Evolving Optimal Hyperparamters: Learning Enhanced Model Training

Braden Eichmeier
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Email: beichmei@andrew.cmu.edu

Shaun Ryer
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Email: sryer@andrew.cmu.edu

Stefan Zhu
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Email: zheyaoz@andrew.cmu.edu

***Abstract*—We explore a method to effectively select optimal hyperparameters for a general learning task to reduce model training time. Randomly selecting hyperparameters is time-consuming and often ineffective. This paper explores using genetic algorithm (GA) exploration of the hyperparameter search space to improve parameter selection in learning tasks. We compare the effectiveness of the GA exploration compared to a random, grid search, or Latin hypercube selection of hyperparameters. We also developed a novel way to keep under-performing genes around by using a neural network as a method of introducing "invasive species" into the genetic algorithm optimization. This helps encourage exploration in the genetic algorithm towards lower loss values and increased performance in non-deterministic environments. Using this system we were able to get results even in more random settings such as parameter selection within a reinforcement learning (RL) framework.**

I. Introduction

Hyperparameter selection affects a wide array of robotics problems including machine learning, control gains, feature detection in computer vision, and many others. User-defined parameters selected before algorithm execution heavily affect the overall success of these algorithms. In many settings, improper hyperparameter selection leads to complete failure of the system. As a consequence, a large amount of time and energy goes into selecting hyperparameters by the user. Typical methods to select hyperparameters involve random searching [1], "rules of thumb" [8], domain knowledge, or brute force grid searching. Each of these methods can produce sufficiently acceptable results given enough time. Recent concerns in response to the use and consumption of energy have questioned the ethics of training complex data models [20]. Developing methods to improve the hyperparameter tuning process is thus an important field for both the individual researcher and the community at large.

Effectively searching the hyperparameter space falls within two major categories: sampling techniques and learning techniques. Sampling techniques involve selecting individual sets of hyperparameters independent, or mostly independent, of one another. Learning techniques, on the other hand, use the knowledge gained from previous samples to effectively select future samples.

We propose combining the two search categories to alleviate their individual shortcomings. In this method, the sampling methods will sparsely search the parameter space to gain a rough understanding of the space. After generating an initial sample set, a learning algorithm will be used to effectively explore the spacing between samples. We use a genetic algorithm (GA) as the learning agent to compare the effectiveness of the three described sampling techniques (grid search, random search, Latin hypercube search).

Genetic algorithms are a form of search-based optimization algorithm inspired by genetic mutation in evolutionary biology [9]. The base genetic algorithm mimics nature by simulating natural selection within a population of samples. Given a population, each sample is evaluated in terms of a fitness function. Low performing samples are discarded while the high performing samples are selected for reproduction. The next generation of samples is instantiated by performing a gene crossover from multiple samples in the previous generation. Random alterations in the new sample's genes simulate gene mutation to further explore the search space.

In addition to exploring the effectiveness of various sampling techniques, we also explore various methods to improve GA performance and reduce hyperparameter learning. Though this framework can be applied to any general learning problem, we verify the performance of the automated tuning process in various OpenAI Gym environments [2].

II. Related Works

A. Tuning with Sampling Techniques

Sampling techniques involve methods that do not use previous samples to focus the search space. These methods are commonly implemented for hyperparameter tuning due to their simplicity. Sampling methods explore the search space using either uniform or stochastic methods. More complex sampling techniques use a mixture of these approaches to more effectively represent the entire space with fewer samples. An example of a uniform method is a grid search, which tests all pairs of parameters. In high dimensional spaces, this method quickly becomes ineffective [5]. This problem is exacerbated if the hyperparameters are continuous and therefore provide infinite possibilities. To combat the problems of uniform sampling, a common approach is to randomly sample the search space. Users must select a limit to the number of parameter sets in these problematic settings. If the number

is too low, the model runs the risk of not training within an optimal reason of the search space. Too high a number can drastically increase training time. Other methods, such as Latin hypercube sampling and Sobol sequences [13], attempt to maximize the spacing between seemingly random samples.

B. Tuning with Learning

Learning methods, in contrast to sampling methods, use previous data to influence the sampling for future parameters. Recent reviews of hyperparameter tuning detail the use of "particle swarm optimization, genetic algorithms, coupled simulated annealing and racing algorithms" [5]. Learning models can be ineffective for hyperparameter optimization when they require large amounts of data to converge on a solution due to the cost of computing the objective function. Another problem of learning methods stems from seeding the algorithm. Several algorithms, such as particle swarm optimization, genetic algorithms, and gradient-based approaches are dependant on the starting sample(s). Selecting a poor starting sample can lead to an inefficient exploration of the space and susceptibility to local minima.

Recent literature has shown success for GAs tuning hyperparameters in deep learning. Examples include image recognition [26] [25], mineral concentration estimation [22], and online news popularity prediction [24]. Each of these methods used either artificial neural networks (ANNs) or convolutional neural networks (CNNs). In other training applications, GAs have replaced gradient descent as the optimization method to tune the parameters in ANNs [11].

C. GAs and RL

In the field of reinforcement learning, GAs have been used to tune learnable parameters [15]. Applied examples of GAs and RL include traffic light management [14], and replacing back-propagation in deep reinforcement learning benchmarks (such as Atari) [21]. Recent work in this field performed hyperparameter selection in Deep Reinforcement Learning (DRL) for a manipulation task [17]. However, their work only selected the initial population with random sampling and did not consider the effects of other sampling techniques. Another interesting paper in this field uses reinforcement learning to improve the performance of a GA in the mutation and crossover functions [4].

D. Improving GA Performance

Extensive research has been performed in improving GA convergence. The research focuses on improving the four primary GA operators: encoding [12], selection [10], crossover [18], and mutation [3]. Encoding is representing the gene structure of the parameter set. Selection is the method used to determine which members of the population will reproduce the next generation. Crossover is the method used to mix the genes between two parents. Mutation is used to modify the genes from the parents to further search the parameter space. Several recent works have used a heuristic to guide the mutation and crossover functions [7] [23].

Evaluating the fitness function perfectly is often a high cost process. Another field of improving GA training is Evaluation Relaxation. In this scheme, the training uses an approximation of the fitness function to reduce computational complexity [16]. This method reflects "Dyna" learning in model-based RL. As with general function approximation, this can be done with any form of regression. The complexity of modeling the fitness function would necessitate using a more complex learning agent, such as a neural network or a support vector machine.

Another tunable parameter within GAs is how long each sample is allowed to perform. In other words, what is the stopping criterion for each sample that does not encounter a failure condition within the environment? This concept is known as Time Continuation [19]. At its core, Time Continuation is an instance of the exploration-exploitation trade-off. Stopping the exploration of a population early introduces uncertainty in selecting the best parameter sets for reproduction at the benefit of reducing training time.

E. Position Relative to Related Work

Our proposed algorithm has several key differences compared to the aforementioned related works. Previous applications of GAs in RL tasks have focused on using GAs to replace backpropagation in deep Q learning. Only a single paper shows work in using GAs to optimize hyperparameters in this area. We extend their work using several methods. First, the previous paper only used purely random sampling to initialize their GA. We explore the effectiveness of two other sampling techniques. Second, we introduce a novel concept of "invasive species" within a GA to promote better exploration of the search space. Finally, literature in this field reports the resultant optimal loss function without any intuition behind the learned hyperparameter space. Due to the high variability of many RL tasks [6], only reporting loss results can be non-informative. As such, we report the results of hyperparameters as a distribution of parameter values that produce sufficiently low loss values.

III. Methods

The Genetic Algorithm that will be developed is based on a variant of the Multi-offspring Improved Real-Coded Genetic Algorithm (MOIRCGA). This decision was made to decrease the number of iterations needed to get the Genetic algorithm to converge. The specific structure of the MOIRCGA algorithm is shown in Fig. 1.

A. Genetic Algorithm

The Genetic Algorithm functions as follows. First, an initial population is generated using any sampling method. This population is then evaluated and sorted according to its loss function from the environment. The loss function is returned from the environment the GA is trying to optimize. For this paper, most tests are run on a basic RL cart-pole problem. Then, the algorithm selects the n individual samples from the population with the smallest loss. These individuals are used

to produce children using crossover and mutation functions. The children created from the current generation of parent samples comprise the next sample generation. This process is repeated until the loss is sufficiently minimized or the number of iterations is met.

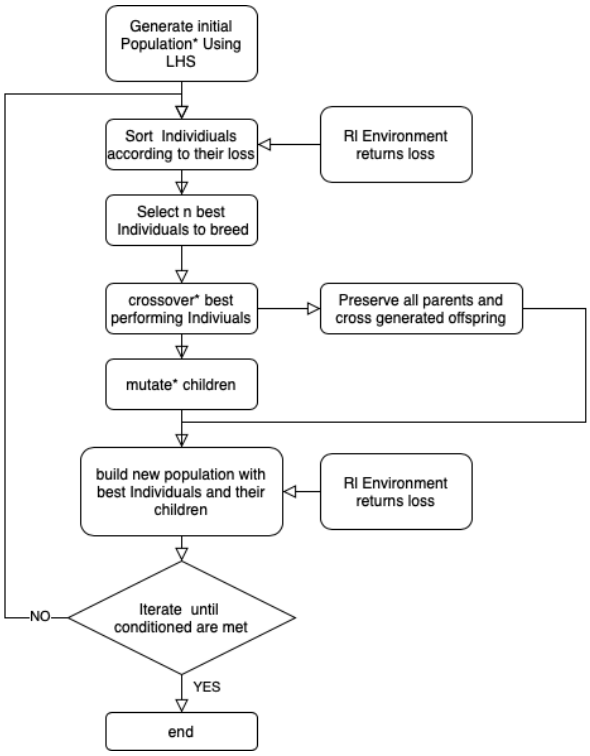

Fig. 1. Flowchart showing the functionality of a generic genetic algorithm framework. The basic steps are sample initialization, generation evaluation, parent selection, cross-over, mutation, and generation creation.

B. Initial Population

The rate of convergence and success of a GA often depending on the quality of the initial population. Many parameter searches converge only once the algorithm has sampled sufficiently close to an optimal region. To balance the characteristics and flaws of uniform and stochastic sampling methods we utilize the Latin hypercube sampling (LHS) method. In this sampling technique, each dimension in the sampling space is divided into n equally sized regions, where n is the number of samples. Next, the sample is selected at random and placed within the hypercube. Next, the dimensional index of the sample is removed from consideration for future samples. In two dimensions, this means each row and column in a unit square region can only be sampled once using this method. The sampling is repeated until no more n-dimensional hyperunits remain in the sampling space. Figure 2 shows a simple 2-dimensional example of LHS sampling with 4 samples.

Additionally, we use uniform random sampling and sparse grid sampling as alternative methods in generating the initial sample population. We compute both the sampling and hyperparameter optimization within a unit hyperparameter space. The parameters are scaled to the environment external to both

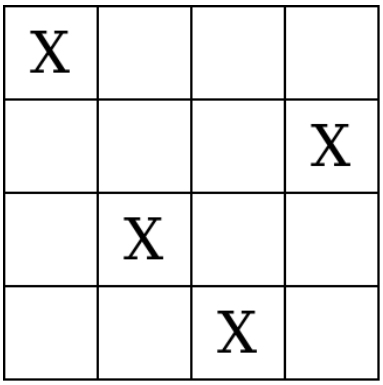

Fig. 2. Latin hypercube sampling with 4 samples. Sampling in this manner is a pseudorandom procedure that encourages the sampling to further explore the state space. Using LHS sampling may improve the base performance of the GA.

the GA and the sampling functions to promote environment agnostic adaptation.

C. Crossover

Crossover combines the genetic information of two or more parents to create children that have similar genes as the parent. In this application, the genes of the population are the hyperparameters for the reinforcement learning agent. To accomplish crossover, the highest performing parents from the previous generation are paired to create two children. One child is created from the left half of one parent and the right half of the other. The other child is created from the right half of one parent and the left half of the other. This is repeated for each pair of parents. If an odd number of parents exists, the lowest loss parent will not share its genes. Crossover is important as it allows good genes to survive but induces change to explore populations close to the best parameters.

D. Mutation

Since crossover tends to save older genes, mutation is used to add new genes into the population. This allows the algorithm to explore new combinations that are not present in the parents and further explore the search space. Mutation occurs after a child sample is created using the crossover function. For generic genetic algorithms, mutation will randomly select a gene and change its value to another random value. This random selection typically uses a uniform distribution. In testing, this mutation was found to produce poor results. If a given gene has very high extreme values, the mutation would most likely pick a value in-between these two extremes. We modify the mutation to have a fifty percent probability of choosing a value either higher or lower than the best parent. Then, the mutation algorithm performs a uniform sampling within the designated sub-region. This helps push exploration for parameters that are close to the edge of the search space. We found this produced better results for parameters with small values, such as learning rate.

E. ML Approximation

A major issue with genetic algorithms is that genes that perform poorly do not have an effect on the algorithm and are discarded. However, this information can be important when determining the overall behavior of a system or predicting values that might produce good results later. These values can create good predictions of parameters with low loss values.

Additionally, Genetic Algorithms can struggle with parameter tuning in reinforcement learning environments. This is due to the variance of results in the RL environment where the same parameters can return very different results. If the non-deterministic environment received bad results once, the GA could discard good values that might produce good values every other time. Again, having a function approximator can help the GA find low loss values.

We solve this problem by using inspiration from ecosystem diversity. Within a given ecosystem, the local species will adapt and evolve into some optimal state. When an invasive species from other ecosystems are artificially introduced, the local species must adapt and improve or be wiped out. These species can be created based on a function approximation of all the results. This concept of invasive species pushes both the native population and the invasive population into a more optimal state.

For this system, we applied the invasive species concept by using all values in the training history to update a small neural network to approximate the parameter space. Using this approximation, we predict the best parameter values. This was then introduced back into the GA by populating half of the parents with the predicted best parameters. The neural network gives optimized values within the approximated space that are then introduced into the true hyperparameter space as an ”invasive species”. This is shown in Fig. 3.

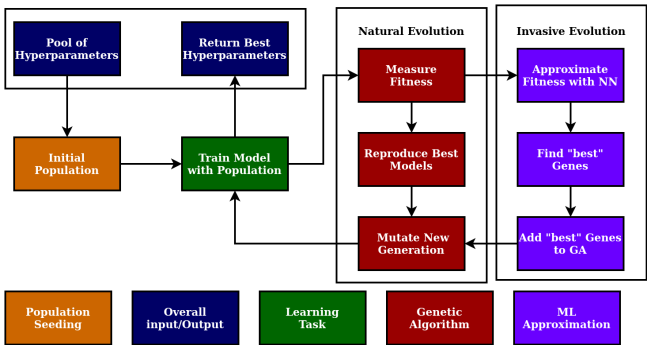

Fig. 3. Genetic Algorithm with ML Approximation.

IV. Results

A. Deterministic Number Guessing

We first conducted a simple experiment to perform a preliminary validation of our implementation. The task of the genetic algorithm was to guess a pre-defined array of 10 numbers. The loss was defined as the L2 norm of the difference between the target array and the array output by the genetic algorithm. The genetic algorithm converged to the target array with around 150 generations. Plotting the resulted loss of the genetic algorithm shows the loss is monotonically decreasing. Using this task, we repeated the experiment with different sampling methods for the initial population generation. One was the Latin hypercube sampling (Fig. 4) and the other was uniform random sampling (Fig. 5. However, the different sampling parameters did not produce noticeable results in this simplistic learning task.

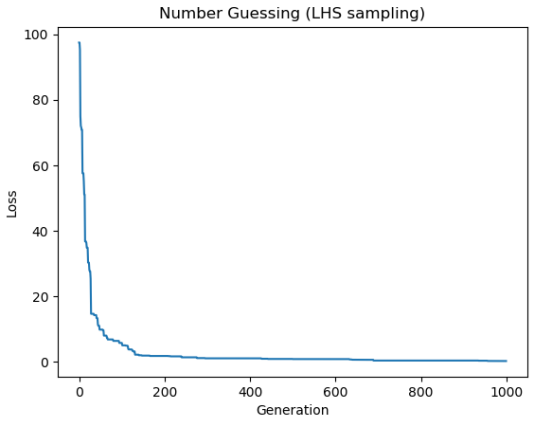

Fig. 4. Number Guessing with Latin HyperCube Sampling

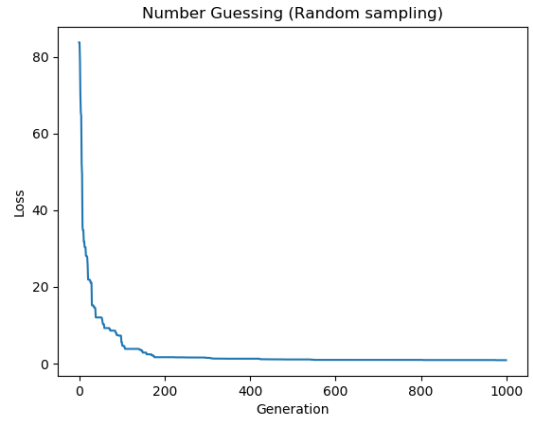

Fig. 5. Number Guessing with Uniform Random Sampling

A second test further explores the effectiveness of each sampling method in generating the initial GA population. Three methods are evaluated in this test: Latin hypercube, random, and uniform sampling. The random samples for this test are all drawn from a uniform distribution of the entire search space. Uniform sampling tries to maximize the distance between each sample. In an ideal case, such as a square value in a 2D search space, this would form a perfect square grid. In practice with non-square sample counts, this cannot be achieved using a simple grid. We find uniform samples by creating a dense set of uniform sample points within the

search space, then finding a k-means clustering algorithm for the dense samples. Though computationally expensive, this method generally performs well-spaced samples. Figure 6 shows example results for 5 and 12 samples for the 3 sampling methods.

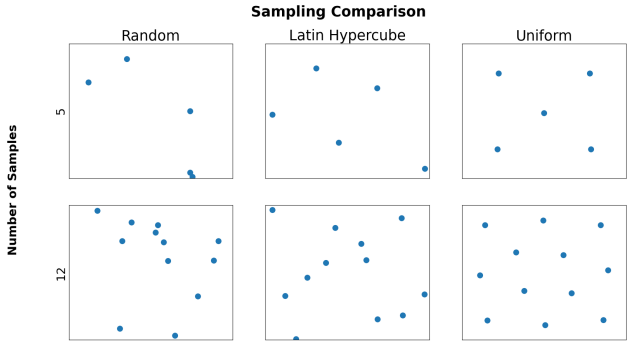

Fig. 6. Example results for each of the 3 sampling methods for 5 and 12 samples.

The effectiveness of each sampling method is found by running the algorithm for 100 training iterations per method, and finding how many samples does it take for the GA to converge below a loss of 10. Then, each method is plotted as a distribution of iterations required. Figure 7 displays the results of this procedure. All three methods are centered around 550-600 samples with an improved performance by the Latin hypercube method. Repeating this procedure shows some variance in which method is best, but LHC is often the highest performing method. Also note the legend shows a fourth method for comparison: fully random sampling.

We also attempted this test by repeatedly selecting each parameter from a uniform distribution without employing the GA optimization. The lowest loss produced by this method after 100,000 iterations was 21.2. In a low dimensional problem, randomly sampling points can achieve sufficiently optimal results within a reasonable amount of iterations. However, this test shows a better optimization method is required for as little as 10 parameters.

The final validation for the algorithm in the test environment is to report parameter evaluation in an informative manner. To do so, we save the loss values from all training samples through 100 generations of testing. With 10 samples per generation, this yields 1,000 data points per parameter. Then, we remove all samples with a loss higher than 10. Finally, the values are reported as a distribution of successful parameter values. Figures 8 and 9 show examples of bad training and good training respectively.

In Fig. 8 the optimal value was 30. The distribution shows the best values are nearly equally distributed between 20-30. These results show no clear value and the highest peak (around 20) is a non-optimal value. A successful aspect of this training result is it provides a focused band of values for later rounds of training.

Figure 9 on the other hand shows a successful training run. The optimal parameter was 60, and a tall, narrow density peak

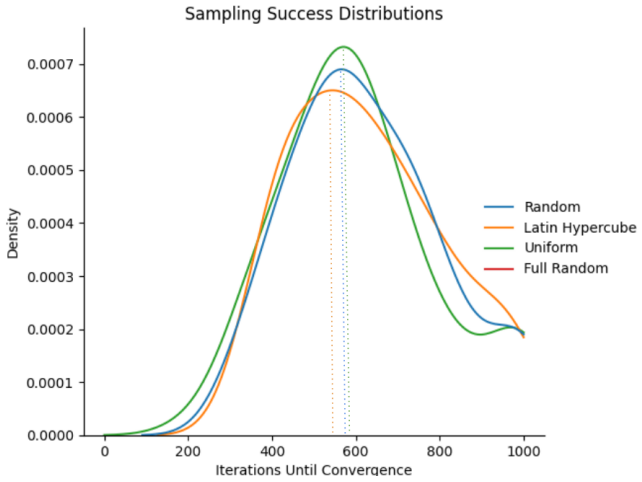

Fig. 7. Poor training result for an optimal parameter value of 30.

is centered on that value. This plot also shows the training also showed success on a nearby value before converging. Given this successful run, the user can confidently select 60 as the optimal value for further training.

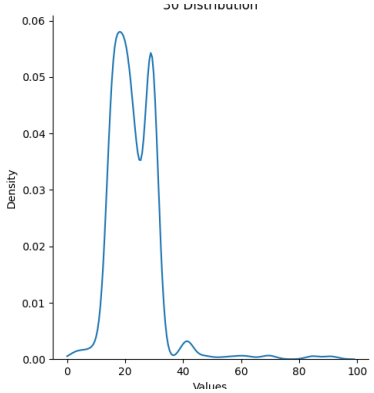

Fig. 8. Poor training result for an optimal parameter value of 30.

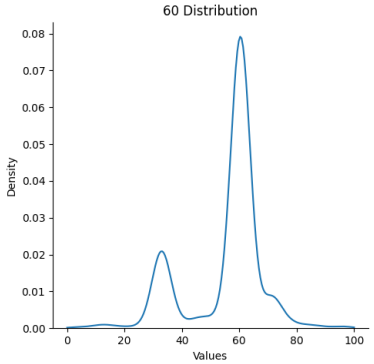

Fig. 9. Successful training result for an optimal parameter value of 60.

B. Neural Network Optimization

We also experimented with our GA framework with a neural network. The neural network we used was a simple multi-layer perceptron model with two hidden layers. The first hidden layer had 200 hidden units and the second hidden layer had 100 hidden units. The task for the neural network is hand-written digit classification on the UCI Optical Recognition of Handwritten Digits Data Set. The two parameters we used GA to optimize are the learning rate and the momentum for the SGD optimizer. Each set of generated hyper-parameters are evaluated by training the NN with them for 100 epochs and calculating the testing accuracy with the trained model. The loss is obtained by subtracting the testing accuracy with 1. We were able to observe that our GA algorithm consistently improved the testing accuracy of the trained models. In Fig. 10, we can see that the loss dropped monotonically within the first 5 generations and then converged to a quasi-steady state after that.

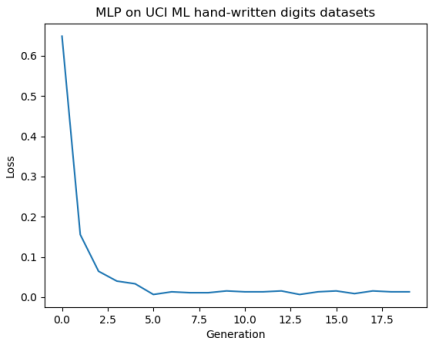

Fig. 10. MLP on UCI hand-written digits dataset.

In this experiment, we also plotted the density graphs for each of the hyper-parameters. In Fig. 11 we can see that the generated learning rate parameter peaked around 0.002 which is close to the recommended value. In Fig. 12, we can see that the generated momentum peaked around 0.8 which is close to the recommended value of 0.9. These results are impressive because the initial populations for the parameters were generated within the range of 0-1 without the use of domain knowledge.

We extended this testing to optimize discrete hyperparameter value sets. Hyperparameters that could be discrete include the number of nodes per layer, the number of layers, the type of activation function, etc. Instead of reporting the success of these values as a distribution, the results show a bar chart of the values that produce successful results. Repeating the previous optimization problem with the number of nodes, activation function, and the number of layers produces Figs. 13 and 14. The loss for this setup is similar to Fig. 10 and shows the system can optimize at least 5 hyperparameters in a non-trivial environment.

Analyzing the results shows a network with three levels and 25 nodes per layer is most likely to produce adequate results.

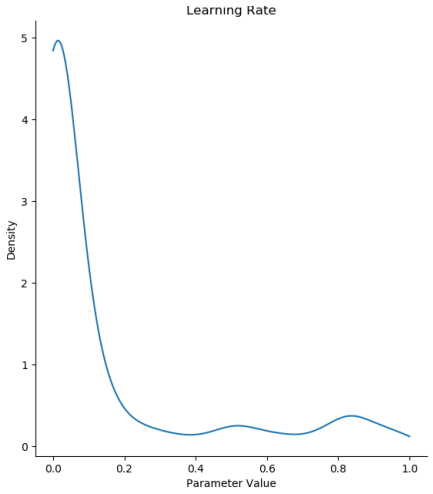

Fig. 11. Training result for MLP learning rate

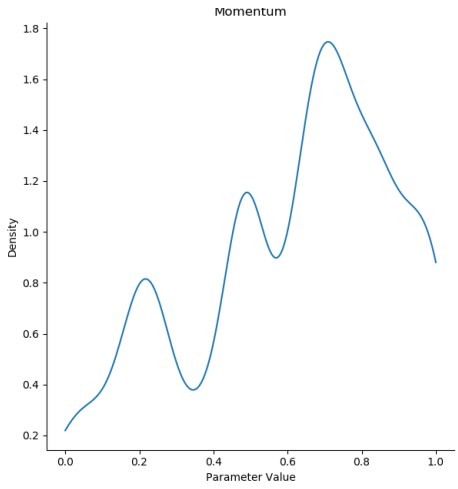

Fig. 12. Training result for MLP momentum

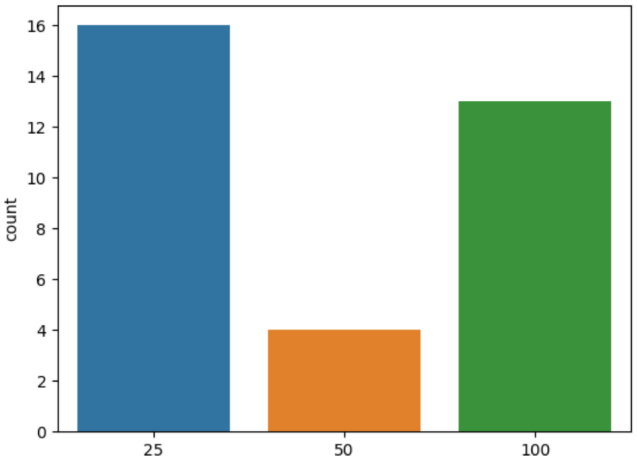

Fig. 13. Distribution of successful values for the number of nodes per layer.

The networks with 100 nodes per layer also performed well. This test also performed an optimization for the activation

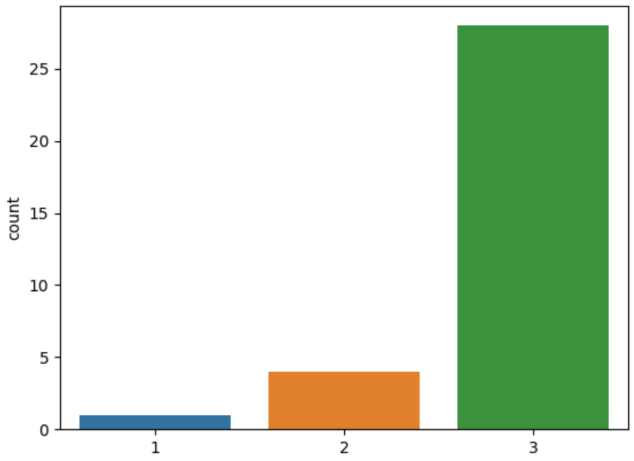

Fig. 14. Distribution of successful values for the number of layers.

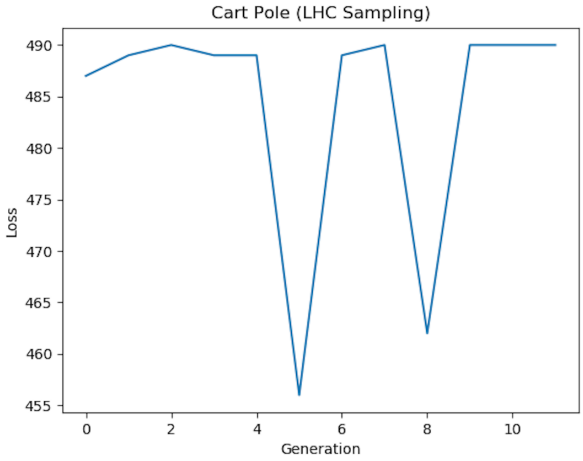

Fig. 15. Cart-pole without ML approximation, failed run

function between the tanh function and the sigmoid function. However, only the tanh function converged to an adequate loss value. Consequently, the plot is not displayed. Though these plots do not show relationships between the hyperparameter values, they do inform the user of the more successful values to achieve convergence.

C. Reinforcement Learning Optimization

The other experiment that we conducted was learning hyper-parameters for the proximal policy optimization algorithm in the cart-pole environment. We picked the learning rate and the maximum value for the gradient clipping as the 2 parameters to learn for this experiment. We initialized the initial population with Latin hypercube sampling and set the range for both parameters to be 0-1. The GA algorithm setup used 10 samples per generation. For each generation, the loss of each sample was calculated by learning a policy with generated hyper-parameters and replaying the learned policy in a cart-pole RL environment. We ran the algorithm for 20 generations both with and without neural network optimization.

The performance of the GA does not always return good results without the ML approximation. It was found that the GA could converge on optimal solutions, within 12 runs, only 60 percent of the time. Figure 15 shows an example of a failed run. When solutions were found, it took on average 9 generations to get a loss of 0 once. When the neural network is introduced, the GA found a loss of 0 every single time during testing. It also found an increased number of losses lower than when compared to only using a GA. Figures 16 and 17 shows the different between using the ml approximation for Reinforcement Learning Optimization. Figure 16 converges quicker and finds more parameters that return a loss of zero. This same behavior can be seen in Fig. 17 but converges slower with fewer parameters returning a loss value of zero.

Using the ML approximation, the algorithm returned good results for the RL environment. The learned hyper-parameter from this particular run was 0.0035 for the learning rate and 0.55 for the maximum value for the gradient clipping. The

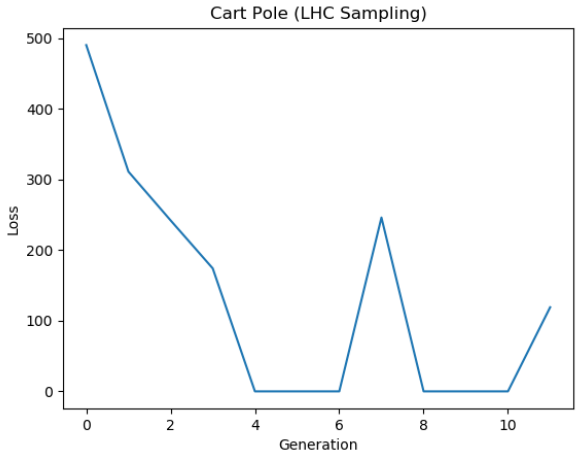

Fig. 16. Cart-pole with ML approximation

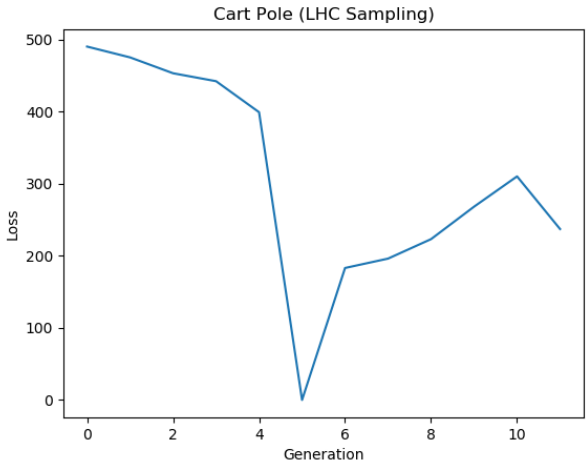

Fig. 17. Cart-pole without ML approximation

recommended parameter set from the RL library is 0.00025 for the learning rate and up to 0.5 for the gradient clipping.

The distribution of valid parameters over 50 iterations is shown in Figs. 18 and 19.

Despite the GA having no domain knowledge to guide its parameter search, the calculated parameters converged close to the recommended values. The similarity between the recommended values and the calculated values further validate the algorithm's performance.

