GenAlg Search: Utilizing a Genetic Algorithm For Neural Network Hyperparameter Optimization

Tafadar Soujad

Abstract

We have seen in past research that Random Search has shown to be an effective method at estimating optimal hyperparameters for the training of a machine learning system. While Random Search has shown promise over the classical Grid Search method by adding in a touch of stochasticity to more effectively explore hyperparameter values, we proceed to use a Genetic Algorithm implementation for this task. We show how the solution space can be more effectively explored with another heuristic method that still exploits the strength of randomly choosing solution candidates but with a more directed approach at maximizing the output measures of a neural network, namely maximizing validation accuracy and achieving a shorter wall clock training time. In this project, we compare the validation accuracy of a neural network trained with parameters searched by both Random Search and Genetic Algorithm and find that the Genetic Algorithm finds hyperparameters that produce a competitive or better validation accuracy that results in a lessened training time for the neural network.

1. Introduction

A typical modern machine learning framework makes thorough use of Multi-Layered Perceptrons in the form of a neural network, with each perceptron being an abstraction of an algorithm that helps takes in input and performs a regression operation to isolate a targeted part of the solution space when used in congruity with other perceptrons. How the perceptron is weighted is dependent on the parameter of the perceptron. Subsequently, this decides how the specific algorithm that the perceptron is abstracting works in relation to other perceptrons in the network.

The operation of training a neural network is, in essence, creating an algorithm that manages the fine tuning of a network of interconnected algorithms. Thus, we can call the algorithm that trains the neural network a sort of "hyper-algorithm" - an algorithm whose operation controls the operations of sub-level algorithms. It follows that, just as an algorithm has parameters that affect it functions, a "hyper-algorithm" will have a set of hyper-parameters that affect the how the training process happens and can end up changing the overall functionality of how the parameters if the perceptrons are fine tuned.

This can easily lead to a more or less robust model that can truly capture the essence of the model at hand and can drastically change how the end result of the model training process happens. Some hyperparameters include number of layers in the model, number of perceptrons in each model layer, overall learning rate, moment learning rates, choice of optimizer, and many others. The importance of hyperparameters can vary according to various factors including type of machine learning technique being used, the task that the training process is attempting to capture, the data available for training, and even other hyperparameters.

While the "hyper-algorithm" is tasked with managing the process of finding the optimal parameters that directly affect the make-up of the neural network, we can also have a sort of "super-algorithm" to try and find optimal values for the hyperparameters. There have been multiple super-algorithms in the past that have been utilized and shown subsequent improvement over one another.

One super-algorithm, Grid Search, that focused on sequentially searching through possible hyperparameters in set ranges until optimal values were found was shown to be very effective at this task, and a later the Random Search super-algorithm which randomly chose values in the search space for hyperparameters andtested them out saw equal performing hyperparameters values found in less time(Bergstra, 2012).

This investment into hyperparameter optimization showed that systemic methods of searching through hyperparameters can lead to better functioning machine learning systems over just manually inputting commonly used hyperparameters and hoping for the best. In this paper, we will be exploring how we can implement an evolutionary algorithm to perform an effective hyperparameter and evaluate it's efficacy compared to a previously established, effective method - more specifically the Genetic Algorithm technique.

A Genetic Algorithm is a heuristic optimization technique of the evolutionary algorithm family which simulates the biological processes of natural selection. It does this by stochastically generating and iteratively evolving a population of candidate solutions, called individuals or chromosomes. Each individual represents a potential solution and is encoded as a string of values, called genes. The algorithm evaluates the fitness of each individual based on how well it solves the problem according to some chosen metric by the architect of the algorithm implementation, and individuals with the highest fitness values are selected for reproduction.

Through a process of selection, crossover, and mutation, new individuals are generated, combining traits from the fittest individuals in the population, less fit individuals are eliminated, and new individuals are stochastically generated to replace them. Over successive generations, the population evolves, and the solutions tend to improve. Genetic algorithms are widely used in optimization and search problems, offering a flexible and powerful approach to finding good, but not necessarily strictly optimal, solutions.

Using this method, we can implement a Genetic Algorithm framework as a super-algorithm to try and find optimal hyperparameters for a neural network training schema. In doing so, we must find ways to encode the values we intend to test as values in a chromosome that can be properly manipulated in the scope of the Genetic Algorithm operations.

2. Literature Review

We see in previous works that the problem of hyperparameter optimization has been thoroughly explored for neural networks and other machine learning algoithms, where we see that ascertaining optimal hyperparameters to train a model to learn the discernments relegated from data is an NP-hard problem(Yang, 2020), thus there isn't a simple iterative solution to find optimal hyperparameters. This provides us with a motivation to explore techniques that may be effective at finding solutions to the hyperparameter search problem that may not return the optimal solution, but instead can give us insight to a very effective solution for some application.

One example of this is the Random Search method, which explores how hyperparameters chosen at random can be an effective method for picking efficient hyperparameeter values in order to train a neural network(Bergstra, 2012). We saw how Bergstra and Bengio described that simply running through values for hyperparameters is a heavily inefficient method that can result in sub-optimal results with respect to how much time it takes to find optimal hyperparameters. They proposed and experimented with a heuristic method in the form of Random Search, which aimed at taking random values along the solution space for hyperparameters in a set number of trials instead of searching the solution spacce exhaustively. This turned out to be a very effective improvement over both searching through the solution space and just choosing folkloric values manually.

However this method can be deficient in optimally searching the solution space, as there are other techniques that take a more directed approach. The Genetic Algorithm is one of those techniques that can take advantage of stochasticicty to explore a solution space while also having some direction according to what we are looking for in order to direct the achievement of better results. We have seen in past work that Genetic Algorithms can be a powerful tool for optimizing hyperparameters if implemented and coded properly(Xiao et. al, 2020) Xiao et. al showed that Genetic algorithms can achieve better results while running the Genetic Algorithm for a specific amount of time - 30 hours, so much so that their specific method didn't need to take the full 30 hours before they achieved a vastly better result than other methods. This is different from the experiment we undertake in this paper in so far as we are attempting to see how individual training instances are affected by yperparameters along with accuracy as opposed to just how accuracy fares after training for a specific amount of time like Xiao et. al did.

3. Methodology

For the purposes of our experiment, we employed the use of a very simple and shallow Deep Convolutional Neural Network, due to the constraints of the technology available to for data pipelining and training purposes. For further ease of training our neural network, we employ a simple dataset towards this measure - the MNIST dataset - so that we may keep our operations light and achievable according to the hardware constraints we face. This network consists of two sets of 2D Convolutional with 32 trainable parameters and Pooling Layers, followed by a densely connected perceptron layer with 128 trainable parameters, followed by another densely connected output layer.

We begin the experiment by creating a baseline measurement of the performance of the Random Search algorithm as a way to measure the performance of the Genetic Algorithm schema on the task of hyperparameter optimization. As this is not a foray into the mechanics and performance of the Random Search super-algorithm, we choose to forgo building a Random Search application from scratch, which would be trivial, and instead opt to use the common hyper parameter optimization library "Keras Tuner" which allows us to take advantage of a professionally built collection of hyperparameter search algorithmss, including the Random Search method. This framework allows us to take advantage of a heavily encapsulated method that requires little effort on our part other than defining a custom training loop for

our neural network.

For Random Search we first create a class that encapsulates the model described earlier to be implemented with an optimizer that utilizes the Stochastic Gradient Descent(SGD) technique. In this encapsulation, we define the structure of the model we are testing as described previously followed by the custom training loop, where the the choices are made apparent for our chosen "test" hyperparameters. This includes choosing a batch size randomly from a value between the range of 32 and 512 in multiples of 32 and choosing a learning rate that is sampled logarithmically(values are chosen at random from a logarithmic scale rather than a linear scale) from a range of 0.0001 to 0.01. In these choices, we see some folkloric influences for possible hyperparameter values guiding how we set up the experiment.

We proceed to develop another encapsulated Random Search class, this one utilizing the Adaptive Moment Estimation(Adam) technique in it's optimizer. We proceed to choose batch size and learning rates as described before, but we also introduce the search schema to the decay rates for both the first and second moments that are sampled logarithmically from a lower value of 0.8 to an upper value of 1.

Once these are set up, all that is required of us is to run the super-algorithm. For each training loop we use two epochs and 5 trials - notions that we keep consistent in the Genetic Algorithm implementation. While the library implementation does not return training time, it does log the beginning wall clock time and end wall clock time for each trial allowing us to manually discern the training time for each chosen set of hyperparameters

The Genetic Algorithm implementation was developed by hand by the authors of this paper. Albeit somewht crude, it got the job done. Instead of encapsulating the model building method and the training loop, we simply define them as a function to feed to the Genetic Algorithm to build the model and another function to be utiilized by the Genetic Algorithm model that takes in our choice hyperparameters and spits out a validation accuracy, also the result of two epochs, and a wall-clock training time for the training loop.

The actual Genetic Algorithm had separate implementations, one for the SGD optimizer and one for the Adam optimizer. The reason this needed separate encapsulations as opposed to a single encapsulation that could had flexible optimizer options upon instantiation is because the varying hyperparameter values associated with the methods need to be encoded into the structure of how the Genetic Algorithm's chromosomes respective to the values we are choosing to test.

The methodology used in the Genetic Algorithm implementation followed exactly the process shown in Figure 1.

We first initialize a population of candidate solutions that contain values for batch size randomly chosen in a fashion similar to the Random Search implementation, and choose a learning rate randomly also from the same interval as before. Wheras before we sampled the learning rate directly from the interval, here we instantiated a series of learning rate candidates from creating a list of evenly spaced points between the ends of the interval and shuffling it randomly before taking the first one to be our candidate solutions learning rate. val, here we instantiated a series of learning rate candidates from createing a list of evenly spaced points between the ends of the interval and shuffling it randomly before taking the first value in the shuffled list to be our candidate solutions learning rate.

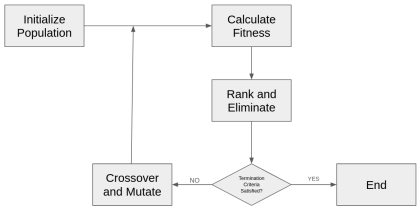

Figure 1. The Genetic Algorithm Process

Afterwards we calculate the fitness of each candidate solution by passing in he values of the candidate solution as their respective hyperparameter values to the training loop function. The resulting fitness value is derived from the validation accuracy we get from the training loop function as mentioned before, which also makes note of he wall clock training time of the function. Afterwards, we rank the candidate solutions from greatest validation accuracy to lowest validation accuracy and eliminate the bottom half of these rankings from consideration.

Upon reaching this step, we reach an option on whether or not to continue based on the supposed response to some predetermined termination criteria. Typically this can take the form of some exceptional fitness value being reached, but for our purposes of carrying out an exploratory work we simply carry out the number of trials and take the best value we have procured thus far. This strategy can work in our favor as the solutions will not degenerate over time since the top performing solutions are either retained or improved upon, thus we are constantly searching and comparing according to a Pareto efficient principle.

Should the termination criteria not be broached by the current iteration of the Genetic Algorithm, we then move on to the crossover and mutation functions. As we have elim-

inated the bottom half of the candidate solutions for low performance, we can now breed the candidate highest perfoming candidate solutions to create new candidate solutions, or individuals, for our population. Usually, these breeding pairs from the list of existing candidate solutions would be partnered randomly to simulate natural population diffusion, but we opted to instead breed the top candidate solution with the second best, the third best with the fourth best, and so on. This involved switching every other hyperparameter between the two to create two new candidate solutions from each pair, so the the switched hyperparameters are staggered betweeen each breeding pair, as shown in Figure 2.

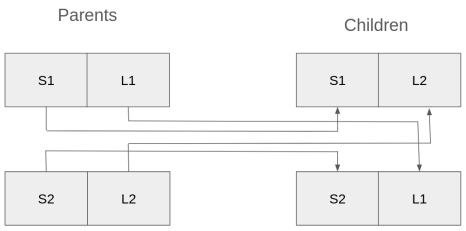

Figure 2. The crossover operation as applied in our implementation for the SGD case, with S being the batch size and L being the learning rate values

Once this is done we append the newly created individuals to the population and we also randomly generate new individuals so as to keep the population size constant over generations. This helps us thoroughly search the solution space as we are evolving and retaining best solutions while also stochastically generating new candidate solutions at the same time.

We also perform a mutation operation to prevent the stagnation and getting stuck at local minima, a problem that is a common bane of the Genetic Algorithm. We simulate mutation in biological processes by sampling a Poisson distribution with a low mean value, and we use this to determine to either pick a new batch size or slightly shift the learning rate by 5%, and we randomly choose to either grow or shrink it.

This process is repeated until the termination criteria of a pre-determined number of trials in the form of some number of generations has been undergone, at which point the top validation accuracy is returned with the respective hyperparameters and training time for inspection.

