Neural Network Hyperparameter Optimization

Qiqi Ouyang (Chinese University of Hong Kong)

Daniel McBride (University of Tennessee, Knoxville)

in collaboration with Kwai Wong (Joint Institute for Computational Sciences)

Stan Tomov (Innovative Computing Laboratory)

Junqi Yin (Oak Ridge National Laboratory)

August 2, 2019

Acknowledgements

This research report is the culmination of work completed during the the Research Experiences in Computational Science, Engineering, and Mathematics (RECSEM) REU site program at the University of Tennessee, Knoxville, which provided use of facilities and many other resources for the duration of the research. The program was funded by a grant from the National Science Foundation. Computing resources were provided by Extreme Science and Engineering Discovery Environment (XSEDE). Workstations were donated by BP High Performance Computing Team. Oak Ridge National Laboratory provided access to ongoing research as well as giving the work a place in the lab's AI Expo which allowed the authors to gain insight from many experts in the field.

Table of Contents

Introduction

The use of artificial intelligence, machine learning, and deep neural networks continues to expand in industries as far afield as speech recognition, medical imaging diagnostics, and traffic engineering [1]. Such industries are critical not only to the convenience of the users of Siri and Alexa, but also to the health and safety of hospital patients and the drivers and passengers on our roads. This short list of AI-dependent industries is but a small sample of both private commercial and governmental services in which the significance of deep neural networks is becoming continually more entrenched, and as such, a proper understanding of how to correctly implement and maintain sophisticated machine learning systems is increasingly important. In designing neural networks, the use of which may ultimately dominate the artificial intelligence industry, a paramount concern is the choice of hyperparameters [2].

The efficacy of a given neural network of sufficient complexity, whose interconnected nodes are essentially a chain of simple functions, the composition of which approximates either a classification task, or some other predictive objective, is determined by the weights assigned to each of these nodes [1]. The optimal weights, the network's parameters, are discovered through what is termed neural network training. The design decisions made before this training regiment, including, among others, those concerning network architecture, node weight optimization methods, and input data processing, are what are called the neural network's hyperparameters. Care must be taken when choosing hyperparameters, because a poor choice of hyperparameters may cause a network's weights to converge slowly during training, or even worse, they may not converge at all, resulting in much wasted computing resources [2]. As these design decisions have been of particular import to data scientists since they began to endeavor in the construction of neural networks, there is a body of theory devoted to solutions to the problem of how hyperparameters may best be optimized.

Classical approaches to hyperparameter optimization include Grid Search and Random Search [2]. However, both of these approaches are subject to inefficiencies which more modern approaches, like evolutionary hyperparameter optimization algorithms such as Population Based Training, attempt to overcome [3]. Although there are distinct differences between the classic and the modern approach, the intricacies of which will be touched on below, there is broad consensus that utilizing the power of parallel computing is a most effective jumping-off point for any hyperparameter optimization algorithm.

Motivation and Research Objectives

Leveraging frameworks like MagmaDNN and OpenDIEL, optimized for hybrid CPU/GPU architectures and distributed multi-node tasks, as well as exploring novel early stopping hyperparameter tuning algorithms amenable to large distributed systems, would likely improve the scalability and efficiency of machine learning aided research, such as Oak Ridge National Laboratory's ongoing materials science research into electron microscopy image classification using the Summit supercomputer [4]. Such complex machine learning classification tasks require comparably complex neural networks. In the case of image datasets with a large number of data points, convolutional neural networks are most often used [5]. Naturally, as the complexity of a network grows so does the complexity of the space of possible hyperparameter configurations.

Common characteristics of hyperparameter search spaces include non-convexity as well as non-differentiability, and thus the hyperparameter optimization community has developed algorithms based on global optimization theory [6]. The evolutionary algorithm, Population Based Training, developed for UC Berkeley's hyperparameter optimization framework Ray Tune is one such algorithm. While the materials science division at ORNL is currently employing the Ray Tune system, it suffers from a scalability bottleneck due to its shared memory model built with the Spark framework used as a backend. Hence, one of the primary objectives of this research project was to overcome this bottleneck by utilizing neural network framework MagmaDNN's native model parallelization in an implementation of Ray Tune's Population Based Training algorithm.

As hyperparameter optimization is still a nascent field, a duel objective of this research project was the implementation of novel hyperparameter tuning algorithms that use insights from recent advances in the field. Many algorithms published in the last few years utilize early stopping, and thus this technique was further explored [3]. Likewise, distributed systems were kept in consideration during the development of these experimental hyperparameter tuning algorithms.

Recording experimental data is central to any scientific research, and so an overarching objective of the project was to document and organize the results of experiments. These experiments range from proof-of-concept trials to benchmarking analysis. The clear and concise presentation of these findings, the methodology involved, and any obstacles encountered is the goal of the present report.

Learning Curve Matching Algorithm

3.1 Early Stopping

Early stopping algorithms are a class of hyperparameter tuning algorithms. The main purpose is to trigger the early stopping action based on some explicit criteria during the training process. Figure 3.1[1] . Successive Halving Algorithm (SHA) and Hyperband are typical early stopping algorithms which demonstrate the desired implementablity with parallel programming and relatively better performances compared to other popular algorithms, such as Bayesian optimization [7]. Besides, there is an improved version of SHA, Asynchronous Successive Halving Algorithm (ASHA), which shows great potential on solving problems with large hyperparameter spaces and it outperforms other state-of-the-art hyperparameter tuning methods [8].

This chapter is aimed at introducing a new early stopping algorithm, Learning Curve Matching Algorithm (LCM). The inspiration and implementation of LCM will be discussed. Comparison experiments between LCM and random search are also described below. The empirical results and experiment limitations will be analyzed and illuminated. Finally, possible improvements and other future work will be discussed.

3.2 Inspirations and Implementations

The Learning Curve Matching algorithm is inspired by Successive Halving Algorithm (SHA) and Asynchronous Successive Halving Algorithm (ASHA). A rough description of SHA is that for a set of hyperparameter configurations, the training process is initialized using every hyperparameter configuration synchronously, and then half of the training processes are stopped early based on their performances when going through several

[1]In the flow chart, *Hyperparameter pool* contains all hyparameters needed tuning shows the whole process of hyperparameter tuning and their properties, such as data types and value ranges; *A trial* means a hyperparameter configuration, i.e. a set that contains a single sample for every hyperparameter.

