Algorithms for Hyperparameter Optimization of Neural Network Training

Progress Report

October 15th, 2024

Members	Roles	Responsibilities
s m tanvir faysal alam Chowdhoury	Project Manager	Managing the overall project progress; Communicating internally and externally; Coding the parallel algorithm; Writing reports.
Sait Suer	Tech Lead	Writing the base code for the experiment; Coding the breadth-first search and the greedy algorithm; Running the experiments; Writing reports.
Yinning Zhang	Project Designer	Draft the structure of the project; Evaluating the results; Coding the decrease-and-conquer algorithm; Writing reports.

Abstract

Hyperparameter searching is a crucial process for every neural network training. Our goal is to attempt new algorithms for this problem that could possibly return better time efficiency. In this report, we describe the experimental settings and explain the proposed algorithms in detail. We also highlight some preliminary results we have obtained so far and analyze them. At the end, we summarize our progress compared to the original timeline and the remaining tasks.

Background

For every training, the optimization of hyperparameters is essential for every neural network training in order to obtain the best performance of the models. Hyperparameters directly control model structure, function, and performance, and influence the values of the model parameters that a neural network model ends up with. Due to the vast number of possible combinations and the influence these hyperparameters have on each other, even with the promise of advanced GPU units, this process is notably time-consuming. The common approach is using grid search to exhaust all the options, which is computationally very expensive. So, we aim to find some alternative algorithms for hyperparameter search that reduces time complexity compared to grid search but returns equivalent results.

Experiment Settings

Dataset description

We use the MNIST database[1] of handwritten digits which is widely used as a benchmark dataset for experiments. It consists of a training set of 60,000 examples, and a test set of 10,000 examples.

Neural Network Model

We use ResNet18 [2] as the convolutional neural network model because it is widely used as a benchmark model. Resnet consists of 18 layers, each layer including convolutional layers, batch normalization, and ReLU activation functions. ResNet architectures utilize residual learning, which involves skip connections or shortcut connections that bypass one or more layers. These skip connections allow the network to learn residual mappings, which addresses the problem of vanishing gradients and enables the training of deeper architectures. This allows us to use a wide range of layers. We modified the code to allow the model to have 1 layer to 4 layers, depending on what is optimal for this dataset.

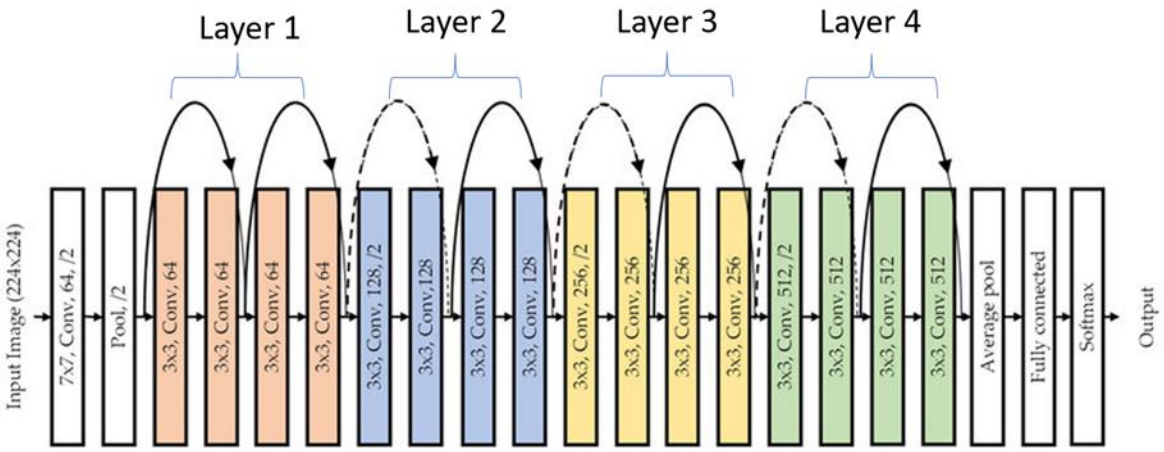


Figure 1. ResNet-18 Architecture

Machine Environment

NVIDIA DGX A100 640GB

GPUs : 8x NVIDIA A100 80GB Tensor Core GPUs

GPU Memory: 640GB total (80GB memory per GPU)

Performance: 5 petaFLOPS AI 10 petaOPS INT8

System Memory: 2TB

Experiment Design

To make the experiments manageable, we fixed all other hyperparameters but only left three hyperparameters for tuning: batch size, learning rate, and number of layers. Each hyperparameter was narrowed to only three candidates, so there are only 27 possibilities. Please see Table 1 for the searching scope of the experiments.