Each experiment takes a training loop of 2 epochs, with 5 trials for Random Search and 5 generations for the Genetic Algorithm to keep matters consistent between the methods

Relevant code can be found through the following link:

`github.com/trs-code/GenAlgSearch`

4. Results

In Figure 3 we observe a scatter plot of the results of our experiment when applied to a model that utilizes an optimizer ming use of SGD, with the red data points showing results of the Genetic Algorithm and blue data points showing the results of the Random Search, with Validation accuracy running along the x-axis and Wall Clock Training Time running across the y-axis. In Figure 4 we see the results under the same conditions for a implementation that uses an optimizer taking advantage of the Adam method. It should be noted, that the optimal trend for this data is data points that are more towards the right on the graph, or closer to a validation accuracy of 1, and more towards the bottom of the graph, or closer to a training time of 0.

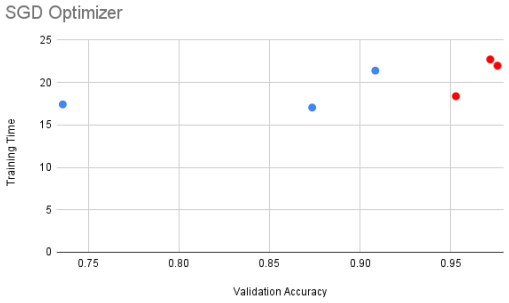

Figure 3. Results of Experiments On Random Search(Blue) and Genetic Algorithm(Red) With SGD Optimizer, Validation Accuracy vs. Training Time

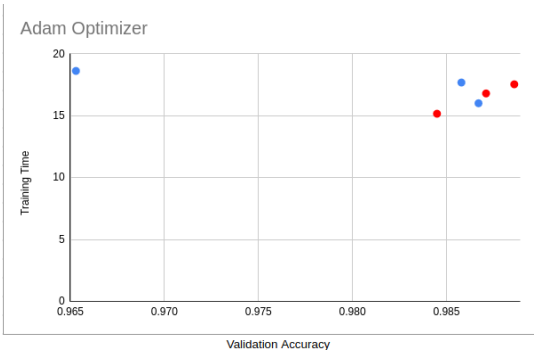

Figure 4. Results of Experiments On Random Search(Blue) and Genetic Algorithm(Red) With Adam Optimizer, Validation Accuracy vs. Training Time

5. Analysis and Conclusion

We see from the data we have collected that there is a strong case that the Genetic Algorithm is a worthy contender for a hyperparameter search. In the SGD case shown in Figure 3, the Genetic Algorithm consistently outperforms the Random Search method in terms of validation accuracy in all 3 experiment runs. However, it seems that Random Search method beats out the Genetic Algorithm in terms of training time.

This is not true in the case of the Adam optimizer method, where we see that the Genetic Algorithm method generally beats out the Random Search method in terms of both accuracy and training time. These observations are based on a majority vote scale(e.g. if method A shows better performance than method B for training time on 2 out of the 3 experiments, we say that method A is superior to method B in terms of training time).

Thus we see that Genetic Algorithm Search is a decently strong improvement on the past work related to hyperparameter search methods in the form of Random Search. We base this off of not only previous results that show that a hyperparameter search method that utilizes a Genetic Algorithm can result in higher accuracy, but also that it may result in faster training times for a model.

For further work, one may experiment with larger models and more dynamic datasets to see if how the Genetic Algorithm model fares against other hyperparameter search methods in industrial settings. One may also want to experiment with hyperparameter searches using other evolutionary algorithms that are commonly seen as similar to Genetic Algorithms but is also recognized as superior, such as a hyperparameter search based on the Differential Evolution algorithm.

References

Bergstra J, Bengio Y, Random Search for Hyper-Parameter Optimization, Journal of Machine Learning Research, Volume 13, 2012, Pages 281-305

Xiao X, Yan M, Basodi S, Ji C, Pan Y, Efficient Hyperparameter Optimization in Deep Learning Using a Variable Length Genetic Algorithm, 2020, https://doi.org/10.48550/arXiv.2006.12703

Yang L, Shami A, On hyperparameter optimization of machine learning algorithms: Theory and practice, Neurocomputing, Volume 415, 2020, Pages 295-316, ISSN 0925-2312, https://doi.org/10.1016/j.neucom.2020.07.061.