as our 2000 iterations per configuration. However, this may not be a good indicator of the performance of a model trained for a longer period of time. This implies that each configuration of the models need to be trained for longer periods of time. Given the time constraint we placed on our search, this would result in fewer configurations being evaluated.

V. CONCLUSION

In this paper, we explore different methods of automatically developing neural architectures trained using reinforcement learning. These experiments are conducted in hopes of creating a lightweight policy making model for embedded systems where robot must explore an unknown environment. Specifically, we adapt the architecture of an actor-critic model

trained to gather rewards in a maze with 2 methods of neural architecture search. The first is an evolutionary search through a set search space. The second is through an iterative process of training the model and adding new layers.

Our experiments show that evolutionary search methods result in model with significantly fewer parameters resulting in a smaller model than the baseline actor-critic model. However, we show that this expected reward from this search is smaller than that of the baseline and splitting method.

Our experiments in the splitting setting show that models have the potential to continue learning with the addition of new layers. However, new layers must be added after a model has been trained to convergence. Additionally, this method results in larger architectures than the baseline simply due to the nature of the growing. Future work could explore methods of growing a models architecture within a resource constrained environment or additionally prune the network to prevent the network from growing too large.

VI. References

References:

[1] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," 2017.

[2] A. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, p. 70–76, Jan 2017. [Online]. Available: http://dx.doi.org/10.2352/ISSN.2470-1173.2017.19.AVM-023

[3] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, D. Kumaran, and R. Hadsell, "Learning to navigate in complex environments," 2017.

[4] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," 2021.

[5] J. Koutník, G. Cuccu, J. Schmidhuber, and F. J. Gomez, "Evolving large-scale neural networks for vision-based reinforcement learning," in *GECCO '13*, 2013.

[6] Q. Liu, L. Wu, and D. Wang, "Splitting steepest descent for growing neural architectures," 2019.

[7] T. Wei, C. Wang, Y. Rui, and C. W. Chen, "Network morphism," 2016.

[8] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, J. Schrittwieser, K. Anderson, S. York, M. Cant, A. Cain, A. Bolton, S. Gaffney, H. King, D. Hassabis, S. Legg, and S. Petersen, "Deepmind lab," 2016.

[9] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[10] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf