Sequentially constructing and optimizing artificial neural networks (ANNs) using Monte Carlo tree search (MCTS)

Kawther Rouabhi
ASEN5264: Decision Making Under Uncertainty
Ann and H.J Smead Department of Aerospace Engineering Sciences, University of Colorado Boulder
Boulder, CO, USA
kawther.rouabhi@colorado.edu

Keywords—neural architecture search, automated machine learning, Monte Carlo tree search

I. Introduction

First introduced in the mid-20th century, artificial neural networks (ANNs) have evolved from theoretical constructs to indispensable tools in artificial intelligence research and applications. ANNs are composed of interconnected layers of neurons that use weighted sums and nonlinear activation functions to capture complex patterns in a data set. Their ability to learn from data and emulate human-like decision-making processes has revolutionized various domains, including computer vision, natural language processing, and autonomous systems. Supervised learning involves two fundamental types of big data problems: regression and classification. When performing regression, ANNs predict continuous outputs, such as predicting house prices, while in classification problems, they categorize inputs into discrete classes, such as identifying whether an email is spam or not. During training, a neural network adjusts its weights and biases iteratively through backpropagation, minimizing the error between predictions and true labels. Optimization algorithms such as gradient descent facilitate this process, gradually improving the network's ability to capture complex patterns in the data.

Navigating the intricacies of ANN development poses significant challenges due to the many architectural decisions and parameter adjustments involved. A brute force approach to exploring all possible configurations is impractical and computationally expensive. Finding the best configuration can be more traditionally framed as an optimization problem wherein the network design characteristics are optimized in addition to the networks hyperparameters. This project explores the use of classic sequential decision-making techniques to build a model that outperforms other ANN structures for a specific supervised learning task.

II. Background and Related Work

Automated machine learning (AutoML) streamlines the process of model development by leveraging algorithms to automate various phases of analysis, from data preprocessing to model selection. These analysis steps are non-trivial even to expert data scientists, and AutoML aims to make the application of machine learning to various domains more accessible through this automation.

First released in 2019, Auto-Keras builds on Python's Keras library, commonly used to construct ANNs, to automate the process of neural architecture search (NAS) [1]. It uses Bayesian optimization techniques to systematically explore and evaluate different network architectures, including various combinations of layers, activations, and hyperparameters, to optimize performance on a given task. Bayesian optimization starts by selecting an initial set of configurations and fitting a probabilistic model to the observed data. It then uses an acquisition function to choose the next configuration to evaluate, balancing exploration and exploitation. After evaluating the selected configuration, the observed data are used to update the model iteratively, refining its estimates of the loss or accuracy. This process continues until a stopping criterion is met, yielding the configuration with the highest estimated performance. AutoPytorch, like Auto-Keras, specializes in automating the selection and optimization of neural network architectures using Bayesian optimization and NAS techniques but is tailored to the PyTorch framework [2].

AutoWeka, an early player in AutoML published in 2013, employs a metalearning approach, utilizing historical performance data from various machine learning tasks to learn which algorithms and hyperparameter configurations are most effective for different types of problems, enabling it to make informed decisions in future model development tasks [3]. Researchers launched Auto-Sklearn in 2015, which, using the popular machine learning library sci-kit

learn, optimizes the choice of 15 classifiers, 14 feature preprocessing methods, and four data preprocessing methods, resulting in a structured hypothesis space of 110 hyperparameters [4]. In contrast to AutoWeka, Auto-Sklearn encompasses a broader scope, automating algorithm selection, hyperparameter tuning, and feature preprocessing using techniques like Bayesian optimization, ensemble construction, and model selection.

In 2016, Tree-based Pipeline Optimization Tool (TPOT) was introduced by researchers as an automated approach to optimize preprocessing, feature selection, and model selection steps of machine learning development, particularly for classification methods [5]. The nodes of the trees represent some characteristic of the training data pipeline, such as normalizing the data or applying the K-nearest neighbors classifier. Such characteristics are combined to make up a complete pipeline, which is then trained on a data set and yields some classification accuracy. The authors implemented genetic programming techniques which are generally used to construct trees of mathematical functions to optimize some criteria, in this case, the classification accuracy. Using this approach, characteristics of the pipeline are modified to maximize accuracy. TPOT was able to outperform a stand-alone Random Forests classifier for several well-known data sets and demonstrated the potential of these automated tools for data science.