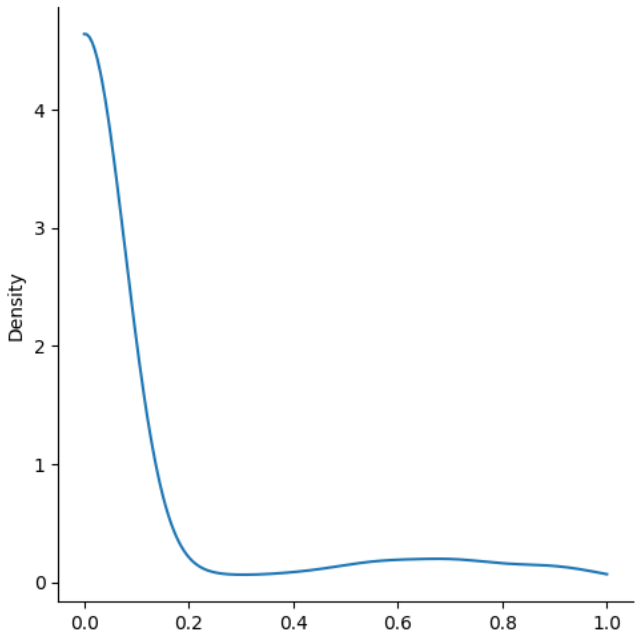

Fig. 18. Cart-pole distribution for the parameter learning rate 50 generations

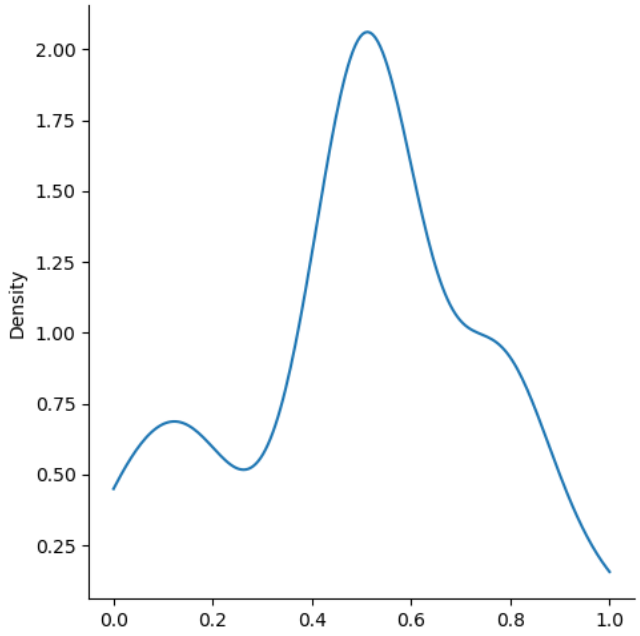

Fig. 19. Cart-pole distribution for the parameter Max Grad Norm 50 generations

This system also had the advantage of providing initial results very quickly. Figures 20 and 21 shows similar results to Figs. 18 and 19 but with only 10 generation compared to 50.

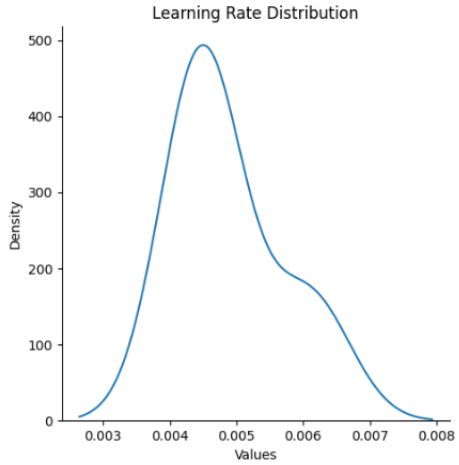

Fig. 20. Cart-pole distribution for the parameter learning rate 10 generations

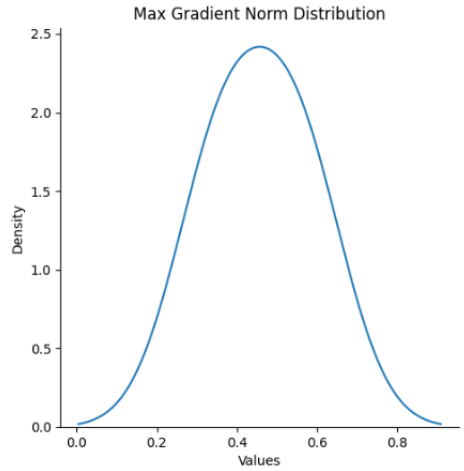

Fig. 21. Cart-pole distribution for the parameter Max Grad Norm 10 generations

V. Ethics Analysis

Many ethical concerns relating to artificial intelligence and machine learning focus on data collection and distribution, considerations in training complex models, and the application and deployment of trained models. This work presents a generic training tool that is both data source and application agnostic. Accordingly, the ethical considerations for this work center on the ethical training of complex machine learning models. Within this realm, our work applies to two specific concerns: reducing the carbon footprint of training ML models and the black-box nature of complex machine learning.

The purpose of this work is to reduce the time and energy used to optimize hyperparameters in model training. As

such, the intended outcome aligns with the ethical concern of excess energy consumed by ML development. Applying hyperparameter optimization on a wide scale holds potential benefits in alleviating, but not eliminating, the concerns of the AI carbon footprint. As such, the work in this paper reflect ethical behavior as it relates to efficient model training.

Interpreting machine learning training and actions is the second major area of concern for this work. Increasing scrutiny has arisen in knowing what an ML model represents, and understanding why/how an ML model maps inputs to actions. This field resembles a major ethical concern with this work. By employing a hyperparameter optimization tool, inexperienced users may create a dependence on ease of use tools. Prolonged dependence could naturally increase ignorance by stunting the development of domain knowledge. A second related problem is optimizing hyperparameters for high variance environments. In these settings, a highly non-optimal hyperparameter set could mistakenly be reported as the optimal solution. Users with limited domain knowledge have poor intuition for the quality of hyperparameters, and would not know if the training potentially failed.

We combat this issue by reporting a distribution of successful hyperparameter values, not the parameter set that produced the lowest cost. Using this method, the user increases their domain knowledge each time they perform a hyperparameter search. Then, the user has better intuition as to the quality of a reported parameter set by comparing the parameter values to the distribution of successful parameter values. If the returned parameters fall within a region of the distribution with a high density of successful runs, the user has higher trust in the quality of the solution. If the values diverge significantly from the distribution, the user can visualize the discrepancy and adjust or retrain the values accordingly.

VI. Conclusion

In this project, we have created an environment agnostic genetic algorithm framework for obtaining the optimal hyperparameters. The effectiveness of our framework has been validated and proven in three different environments, a deterministic number guessing, RL cart pole, and NN handwritten digit classification. We have shown promising results in all three of the environments as shown in the results section. We have shown that our GA framework was able to converge to near-optimal parameters despite initializing the samples without the use of domain knowledge. Additionally, we have come up with the novel approach of approximating task environment with a neural network to obtain better results faster. Overall, our GA framework showed superior performance than random sampling regardless of the tasks.

VII. Future Work

This work proposes and validates a general hyperparameter optimization framework without performing a formal analysis of the system. Now that the framework is functionally validated, future work should focus on formalizing the variance and lower-bound performance of the system. Performing an academically rigorous analysis of the framework will validate the system beyond the multiple successful experiments shown throughout this paper.

A second area of future work should further explore the invasive species concept. The hyperparameter space approximator uses a two-layer neural network with 100 nodes per layer. Experimenting with a variety of machine learning algorithms introduces a rich area of exploration. Algorithms that may perform well in this sphere include decision forests (DFs), support vector machines (SVMs), Gaussian processes (GPs), different neural network architectures, and different neural network paradigms (convolutional neural networks, recurrent neural networks, etc). These algorithms could better approximate the hyperparameter loss space in different application areas. The factors that could be explored include the amount of available loss data, the variance of the training space, and the computational complexity overhead of the parallel training. Further exploring the performance of the ML approximator in this space offers further room for future work.

References

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13 (null):281–305, February 2012. ISSN 1532-4435.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Vijay Chahar, Sourabh Katoch, and Sumit Chauhan. A review on genetic algorithm: Past, present, and future. *Multimedia Tools and Applications*, 10 2020. doi: 10.1007/s11042-020-10139-6.

[4] Qiong Chen, Mengxing Huang, Qiannan Xu, Hong-Ming Wang, and J. Wang. Reinforcement learning-based genetic algorithm in optimizing multidimensional data discretization scheme. *Mathematical Problems in Engineering*, 2020:1–13, 2020.

[5] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning, 2015.

[6] Kaleigh Clary, Emma Tosch, John Foley, and David Jensen. Let's play again: Variability of deep reinforcement learning agents in atari environments, 2019.

[7] Boxin Guan, Changsheng Zhang, and Jiaxu Ning. Edga: A population evolution direction-guided genetic algorithm for protein-ligand docking. *Journal of computational biology : a journal of computational molecular cell biology*, 23(7):585—596, July 2016. ISSN 1066-5277. doi: 10.1089/cmb.2015.0190. URL https://europepmc.org/articles/PMC4931765.

[8] G. Hinton. A practical guide to training restricted boltzmann machines. *Neural Networks: Tricks of the Trade*, pages 559–619, 2012.

[9] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

[10] Khalid Jebari. Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3:333–344, 12 2013.

[11] Jinn-Tsong Tsai, Jyh-Horng Chou, and Tung-Kuan Liu. Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *IEEE Transactions on Neural Networks*, 17(1):69–80, 2006. doi: 10.1109/TNN.2005.860885.

[12] Joon-Yong Lee, Min-Soeng Kim, Cheol-Taek Kim, and Ju-Jang Lee. Study on encoding schemes in compact genetic algorithm for the continuous numerical problems. In *SICE Annual Conference 2007*, pages 2694–2699, 2007. doi: 10.1109/SICE.2007.4421447.

[13] M. D. McKay, R. J. Beckman, and W. J. Conover. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979. doi: 10.1080/00401706.1979.10489755. URL https://doi.org/10.1080/00401706.1979.10489755.

[14] S. Mikami and Y. Kakazu. Genetic reinforcement learning for cooperative traffic signal control. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 223–228 vol.1, 1994. doi: 10.1109/ICEC.1994.350012.

[15] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276, Sep 1999. ISSN 1076-9757. doi: 10.1613/jair.613. URL http://dx.doi.org/10.1613/jair.613.

[16] Kumara Sastry. Evaluation-relaxation schemes for genetic and evolutionary algorithms. Master's thesis, University of Illinois at Urbana-Champaign, 6 2002.

[17] Adarsh Sehgal, Hung Manh La, Sushil J. Louis, and Hai Nguyen. Deep reinforcement learning using genetic algorithm for parameter optimization, 2019.

[18] G. K. Soon, T. T. Guan, C. K. On, R. Alfred, and P. Anthony. A comparison on the performance of crossover techniques in video game. In *2013 IEEE International Conference on Control System, Computing and Engineering*, pages 493–498, 2013. doi: 10.1109/ICCSCE.2013.6720015.

[19] Ravi Srivastava. Time continuation in genetic algorithms. Master's thesis, University of Illinois at Urbana-Champaign, 3 2002.

[20] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp, 2019.

[21] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning, 2018.

[22] David J. J. Toal, Neil W. Bressloff, and Andy J. Keane. Kriging hyperparameter tuning strategies. *AIAA Journal*, 46(5):1240–1252, 2008. doi: 10.2514/1.34822. URL https://doi.org/10.2514/1.34822.

[23] Jiquan Wang, Mingxin Zhang, Okan Ersoy, Kexin Sun, and Yusheng Bi. An improved real-coded genetic algorithm using the heuristical normal distribution and direction-based crossover. *Computational Intelligence and Neuroscience*, 2019:1–17, 11 2019. doi: 10.1155/2019/4243853.

[24] A. Wicaksono and Ahmad Afif Supianto. Hyper parameter optimization using genetic algorithm on machine learning methods for online news popularity prediction. *International Journal of Advanced Computer Science and Applications*, 9, 2018.

[25] Xueli Xiao, Ming Yan, Sunitha Basodi, Chunyan Ji, and Yi Pan. Efficient hyperparameter optimization in deep learning using a variable length genetic algorithm, 2020.

[26] Steven R. Young, Derek C. Rose, Thomas P. Karnowski, Seung-Hwan Lim, and Robert M. Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*,

MLHPC '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340069. doi: 10.1145/2834892.2834896. URL https://doi.org/10.1145/2834892.2834896.