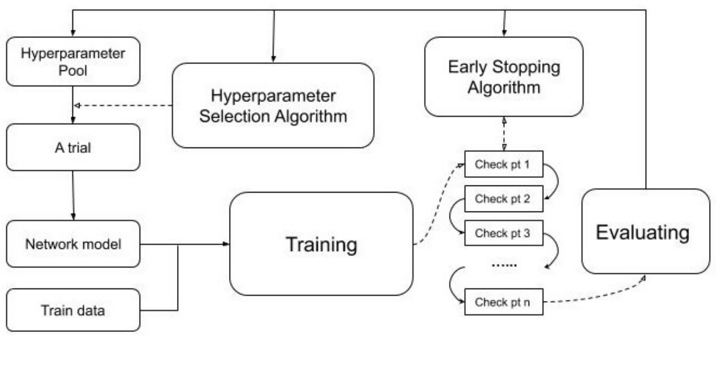
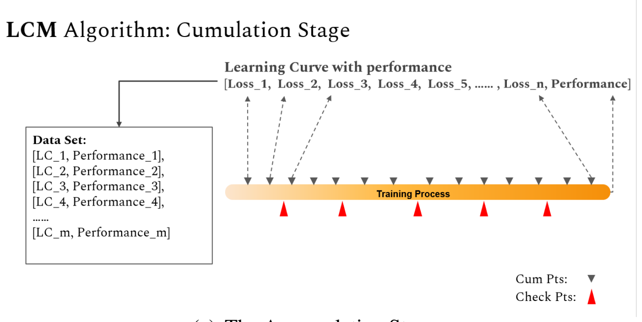
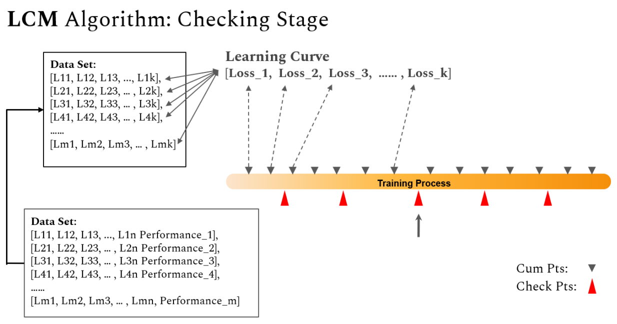


Figure 3.1: A Flow Chart of Hyperparameter Tuning



(a) The Accumulation Stage


LCM Algorithm: Checking Stage
Data Set:
[L11, L12, L13, ..., L1k],
[L21, L22, L23, ... , L2k],
[L31, L32, L33, ... , L3k],
[L41, L42, L43, ... , L4k],
......
[Lm1, Lm2, Lm3, ... , Lmk]
Learning Curve
[Loss_1, Loss_2, Loss_3, ... , Loss_k]
Training Process
Data Set:
[L11, L12, L13, ..., L1n Performance_1],
[L21, L22, L23, ... , L2n Performance_2],
[L31, L32, L33, ... , L3n Performance_3],
[L41, L42, L43, ... , L4n Performance_4],
......
[Lm1, Lm2, Lm3, ... , Lmn, Performance_m]
Cum Pts:
Check Pts:

(b) The Checking Stage

Figure 3.2: Two Key Stages in LCM

preset checkpoints during the training process. LCM has a similar paradigm, using synchronous training and checkpoints. However, the stopping criterion is determined by assessing learning curves instead of performances in LCM.

There are two key stages in LCM, the accumulation stage and checking stage, as shown in Figure 3.2. In the accumulation stage, only accumulative points are activated. The value of the loss function at accumulative points and the corresponding final performances during every complete training, whose combination refers to the so called *learning curve*, will be collected for the next stage. In the checking stage, both the accumulative points and checkpoints will be activated. Besides collecting needed data as in the accumulation stage, learning curve comparison and early stopping will also be implemented when the training process goes through the checkpoints.To implement LCM, a set of accumulative points and a set of checkpoints are required. Furthermore, the split rate is also needed since the division of the whole population n of parameter configurations into these two stages relies on the split rate r.

As to learning curve comparison, preprocessing is required. Previous learning curves would be cut to keep the same length as the partial learning curve γ from the model under training. Moreover, there are additional necessary inputs, the metric d for vector distance calculation and the early-stopping rate s. Based on this metric d The distances between the learning curve γ and every fitted previous learning curves are calculated and collected in a list, the distance list. The previous trial which corresponds to the smallest one in the distance list is the most similar trial i, and its final performance would be viewed as the predicted performance g. According to the comparison between the rank percentage of performance g and the early stopping rate s, early stopping action would be triggered or not.Figure 3.3 is a detailed flow chart of learning curve comparison.

Algorithm 1 and Algorithm 2 summarize the above introduction to LCM and provide a complete picture of LCM. As for the practical implementation, Keras with Tensorflow backend applies suitable functions and classes. A new subclass of callbacks is defined and would be called during every single training process.

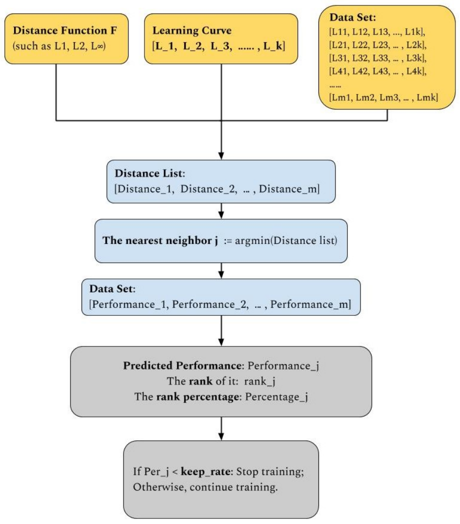

Distance Function F
(such as L1, L2, L∞)
Learning Curve
[L_1, L_2, L_3, ... , L_k]
Data Set:
[L11, L12, L13, ..., L1k],
[L21, L22, L23, ... , L2k],
[L31, L32, L33, ... , L3k],
[L41, L42, L43, ... , L4k],
......
[Lm1, Lm2, Lm3, ... , Lmk]
Distance List:
[Distance_1, Distance_2, ... , Distance_m]
The nearest neighbor j := argmin(Distance list)
Data Set:
[Performance_1, Performance_2, ... , Performance_m]
Predicted Performance: Performance_j
The rank of it: rank_j
The rank percentage: Percentage_j
If Per_j < keep_rate: Stop training;
Otherwise, continue training.