This project is essentially the development of a simplified AutoML technique, specifically automating the step of model construction using Monte Carlo tree search (MCTS), as described in the following sections. The discussed AutoML frameworks can be utilized for any supervised learning task, but this work focuses on classification. TPOT exhibits some similarities to this project in that it too is designed for classification and involves traversal of a tree with nodes that represent functions applied to a data set. In this project, however, those nodes are specifically building an ANN, rather than choosing another algorithm or applying a preprocessing step. As is done with Auto-Keras and AutoPytorch, the objective of this project is neural architectural search, though this process is optimized using MCTS and not Bayesian optimization techniques.

III. Problem Formulation

This experiment focuses on a classic supervised classification task: identifying handwritten numbers. The MNIST data set, shown in Fig. 1, is a collection of 60,000 labeled images of handwritten numbers zero through nine, with 10,000 additional labeled images in the testing set [6]. The 784-pixel images are flattened into a vector of grayscale values, which is the input to the neural network. The network has an output layer of ten nodes that correspond to the ten number categories. One-hot encoding was performed to transform categorical variables into a numerical format suitable for an ANN. After the output layer, the Softmax activation function is applied to model the probability distribution across classes. Categorical cross-entropy (CCE) is then utilized as the loss function, measuring the dissimilarity between predicted and actual class distributions.

Fig. 1. Samples of the MNIST data set

This project aimed to frame the construction of an ANN as a Markov Decision Process (MDP) and analyze potential policies implemented to build such networks for a particular data set in a step-by-step approach. Each state of the MDP represented a particular configuration of an ANN, including the number of hidden layers, the number of nodes in each layer, and the activation function applied between layers. The action space initially comprised of six actions: the ability to add a hidden layer of 32 nodes, add a hidden layer of 128 nodes, change the activation function to be rectified linear unit (ReLU), change the activation function to linear, delete the last hidden layer, and change nothing. Taking the action to change nothing results in a terminal state of the MDP. Two additional actions were added to the action space to address the project Level 3 goal to increase complexity and flexibility: adding a hidden layer with 256 nodes and changing the activation function to LeakyReLU with an alpha value of 0.1.

The state transitions of the MDP are deterministic. When an action is taken, the change is applied to the model in the next state if the action would alter the network. For simplicity, the number of hidden layers a network has cannot exceed four. If an action is taken to add a hidden layer in this case, the network remains unchanged. Likewise, taking the action to change the activation function to the one already in use does not alter the state. Logically, the action of deleting the last hidden layer only changes the network if there is at least one hidden layer to remove. The reward function encompasses two optimization objectives for a classification problem: maximizing classification accuracy and minimizing network complexity, which is related to the time that it takes to train and test the model. Specifically, the reward is the difference in runtime (T) from one state to the next (with a scaling coefficient) subtracted from the difference in accuracy (ACC) between these states:

$$R(s,a,s') = ACC_{s'} - ACC_s - 0.0001(T_{s'} - T_s) \quad (1)$$

The formulation of this task as an MDP enables access to a versatile toolbox of algorithms tailored for sequential decision-making problems, allowing for the systematic application of various strategies to efficiently find an optimal solution.

IV. Solution Approach

MCTS is designed to handle MDPs and performs the following steps each time it reaches a new state: building a search tree, expanding it based on simulations of possible actions, and selecting the most promising paths to explore further, ultimately guiding decision-making toward optimal solutions through a process of repeated exploration and refinement. One significant concern of using an online method for an AutoML task such as this one is the computational complexity of generating a new state and associated reward since doing so requires training and testing of the model. When an action is taken in a given state, the generative model function creates the updated network using Julia's Flux library. From there, the model is trained over 60 epochs using the Adaptive Moment (Adam) optimizer and a learning rate of 0.001. The resulting accuracy and the training time of the model are used to compute the reward according to Equation 1. Given my limited time and computing resources, the time allotted for the algorithm to choose an action based on the tree was limited to one hour. Even with this restriction, validating the results of the MCTS approach took many hours of runtime.