Table 1. Hyperparameter Searching Scope for The Experiments

Batch Size	[64, 128, 256]
Learning Rate	[0.1, 0.01, 0.001]
Number of Layers	[2, 3, 4]

Solutions

Divide-and-Conquer

Imitating binary search, the algorithm [3] will start with the middle element in the array A[(n-1)/2]and compare its result with the next element in the array. If the result of the next element is better, it will focus on the right side of the array. If the result of the next element is worse than the middle element, it will discard the right side but search the left side instead. Again, it fetches the middle element in the new array and compares its result to the result with the next element until it cannot divide any more.

Pseudocode

Algorithm HyperparameterSearch_DivideConquer(A[0.....n-1], B[0.....m-1], C[0.....k-1])

Input: arrays of individual hyperparameters and a neural network model T

Output: The best combination of hyperparameters that returns the highest accuracy

Initialize all the hyperparameters to the middle element of each array, A[(n-1)/2], B[(m-1)/2], C[(k-1)/2]

l<-0; r<n-1

While l<r do

x <- [(l+r)/2]

a1= T(A[x], B[(m-1)/2], C[(k-1)/2]) /// accuracy returned from training neural network with three hyperparameters

a2= T(A[x+1], B[(m-1)/2], C[(k-1)/2])

If a2>a1

l<-x+1

else

r<-x-1

Data Structure

We use the basic Python list to store the hyperparameters. Please see the table below to understand how it iterates the candidates of a hyperparameter.

Table 2. Iterations of the binary search algorithm applied on hyperparameter search

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
Iteration1	l				m1	m2			r

Iteration2						l	m1	m2	r
Iteration3								l/(m1)	r/(m2)

Complexity analysis

Because it will iterate over the entire array anyway, C(worst) = C(best) = C(avg) = $\log_2 \quad N$. So the time for searching will be reduced to $\log_2 \quad N \cdot \log_2 \quad M \cdot \log_2 \quad K$ from N*M*K.

Breadth-First-Search

This algorithm uses a breadth-first search (BFS) [4] to search for the best hyperparameters for a deep learning model by finding the combination that yields the highest accuracy. First, the hyperparameters are provided as input to the graph generation algorithm, which creates a graph. This graph is then passed as input to the breadth-first search algorithm, which returns the hyperparameter combinations in a queue, following a breadth-first order. Each combination in the queue is evaluated sequentially, and the best hyperparameters along with the highest accuracy are returned.

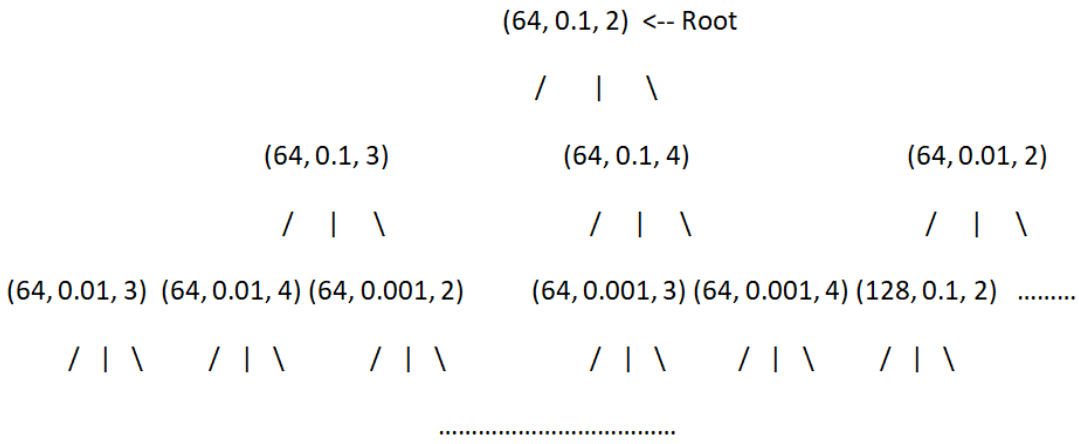
Psudocode

```
best_accuracy = 0
Graph  = create_tree(hyperparameters)
queue = bfs(graph, (root))
While queue not empty do
        (batch_size, lr, num_layers) = queue.popleft()
        model = Model(batch_size, lr, num_layers)
        For epoch in 1 to NUM_EPOCHS do
                Train model accuracy = Evaluate model
                If accuracy > best_accuracy do
                        best_accuracy = accuracy
                        best_hyperparams = (batch_size, lr, num_layers)
Return best_hyperparams, best_accuracy, model
```

Data Structure

For this algorithm, the primary data structure is a graph where the nodes represent different combinations of hyperparameters. The graph is traversed using a queue as part of the breadth-first search (BFS) process. Here's what you can write for the data structure:

- **Graph**: The hyperparameter combinations are represented as nodes in a graph, and their relationships (or progression) can be thought of as edges.