Figure 3.3: A Flow Chart of learning curve comparison

Algorithm 1 Learning Curve Matching Algorithm

```
1:  **Input:** number of configurations $n$, early-stopping rate $s$, split rate $r$, set of checkpoints $C$, set of accumulating points $A$ and distance metric $d$
2:  **Initialization:** $T$ = hyperparameter_configuration_generator($n$), performance list $Z$ = empty list [ ], learning curve list $X$ = empty list [ ]
3:  **for** configuration $\theta \in T$ **do**
4:      learning curve $\gamma$ = empty list [ ]
5:      check trigger $=$ [length$(Z) > n * r$]
6:      **while** training **do**
7:          training progress $p$ = get_training_progress($\theta$)
8:          **if** $p \in A$ **then**
9:              append[$\gamma$, get_training_performance($\theta$)]
10:         **if** $p \in C$ **and** check trigger **then**
11:             stop_training_trigger = check($Y, \gamma, d, s$)
12:         append[$X, \gamma$]
13:         append[$Z$, get_final_performance($\theta$)]
14: **Output:** the best performance *max*$(Z)$
```

Algorithm 2 The Function: `check`

```
1: function CHECK($Y$, $\gamma$, $d$, $s$)
2:     $Y = \texttt{fit}(X, \text{length}(\gamma))$    ▷ keep the first length($\gamma$) elements of every previous learning curve for comparison
3:     $D = \texttt{get\_distances}\ (Y, \gamma, d)$    ▷ compute the distances between $\gamma$ and every fitted learning curve in $Y$ based on the metric $d$
4:     the most similar trial $i = argmax(D)$    ▷ find the most similar trial $i$
5:     predicted performance $g = Z[i]$
6:     rank percentage $q = \texttt{get\_rank\_percentage}(Z, g)$
7:     Return: $(q > s)$
```

Hyperparameter List		
Hyperparameter Name	Data Type	Range
Learning Rate	Float Number	[0, 1]
Momentum	Float Number	[0, 1]
Decay	Float Number	[0, 0.5]
Batch Size	Integer	$\{32, 64, 96, 144, 192, 288, 376, 512\}$
Epochs	Integer	$\{3, 4, 5, 6\}$

Table 3.1: MNIST: Hyperparameter List

3.3 Experiments and Analysis

We conduct two groups of comparison experiments based on two different datasets, MNIST and CIFAR10, separately. [2] In both of these experiments random search is used as the benchmark.

3.3.1 MNIST

In the MNIST group, the training model is a simple network, which contains only one dense layer and applies stochastic gradient descent (SGD) as the optimizer. [3] Learning rate, momentum, decay, batch sizes and epochs are selected to be hyperparameters for tuning. The detailed information of these hyperparameters is listed in Table 3.1. Given a fixed number of hyperparameter configurations (*trials*), we repeat the same experiment 9 times. The average best performances and average computing time of LCM and random search are compared in Table 3.2. Given a fixed computing time, we also repeat the same experiment 5 times. The corresponding experiment results are shown in Figure 3.4. [4]

According to these empirical results, when we fixed the number of hyperparameter configurations, it costs

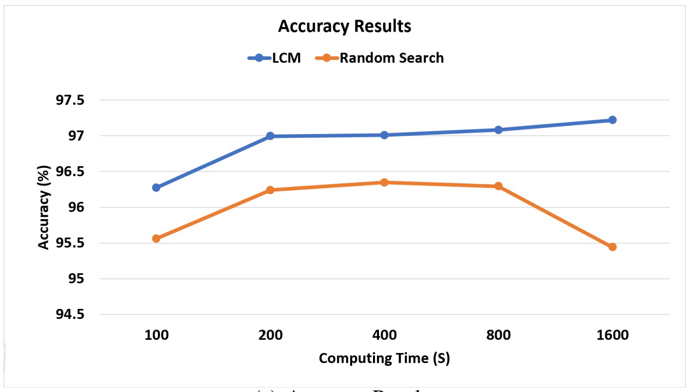
[2]Next, we call these two groups MNIST group and CIFAR10 group respectively.

[3]There are the setting of inputs in LCM: number of configurations $n = 500$, early-stopping rate $s = 0.3$, split rate $r = 0.01$, set of checkpoints $C = \{0.01, 0.02, 0.03, ..., 0.99\}$, set of accumulating points $A = \{0.2, 0.4, 0.6, 0.8\}$ and distance metric $d = L_2$ (Euclidean Distance).

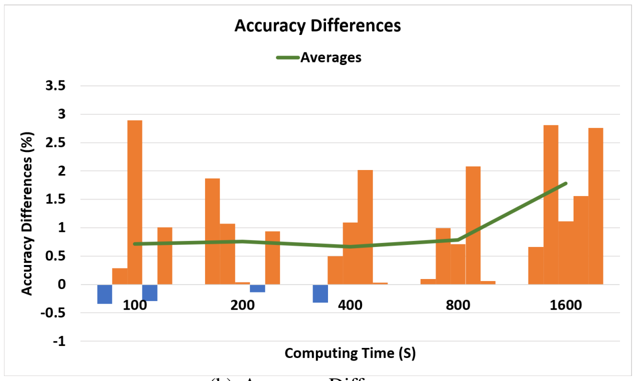
[4]In Figure 3.4, subfigure (a) shows the average best performances. In subfigure (b), the green line indicates the average differences between their best performances and the columns show two algorithms' differences in every single experiment, where orange means that LCM outperforms random search and blue means the opposite. Moreover, subfigure (c) shows two algorithms' standard deviation (SD) of the best performances in every group of fixed-computing-time experiments.

Algorithm Name	Trials	Computing Time (s)	Best Performance (%)
LCM	100	778.50	97.10
Random Search	100	3657.75	97.41
Remark: In 5 of 9 experiments, both algorithms got the same optimal hyperparameter configuration.			

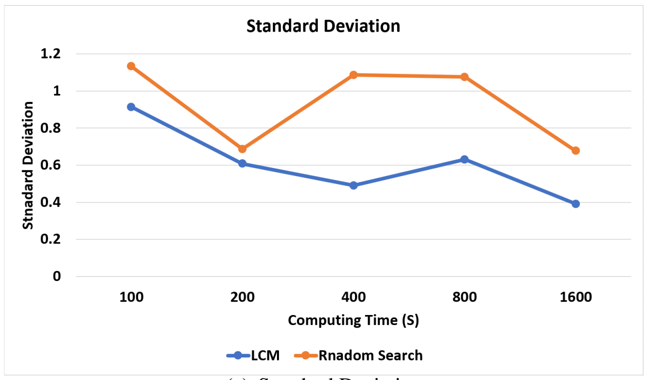
Table 3.2: MNIST: Experiment Result 1



(a) Accuracy Results



(b) Accuracy Differences



(c) Standard Deviation

Figure 3.4: MNIST: Experiment Results 2

less than one third of computing time to implement LCM. Since random search completely goes through every configuration and always gets the optimal configuration, it is reasonable that random search outperforms slightly in the best performance. Nevertheless, the accuracy gap (0.31%) is relatively small and acceptable, especially when time cost is considered. Another interesting point is that, in more than half of the experiments, two algorithms get the same optimal configuration. On the other hand, if the computing time is fixed, random search slightly underperforms in the best performance and the gap is around 0.7%. Furthermore, LCM gains a better result in most of the experiments and completely outperforms with increased computing time, which shows LCM's strong performance stability. In the comparison of the two algorithms' standard deviations, LCM has a smaller standard deviation in every group of fixed-computing-time experiments. Therefore, LCM is more stable than random search in the best performance. In summary, LCM shows a huge advantage in time cost and a slightly worse but comparable performance when the configuration population is fixed. When the computing time is fixed, LCM performs slightly better and with more stability.

3.3.2 CIFAR10

In the CIFAR10 group, we use a much more complicated model, which has four convolutional neural network (CNN) layers and several dense layers. The applied optimizer is Adam.[5] Considering the increased complexity of model, much more hyperparameters are selected for tuning, as shown in Table 3.3. Similar with the MNIST group, we conduct experiments in two different cases, fixing the number of hyperparameter configurations and fixing computing time. The corresponding results are shown in Table 3.4 and Figure 3.5, respectively.[6]

In the experiments with a fixed number of configurations, the empirical results are similar to the results in MNIST group. Compared with random search, LCM achieves a comparable performance with a considerable decrease in time cost. Furthermore, In the more than half of experiments, LCM gets the same optimal hyperparameter configuration as random search. In spite of the increased complexity and sensitivity of the chosen hyperparameters, LCM maintains the stable performance in the comparison with random search. In the group of fixed computing time experiments, the results are more attractive. LCM still outperforms while the gap between the two algorithms is much bigger: the largest gap is over 3%. As for the accuracy differences, LCM gains better results most of the time. However, the two algorithms' standard deviations are close in most of the groups. In the aggregate, LCM shows the same advantages as it does in the MNIST group within a fixed configuration population. With fixed computing time, LCM shows better performances but comparable stability.

3.3.3 Integrated Analysis

These two groups of experiments show a detailed comparison between LCM and random search. Given a fixed population of hyperparameter configurations, LCM achieves a comparable performance with considerable decreasing in computing time. Moreover, the result that in more than half of experiments, two algorithms gain the same optimal configuration gives us a rough understanding why LCM could reach a comparable performance. On the other hand, when the computing time is set, LCM outperforms in the most of experiments and always gets better average best performances. Although The standard deviations of two algorithms in two groups have different

[5]There are the setting of inputs in LCM: number of configurations $n = 800$, early-stopping rate $s = 0.3$, split rate $r = 0.01$, set of checkpoints $C = \{0.01, 0.02, 0.03, ..., 0.99\}$, set of accumulating points $A = \{0.2, 0.4, 0.6, 0.8\}$ and distance metric $d = L_2$ (Euclidean Distance).

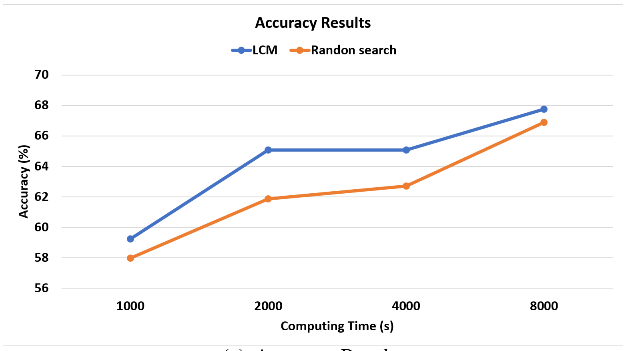
[6]Figure 3.5 has the same structure as Figure 3.4.

Hyperparameter List		
Hyperparameter Name	Data Type	Range
Learning Rate	Float Number	[0.001, 0.01]
Beta_1	Float Number	[0.85, 0.95]
Beta_2	Float Number	[0.985, 0.995]
epsilon	Float Number	{1e-07, 1e-06, 1e-08, 5e-07, 5e-06}
Batch Size	Integer	{32, 64, 96, 144, 192, 288, 376, 512}
Epochs	Integer	{10, 15, 20, 25, 30, 35, 40}
Kernel Size of 1st CNN	Integer	{2, 3, 4, 5}
Strides of 1st CNN	Integer	{1, 2}
Dropout After 1st CNN	Float Number	{0.1, 0.2, 0.3, 0.4, 0.5}
Kernel Size of 2nd CNN	Integer	{2, 3, 4, 5}
Strides of 2nd CNN	Integer	{1, 2}
Dropout After 2nd CNN	Float Number	{0.1, 0.2, 0.3, 0.4, 0.5}
Kernel Size of 3rd CNN	Integer	{2, 3, 4}
Strides of 3rd CNN	Integer	{1, 2}
Kernel Size of 4th CNN	Integer	{2, 3, 4}
Strides of 4th CNN	Integer	{1, 2}
Number of Dense Layers After CNN	Integer	{1, 2, 3}
Dropout After Dense	Float Number	{ 0.1, 0.2, 0.3, 0.4, 0.5}

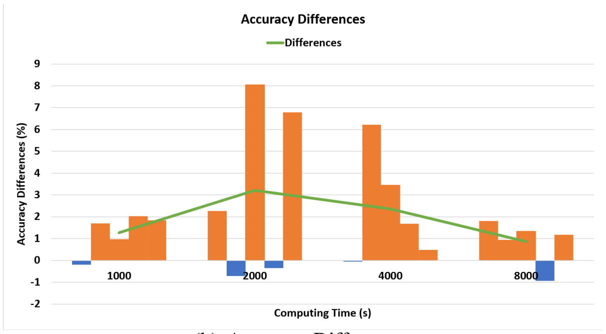
Table 3.3: CIFAR10: Hyperparameter List

Algorithm Name	Trials	Computing Time (s)	Best Performance (%)
LCM	100	8069.08	67.05
Random Search	100	26498.00	67.25
Remark: In 7 of 12 experiments, both algorithms got the same optimal hyperparameter configuration.			

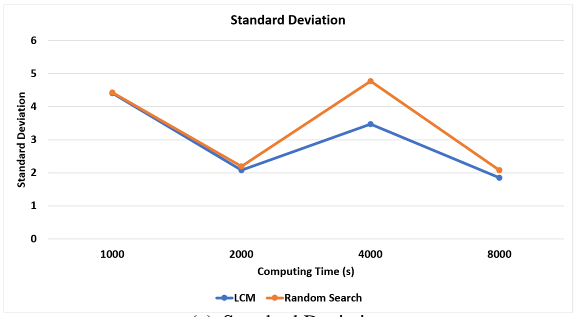
Table 3.4: CIFAR10: Experiment Result 1



(a) Accuracy Results



(b) Accuracy Differences



(c) Standard Deviation

Figure 3.5: MNIST: Experiment Results 2

results, LCM never gets bigger standard deviation then random search. We can deduce that LCM shows a better performance and at least comparable stability compared with random search. Besides that, the gap of accuracy differences between two algorithms are enlarged when we use a much more complicated dataset and choose more hyperparameters. It is reasonable to consider the great potential of LCM in more complex and sensitive cases. More complicated experiments would likely provide even further evidence to bolster these conclusions.

3.4 Further Discussion

3.4.1 Parallel Programming and ALCM

Inspired by SHA and ASHA, Asynchronous Learning Curve Matching algorithm (ALCM) is aimed at the asynchronous implementation of LCM and better utilization of parallel programming. Roughly, LCM can be viewed as an instance of ALCM, with only one worker.

Algorithm 3 Asynchronous Learning Curve Matching

```
1: Input: number of configurations $n$, early-stopping rate $s$, split rate $r$, set of checkpoints $C$, set of accumulating points $A$ and distance metric $d$
2: Initialization: $T$ = hyperparameter_configuration_generator($n$), performance list $Z$ = empty list [ ], learning curve list $X$ = empty list [ ]
3: while free worker do
4:     $\theta$ = get_new_one($T$)                    ▷ Return a new configuration for training.
5:     check trigger = get_check_trigger()
6:     for every check point $p \in C$ do
7:         learning curve $\gamma$ = update_lc($\theta, p$)    ▷ Update the learning curve until meeting the checkpoint $p$.
8:         send_to_supervisor($\gamma$)
9:         stop_training_trigger = receive_from_supervisor()
10:    $z$ = get_final_performance($\theta$)
11:    send_to_supervisor($\gamma$, $z$)
12: for supervisor worker do
13:     for $\gamma$ = receive_from_worker() do
14:         trigger = = check($Y, \gamma, d, s$)       ▷ This function refers to Algorithm 2.
15:         send_to_worker(trigger)
16:     for $\gamma, z$ = receive_from_worker() do
17:         $X, Z$ = update($X, Z, \gamma, z$)
18:         check trigger = [length($Z$) > $n * r$]
```

3.4.2 Combination with other selection algorithms

Another potential improvement of LCM is the combination with other hyperparameter selection algorithms. As shown in Figure 3.1, selection algorithms and early stopping algorithms are two independent categories, which are both necessary in hyperparameter tuning. In the above experiments, we only considered the combination of LCM and random search. Nevertheless, besides this basic selection algorithm, random search, there are other advanced selection algorithms, such as Bayesian Optimization. Some of them have good performances and are applied widely. In consideration of the complexity of hyperparameter tuning and the uniqueness of every tuning problem, we cannot predict the performance of such combinations. Further implementation and related experiments are required.

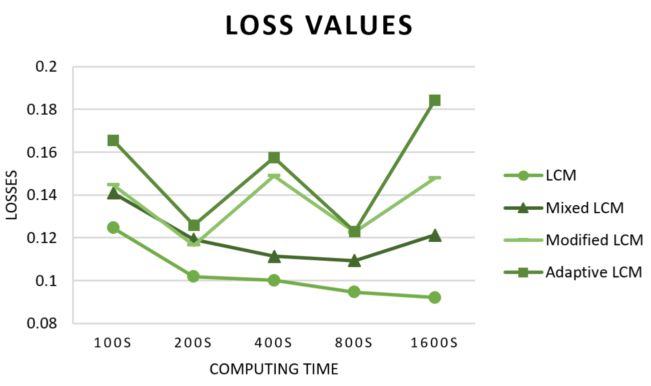
3.4.3 Ultraparameters

As hyperparameters are required for parameter tuning in machine learning training, there are also some parameters which are needed in hyperparameter tuning algorithms, 'ultraparameters'. Considering our little understanding of hyperparameters because of their complicated and sensitive properties, we know even less about ultraparameters. In the CIFAR10 group, setting the number of configurations to be 500 causes some unexpected instability. Somtimes, LCM gets worse results with the increase of computing time. Since hyperparameters and underlying networks are complex, number of trials in the accumulation stage becomes relatively small. It is also possible that initial picks are in a bad comaparison candidates and consequently, LCM is led in the wrong direction. How to decrease the number and complexity of such ultraparameters and even implementing so called auto-tuning are critial for hyperparameter tuning. Otherwise, we always need new algorithm to tune the parameters generated by previous tuning algorithms.

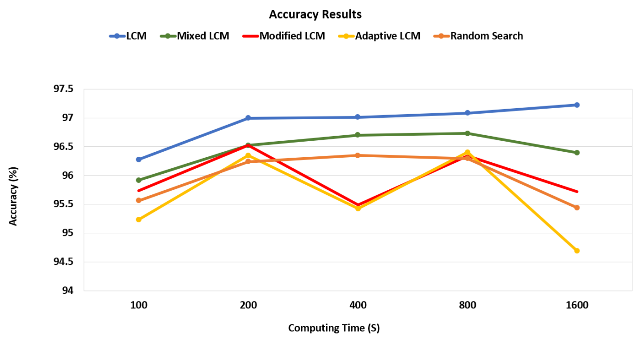
3.4.4 Some Inconclusive Work

After initial implementations of LCM came several 'improved' versions of LCM. Mixed LCM takes computing time into consideration. Instead of comparing the rank percentage of predicted performance, Mixed LCM compares a linear combination of relative time cost and the rank percentage of predicted performance. Modified LCM tries to modify the method of getting the predicted performance, where distances are normalized and predicted performance is the dot product of normalized distances and corresponding final performances. Adaptive LCM is a combination of LCM and an algorithm for changing learning rates during the training based on learning curves.

Then we repeat the fixed-computing-time experiments in the MNIST group. The results are shown in Figure 3.6. None of those revised algorithms implies any potential improvement. Even compared with random search, Modified LCM and Adaptive LCM perform worse. Mixed LCM shows stable outperformance if compared with random search. Considering the negligible differences between needed computing time for different configurations, we need to conduct more complicated experiments to examine potential validity of Mixed LCM.



(a) Loss Results



(b) Accuracy Results

Figure 3.6: Experiment Results

Population Based Training with MagmaDNN

4.1 Natural Models and Bottlenecks

The recent history of scientific innovation is rich with examples of imaginative minds using nature as inspiration for exploration and discovery. Evolutionary optimization algorithms use natural models to inspire a particular approach to traversing a search space in order to minimize some objective function. One classic case is the Particle Swarm Optimization algorithm, inspired by the swarming behavior of bees [1]. These evolutionary optimization algorithms, such as the hyperparameter tuning algorithm Population Based Training (PBT), since they are often modeling several distinct actors operating concurrently through time, are particularly amenable to parallel implementations [3]. PBT's current most popular implementation, that of its originators at UC Berkeley, while suitable for consumer scale neural network applications, cannot efficiently utilize the thousands or tens of thousands of compute nodes on a supercomputer, such as ORNL's Summit. The scalability bottleneck encountered in this otherwise effective hyperparameter optimization algorithm is the impetus behind implementing PBT from the ground up using MagmaDNN, the developers of which have had supercomputer scale, distributed, high performance computing at front of mind since its inception.

4.2 Learning Rate and Learning Rate Decay

As it relates to PBT, the current work used the optimization of learning rate and learning rate decay as a case study of the malleability of MagmaDNN's hyperparameter tuning functionality. This particular class of hyperparameters was chosen as a focus due to an apparent consensus in the literature that, holding network architecture constant (a necessity due to an inherent trade-off made by PBT discussed further below), optimizing the learning rate made the greatest contribution to neural network convergence [1][3]. While the phrase "learning rate" is suggestive, it will require some deeper clarification in order to fully explain the experimental results obtained.

In all the neural network training trials conducted in the PBT experiments, stochastic gradient descent (SGD) was used to traverse the parameter space of network weights in order to find the weight configuration that would minimize training error. SGD is an extension of classical gradient descent optimization wherein a differentiable convex space, meaning a differentiable space with a single minimum, is traversed in the direction of steepest descent. The iterative gradient descent process is captured in the following equation,

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n),$$

where we see that $\mathbf{x}_{n+1}$ the $n+1^{th}$ position in the parameter space is determined by $\mathbf{x}_n$, minus the gradient ∇, scaled by γ_n, of the objective function F evaluated at $\mathbf{x}_n$. The γ_n term is called the step size since it determines how far through the parameter space each iterative step travels [6].

The SGD process is similar, except that the equation's right hand side is perturbed by some, typically small, additional stochastic epsilon vector. This extra randomness makes SGD suitable for non-convex optimization, like that done in neural network training, since the extra epsilon term as seen here,

$$\mathrm{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n) + \epsilon_n,$$

can prevent the search from settling into a local minimum. The γ_n term of the SGD equation, in the context of neural networks, is known as the learning rate [1]. It is always non-negative, since otherwise movement would occur in the wrong direction, non-zero, since otherwise no movement would occur at all, and less than one, since otherwise the process would diverge for a generic objective function.

Intuitively, one may see that a constant learning rate, even between zero and one, may still result in convergence problems. For illustration consider the scenario where you are visiting a friend across the country at their new apartment in Los Angeles or New York. First you would move at jet speed, then train speed, then taxi speed, and finally walking speed. If you never got out of the cab you would just drive back and forth in front of the apartment, but if you did not travel quickly enough at the beginning of your trip, you may never make it to your friend's city at all. The intuition behind this illustration is borne out in the existing research on neural network learning rate. The generally optimal behavior is to have a learning rate which decays over time, with too high a learning rate associated with oscillations in the network weights and too low with a network converging too slowly [1]. The question then is how to determine γ_{n+1} given γ_n. A common scheme is to apply a constant decay factor α, however indexing α by the iteration number like so,

$$\gamma_{n+1} = \alpha_n \gamma_n,$$

will indicate that some of the PBT experiments do update the α value at different times of the network training. Indeed, these values, the learning rate γ and the decay rate α, are the hyperparameters that were optimized in the experiments documented below.

4.3 Algorithm and Implementation

The principle behind the PBT algorithm is relatively straightforward. A population of neural networks begins training with randomized hyperparameters and network weights. At a determined point during the training, the fitness of the networks is ranked according to some metric. Then the least fit networks copy the hyperparameters and network weights from the most fit networks. This is termed the exploit phase. Next, the copied hyperparameters are perturbed according to some distribution. This is what the PBT authors call the explore phase. This process of train, exploit, explore is then repeated until desired convergence or some other predetermined stopping condition [3]. The original PBT developers' psuedocode of the algorithm can be found in the appendix.

In the particular implementation developed for this project, the population of neural networks, termed "workers," was made up of instances of MagmaDNN training on the MNIST handwritten digit dataset. The network structures were held constant across experiments. These network architectures consisted of three fully connected layers with bias, along with sigmoid activation functions. The fitness ranking metric used in all PBT experiments was training accuracy. While some preliminary experiments were completed (analyzed below) using file input/ouput for the communication of accuracies, network weights, and hyperparameter values, the final implementation uses MPI calls to send and receive data between workers. The stopping condition in all experiments was that training would cease after five epochs, i.e. five full training cycles on the entire dataset.

MagmaDNN, written in C++, is an open source project, and accordingly, its source code was available for free use and extension. To implement PBT using MagmaDNN, the proprietary neural network class was customized to include functionality necessary for the PBT algorithm, in particular inter-worker communication, and evolutionary hyperparameter perturbation. A walkthrough of some of this custom network's code will clarify some implementation details. Focus will be given to modifications made to the proprietary code. Readers interested in the original code can find it at https://bitbucket.org/icl/magmadnn.

Before training begins each network samples their hyperparameters from a given distribution.

```
/* randomly initialize learning rate with a value between .2 and .0001 */
std::srand (time(NULL) ^ getpid());
learning_rate = (rand() % 2000 + 1) * .0001;
(static_cast<optimizer::GradientDescent<T> *> (optim))->set_learning_rate(learning_rate);

/* randomly initialize learning rate decay ratio with a value between 1.01 and .99 */
lr_decay = 1.01 - (rand() % 20) * .001;
```

After a partial training period, the length of which is determined by the evolution pace variable, each network enters the exploit phase.

```
/* check if enough iterations have passed for exploit cycle to start */
if (i != 0 && i % evolution_pace == 0) {
    time_to_exploit = true;
}

/* exploit phase: compare and update hyperparameters */
if (time_to_exploit) {
```

Inside the exploit if statement, accuracies are communicated between workers using MPI Allgather, which updates a vector of doubles for each network instance, which contains in its slot with index n the accuracy of the worker with MPI rank n.

```
/* record current worker's accuracy */
worker_accuracies[worker_num] = n_correct/((double)i*this->model_params.batch_size);

/* communicate accuracies */
MPI_Allgather(
    MPI_IN_PLACE,
    1,
    MPI_DOUBLE,
    &worker_accuracies,
    1,
    MPI_DOUBLE,
    MPI_COMM_WORLD);
```

Still inside the exploit if statement, worker fitness is ranked using the worker accuracies vector. In this implementation, the most fit quartile, via MPI Send calls, sends its weights and hyperparameters to the least fit quartile; most fit worker to least fit worker, second most fit to second least fit, and so on.

```
/* if current network is fit, send weights and hyperparameters to an unfit worker */
if (worker_rank > (num_workers * 3) / 4) {
    /* send weights */
    for (unsigned int j = 0; j < layers.size(); j++) {
        for (unsigned int k = 0; k < layers[j]->get_weights().size(); k++) {
            Tensor<T>* t = layers[j]->get_weights()[k]->get_output_tensor();
            MPI_Send(t->get_ptr(),
                    t->get_size(),
                    MPI_FLOAT,
                    worker_fitness_ranks[num_workers-worker_rank],
                    333,
                    MPI_COMM_WORLD);
        }
    }
    /* send hyperparameters */
    MPI_Send(&learning_rate, 1, MPI_DOUBLE, worker_fitness_ranks[num_workers-worker_rank], 444, MPI_COMM_WORLD);
    MPI_Send(&lr_decay, 1, MPI_DOUBLE, worker_fitness_ranks[num_workers-worker_rank], 555, MPI_COMM_WORLD);
}
```

Similarly, still inside the exploit if statement, the least fit quartile updates its network weights and hyperparameters via MPI Recv calls. Note that the unfit worker else if statement is still open.

```
/* if current worker is unfit, receive weights and hyperparameters from a more fit worker */
else if (worker_rank <= num_workers - (num_workers * 3) / 4 ) {
        /* receive weights */
                for (unsigned int j = 0; j < layers.size(); j++) {
                        for (unsigned int k = 0; k < layers[j]->get_weights().size(); k++) {
                        Tensor<T>* t = layers[j]->get_weights()[k]->get_output_tensor();
                                MPI_Recv(t->get_ptr(),
                                        t->get_size(),
                                        MPI_FLOAT,
                                        worker_fitness_ranks[num_workers-worker_rank],
                                        333,
                                        MPI_COMM_WORLD,
                                        MPI_STATUS_IGNORE);
                        }
                }

    /* receive hyperparameters */
    MPI_Recv(&learning_rate, 1, MPI_DOUBLE, worker_fitness_ranks[num_workers-worker_rank], 444, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&lr_decay, 1, MPI_DOUBLE, worker_fitness_ranks[num_workers-worker_rank], 555, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Inside the unfit worker else if statement, meaning if the worker has copied weights and hyperparameters from a fitter worker, the hyperparameter space is explored via a perturbation of the copied hyperparameters chosen from a given distribution.

```
/* evolve hyperparameters */
else {

    if (rand() % 2 == 0) {
        learning_rate = learning_rate * 0.8;
    }
                else {
        learning_rate = learning_rate * 1.2;
    }

    lr_decay = lr_decay * ( 1.1 - ( rand() % 20 ) * .01 );
}
```

The explore phase ends by leaving the unfit worker else if statement. The exploit if statement ends, returning the network to its training for loop. During these exits, the optimizer's learning rate variable is updated, and the exploit flag is reset until another partial training period completes.

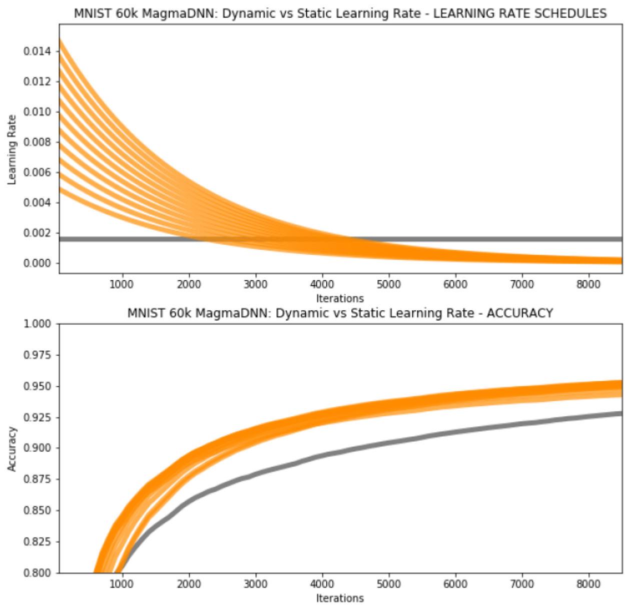
```
        /* update optimizer learning rate */
        (static_cast<optimizer::GradientDescent<T> *> (optim))->set_learning_rate(learning_rate);
    }