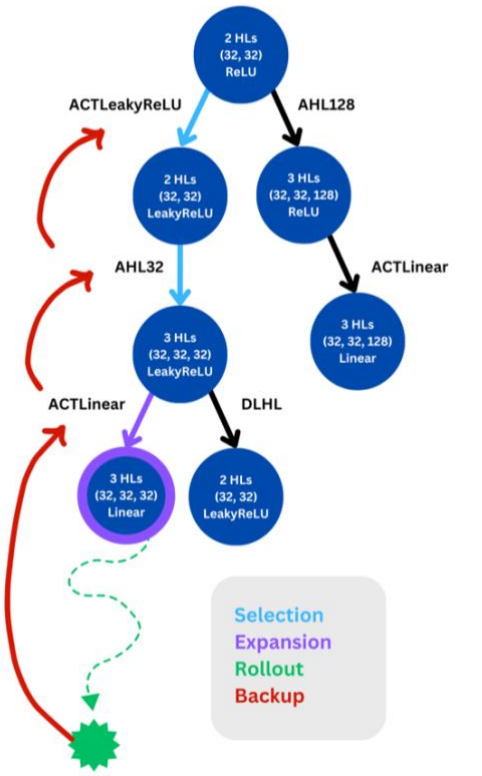

Fig. 2. A diagram outlining an example tree constructed using the described implementation of MCTS

The updates made to the search tree are implemented recursively in Julia. The algorithm searches for a new state to explore, and if it finds one before reaching max depth or a terminal state, it initializes its estimate of the Q-value to zero and performs a rollout from that state using a heuristic policy. This heuristic policy, referred to as Heuristic Policy 2 in the next section, dictates adding a hidden layer with 128 nodes if no hidden layers are present and changing the activation function to ReLU if a linear one is in use, and changing nothing otherwise, with a 70 percent frequency, while selecting a random action 30 percent of the time. A rollout is also done with this policy if max depth or a terminal state is reached. The Q-value estimate is updated for each state-action pair leading up to the rollout. A discount factor of 0.9 was used to diminish the effect of further rewards on the Q-value estimate. An exploration constant of 0.3 was selected to balance the selection of already promising actions and paths not well-explored.

Fig. 2 illustrates an example of one iteration of MCTS as applied to this problem, where the algorithm selects the path that reaches a network with 3 hidden layers of 32 nodes each and a LeakyReLU activation. From there, it expands the tree to an unexplored action, changing the activation function to be linear (ACTLinear). Then, a rollout is performed from that point, and the results are backpropagated up the same path.

V. Results

Using the reward function outlined in Equation 1, a rollout can be performed to simulate the path of actions taken to build a network following some policy and compute the sum of discounted rewards. Limited to a maximum of ten steps per simulation for tractability, five Monte Carlo simulations were conducted for four different policies. Heuristic Policy 1 (HP1) opts to add a hidden layer of 32 nodes if no hidden layers are present or change the activation to be LeakyReLU if not already in use or change nothing otherwise. Monte Carlo simulations were also conducted for Heuristic Policy 2 (HP2), as described in the previous section, a policy that chooses an action completely randomly, and the MCTS policy. Ideally, the number of simulations would be significantly greater to improve confidence in the reported metrics, and limiting the amount to five was done due to limited computing resources for this project. Table I lists the mean discounted reward across five Monte Carlo simulations for each policy, as well as the associated standard error of the mean (SEM).

Assuming these results would be consistent if more simulations were conducted, the positive mean discounted reward of the implemented MCTS approach demonstrates the ability to take actions that improve model accuracy and/or runtime. As one may expect, the random policy yielded the lowest mean discounted reward and the highest SEM, which includes a simulation that resulted in a positive discounted reward. The heuristic policies perform similarly across simulations. Since HP2 chooses a random action 30 percent of the time, it may be having a negative effect on the rewards representing too much exploration, as compared to Heuristic Policy 1, which does not have any built-in exploration strategy.

TABLE I. MONTE CARLO SIMULATIONS

Policy	Mean discounted reward	SEM
Random	-0.0332	0.063
Heuristic Policy 1	-0.0705	0.05
Heuristic Policy 2	-0.0852	0.058
MCTS	0.0325	0.0098

Fig. 3 displays examples of paths taken in accordance with the heuristic policies and the MCTS approach, all beginning with an initial state of zero hidden layers and a linear activation function (Step 0). Since there is no stochasticity in HP1, the resulting network would always look the same given this initial state. In this example for HP2, a random action is taken on Step 2 to add a hidden layer of 32 nodes. The activation function is then changed to ReLU in Step 3, and finally, the action to change nothing is taken for Step 4. The MCTS approach in this example results in a network with two hidden layers of 32 and 256 nodes respectively and uses a ReLU activation.