- **Queue (FIFO)**: Used by the BFS algorithm to store and manage the order of visiting nodes (hyperparameter combinations) during the search.
- **Set**: To track the visited nodes and avoid revisiting hyperparameter combinations.
- **TreeNode**: This is used to structure the graph-like relationships during tree creation.

Complexity analysis

The time complexity of **Breadth-First Search (BFS)** is: O(V+E)

Where:

- **V** is the number of vertices (nodes).
- **E** is the number of edges.

This is because BFS explores each vertex and each edge once.

Dijkstra's Algorithm (Greedy Technique)

The proposed algorithm [5] for hyperparameter search, inspired by Dijkstra's Algorithm, leverages a greedy approach to optimize model performance by exploring hyperparameters efficiently. In traditional Dijkstra's Algorithm, a graph is traversed by selecting the shortest path iteratively, using a priority queue to store vertices with the smallest known distances. Similarly, the hyperparameter search algorithm prioritizes hyperparameter combinations based on their performance (e.g., accuracy) and refines the search by selecting the best-performing combination iteratively. Using a min-heap to manage and evaluate hyperparameter sets, this approach minimizes computational complexity while systematically converging toward the optimal configuration. Both algorithms demonstrate a clear application of the greedy technique, where local optimal choices lead to an efficient global solution.

Pseudocode

ALGORITHM Dijkstra_Hyperparameter_Search(P)

// Greedy search for the best hyperparameter combination

// Input: Search space P of hyperparameter combinations (batch size, learning rate, num_layers)

// Output: Best hyperparameter combination and its accuracy

1. Initialize priority queue PQ as an empty min-heap

2. For each hyperparameter combination h in P:

   - Set priority[h] ← 0 (initialize priority for each combination)

   - Insert(PQ, h, priority[h]) (insert the combination into the priority queue)

3. While PQ is not empty:

   - h* ← Extract-Min(PQ) (extract the combination h* with the best priority)

   - Train the model with hyperparameter set h*

   - For each epoch of training:

      - Evaluate the accuracy of the model with h*

      - If accuracy(h*) > best_accuracy:

         - Set best_accuracy ← accuracy(h*) (update best accuracy)

         - Set best_hyperparams ← h* (update the best hyperparameter combination)

4. Return best_hyperparams and best_accuracy

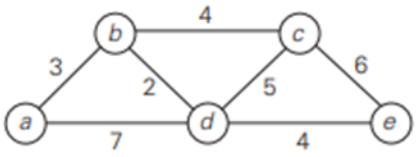
Data Structure

The Dijkstra-inspired Hyperparameter Search Algorithm uses a priority queue to iteratively evaluate and update hyperparameter combinations based on their performance. The best-performing set is tracked and updated to efficiently find the optimal configuration.

Table 3. Dijkstra Hyperparameter Search Table

Iteration	Priority Queue (PQ)	Current Best Hyperparams (Best_h)	Accuracy Updates (best_accuracy)
Start	{h1(0), h2(0), h3(0), ... hN(0)}	None	Initialize all combinations with priority 0
Iteration 1	{h2(0), h3(0), ... hN(0)}	h1	Train with h1, evaluate accuracy

Iteration 2	{h3(0), h4(0), ... hN(0)}	h1 (if better)	Train with h2, update best if accuracy is better
Iteration 3	{h4(0), h5(0), ... hN(0)}	h2 (if better than h1)	Train with h3, update best if accuracy improves
Iteration N	{}	Best_hyperparams	Return final best hyperparameter combination



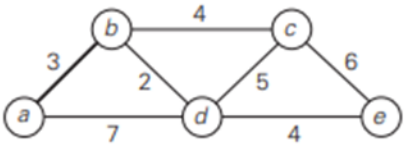
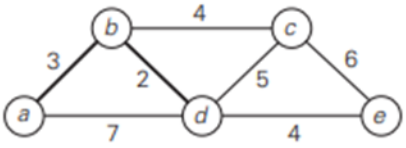
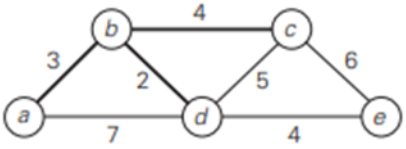
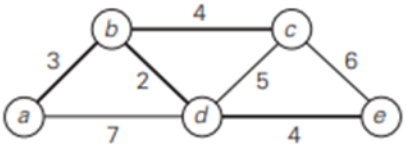
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