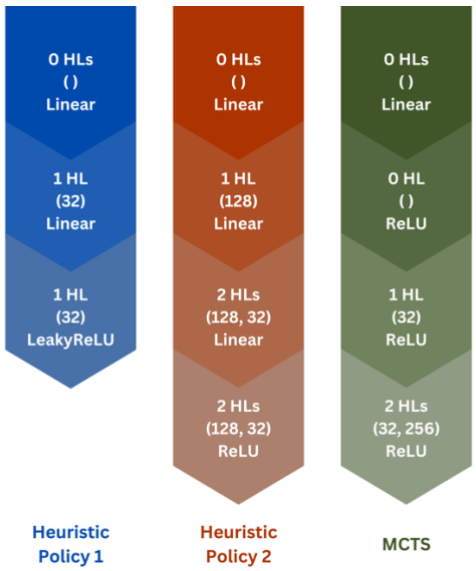

Fig. 3. Example paths taken to build neural networks according to Heuristic Policy 1, Heuristic Policy 2, and the implemented MCTS approach

Table II features the accuracy and timing metrics of the resulting networks illustrated in Fig. 3. The example model yielded from HP2 outperforms that of HP1 but takes over three times more time to train with over 100,000 parameters. In this example, MCTS, which considers both training time and accuracy in its estimation of state-action values, yields a model that trains in about the same time as the HP1 network with a 17.9 percent improvement in accuracy.

TABLE II. EVALUATING THE EXAMPLE MODELS

	HP1	HP2	MCTS
Accuracy (%)	63.2	74.4	81.1
# of Parameters	25,450	104,938	36,138
Training Time (s)	108	383	111

VI. CONCLUSIONS

This project addresses neural architecture search, a growing area of research within AutoML. An MCTS approach was implemented to sequentially select actions to build an ANN that provides the best fit to the data set at hand, MNIST. This method was designed to maximize accuracy while minimizing model complexity. The performance of the MCTS technique was compared to two heuristic policies and a random policy, all of which were outperformed by MCTS in a limited number of Monte Carlo simulations. Given the computational expense of sequentially training neural networks from scratch, time and computing resources play a significant role in the success of these methods. With these improved capabilities, the state and action spaces can be expanded to consider more detailed and diverse configurations. In lieu of some other stopping criteria, actions could be added to change the number of epochs performed in training, as well as actions to change the optimization algorithm used or the learning rate. This technique could, of course, be extended to regression problems. Formulation of the NAS process as an MDP allows for other algorithms to be implemented for finding optimal policies, for example, Deep Q-Networks (DQNs). While this approach was not directly applied to the ANNs used in my graduate research, I learned about several AutoML techniques and look forward to applying them to my research projects in the near future.

CONTRIBUTIONS AND RELEASE

The author *grants* permission for this report to be shared with students taking this course.

The entire codebase was implemented by myself, Kawther Rouabhi, including the object-oriented setup of the model states, a robust generative model, and evaluation of the policies. The implementation of MCTS was based on the algorithm as discussed in ASEN5264 and practiced in Homework 3

REFERENCES

[1] Jin, H., Song, Q., & Hu, X. (2018). Auto-Keras: An Efficient Neural Architecture Search System. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[2] Zimmer, L., Lindauer, M., & Hutter, F. (2020). Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL.

[3] Thornton, C., Hutter, F., Hoos, H., & Leyton-Brown, K. (2013). Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. *Proc of KDD 2013, pp. 847-855.*

[4] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., & Hutter, F. (2015). Efficient and Robust Automated Machine Learning. In

C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems (Vol. 28).*

[5] Olson, R.S., Bartley, N., Urbanowicz, R.J., & Moore, J. (2016). Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. *Proceedings of the Genetic and Evolutionary Computation Conference 2016.*

[6] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner (1998) Gradient-based learning applied to document recognition. *Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324.*