Table 4. Dijkstra-inspired Hyperparameter Search Data structure

Iteration	Tree Vertices	Remaining Vertices	Distance Updates
Start	{}	a(0), b(∞), c(∞), d(∞), e(∞)	Initial setup
Iteration 1	{a}	b(3), c(∞), d(7), e(∞)	Distances from a updated
Iteration 2	{a, b}	c(7), d(5), e(∞)	Distances from b updated
Iteration 3	{a, b, d}	c(7), e(9)	Distances from d updated

Iteration 4	{a, b, d, c}	e(9)	Distance from c updated
Iteration 5	{a, b, d, c, e}	None	All vertices processed

from a to b : $a - b$ of length 3
from a to d : $a - b - d$ of length 5
from a to c : $a - b - c$ of length 7
from a to e : $a - b - d - e$ of length 9

Complexity analysis

The greedy technique in Dijkstra's algorithm selects the smallest distance vertex, with a time complexity of O((V+E) log V), where V is the number of vertices and E is the number of edges. Similarly, the hyperparameter search uses a greedy approach, prioritizing the best-performing set, with a time complexity of O(C log C×T), where C is the number of hyperparameter combinations and T is the number of training epochs.

Progress Report

So far, we have completed the following tasks. Compared to the original timeline we proposed, we are in a good progress.

- Review the relevant literature and confirm the gap in the literature;
- Design the algorithms and write the code for the algorithms;
- Write the code for the base experiment;
- Run the experiments with the proposed algorithms and analyze the preliminary results;

Performance Evaluation

We have run each algorithm three times in the same environment and using the same base code except for the hyperparameter searching function. Below is the summary of the performance of each algorithm.

Table 5. Experiment results

Algorithms	Running Time	Accuracy
Greedy Search -> Dijkstra's Algorithm	1515 seconds	99.37%
	1506 seconds	92.63%
Avg. running time: 1429 seconds Avg. acurracy: 94.82%	1266 seconds	92.48%

Divide-and-Conquer	597 seconds	99.37%
	583 seconds	99.46%
Avg. running time: 580 seconds Avg. acurracy: 99.41%	561 seconds	99.40%
Breadth-First Search	3480 seconds	99.50%
	2791 seconds	99.49%
Avg. running time: 3196 seconds Avg. acurracy: 99.51%	3319 seconds	99.54%
Parallel Algorithm	To be completed	

From the results, we can see that the best accuracies from divide-and-conquer and breadth-first search algorithm are consistent, which means these two algorithms were able to find the best combination of hyperparameters. However, the accuracies from the Dijkstra's Algorithm fluctuate. We will investigate this problem and improve the algorithm. We also observe that the time efficiency of divide-and-conquer has significant improvement compared to the algorithm of breadth-first search which exhausts all the options. It matches our theoretical estimation. The ratio of the running time of grid search to that of divide-and-conquer shall be $3*3*3 : \log_2 3 * \log_2 3 * \log_2 3 = 5.67$ and the result from the experiments is 3196: 580 = 5.5. We will further verify the code and this optimistic result.

Future Work and Challenges

We will further work on these remaining tasks:

- Review the code and results to verify the accuracy of the results.
- Because Dijkstra's Algorithm fails to return the desired result, we will investigate the code and improve the algorithm.
- The Breadth-First Search currently is designed to travel all the nodes. We are hoping to find an approach to trim the tree so as to improve its time efficiency.
- We will apply our algorithms to another dataset and a neural network model to verify the effectiveness of our algorithms.

References

1. L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," in *IEEE Signal Processing Magazine,* vol. 29, no. 6, pp. 141-142, Nov. 2012, doi: 10.1109/MSP.2012.2211477.
2. He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.

3. Smith, D. R. (1985). The design of divide and conquer algorithms. *Science of Computer Programming, 5*, 37-51, 54-58. https://doi.org/10.1016/0167-6423(85)90003-6
4. Kozen, D. C. (Year). Depth-First and Breadth-First Search. In *The Design and Analysis of Algorithms* (pp. 19–24). Texts and Monographs in Computer Science (MCS).
5. Wayahdi, M. R., Ginting, S. H. N., & Syahputra, D. (2021). Greedy, A-Star, and Dijkstra's Algorithms in Finding Shortest Path. *International Journal of Advances in Data and Information Systems, 2*(1), 45-52. https://doi.org/10.25008/ijadis.v2i1.1206