    /* reset exploit flag */
    time_to_exploit = false;
        }

/* decay learning rate if it is time */
if (i != 0 && i % 20 == 0 && i % evolution_pace != 0) {
    learning_rate = learning_rate * lr_decay;
    (static_cast<optimizer::GradientDescent<T> *> (optim))->set_learning_rate(learning_rate);
}
```

As noted in section 4.2, PBT makes a trade-off between which hyperparameters it can optimize and the speed of the algorithm. In particular, since PBT completes in a single training regiment, it cannot optimize network architecture. Because the hyperparameters are updating dynamically, the workers that altered their network structure mid-training would almost certainly be the least fit in the next ranking, as their new network components would not have enough time to sufficiently train between rankings. As such, the initial network structure would be selected as the most fit, providing no new information on the network architecture hyperparameters. This was a primary contributing factor towards the design of the experiments which document the optimization of learning rate and decay.

4.4 Analysis and Results

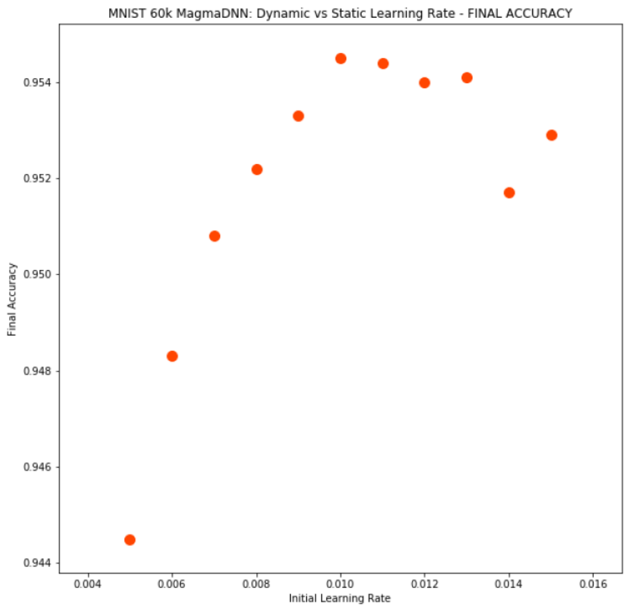
4.4.1 Dynamic Learning Rate

The following graphs show the results obtained from a preliminary experiment, Experiment 0, comparing static to dynamic learning rate. Using MagmaDNN, a network with three fully connected layers was trained on MNIST with stochastic gradient descent. Batch size was constant across trials, at 32 per iteration. The random initial weights of each network were also held constant across trials. Graph A (top) plots learning rate against number of training iterations. It shows a gray line, a trial with static learning rate of 0.0016, while the orange lines plot trials with decaying learning rate with variable initial values. The decay rate was .95, applied every 100 iterations. Graph B (bottom) shows that all of the dynamic trials achieved greater accuracy more quickly. The Y-axis is accuracy, while the X-axis is number of training iterations. These plots affirm, in our case, the benefit of dynamic learning rate.



Experiment 0 - Graph A (top), Graph B (bottom)

Graph C maps the final accuracy of the dynamic trials against their initial learning rate values. The greatest final accuracy is steadily approached from below, as initial learning rate increases. The best final accuracy is achieved with an initial learning rate value of 0.01, and then for greater initial learning rate values, the final accuracy exhibits irregular behavior. These results confirm that lower learning rates, while slowing convergence, contribute to stability, while learning rates that are too high result in oscillations and unpredictability.



Experiment 0 - Graph C

4.4.2 Adaptive Learning Rate

Experiment 1 and Experiment 2 were conducted using PBT to find the optimal schedule for the learning rate (LR) of a network training with the specifications in Table 4.1.

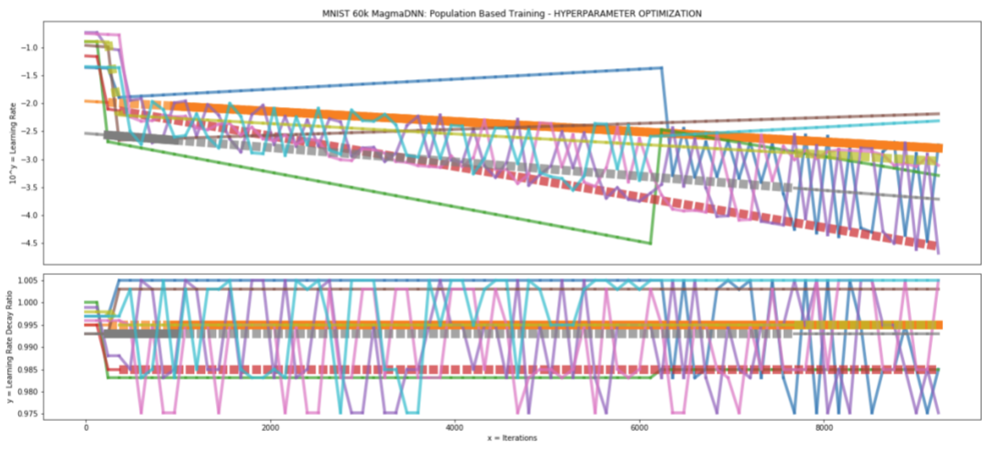
Dataset	MNIST
Layers	3 Dense
Activation	Sigmoid
Optimizer	SGD
Epochs	5
Batch Size	32

Table 4.1: Experiments 1 and 2 - Network Specifications

Experiment 1

LR Sampling Distribution	Uniform Random between .0001 and .2
LR Decay Ratio Sampling Distribution	Uniform Random between .99 and 1
LR Decay Pace	Every 20 iterations
Evolution Pace	Every 120 iterations
LR Perturbation Distribution	1.2 and .8 equally likely
LR Decay Ratio Perturbation Distribution	.99 and 1.01 equally likely

Table 4.2: Experiment 1 - PBT Settings



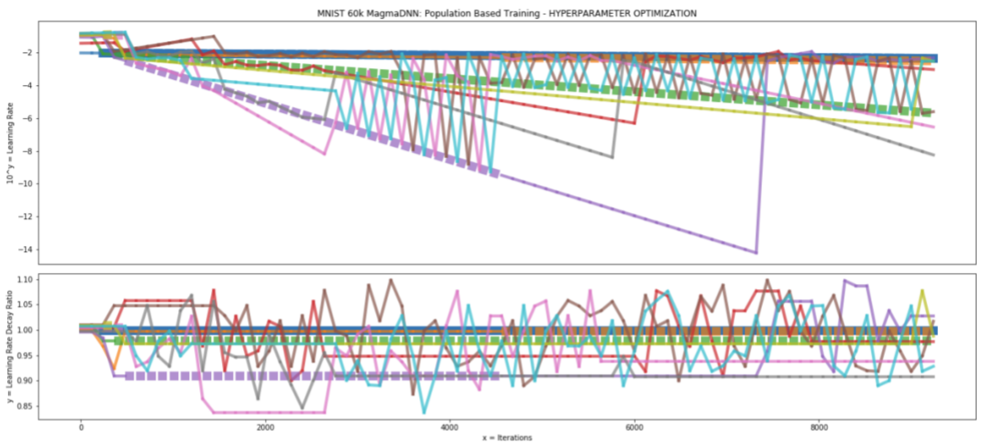
Experiment 1 - Graph D (top), Graph E (bottom)

Graph D plots the learning rates across training iterations of a population of ten workers, instances of MagmaDNN, while Graph E plots the same population's learning rate decay factor. The thick lines signify the workers in the most fit quartile, in this case the three networks with the highest training accuracy. The most fit worker's hyperparameter schedule is plotted with a thick bold line, while the second and third performers lines are dotted. Graph D clearly shows that the many of the random initial values for the population's learning rates quickly drop, before the end of the first training epoch, culling the workers with learning rates that are too high. The behavior of the blue and green workers in Graph D demonstrates a similar culling behavior on a longer time scale, but with the green worker's learning rate sinking too low. By the final training epoch the orange and red workers are delineating an optimal channel for the learning rate between .01 and .0001, with most other members of the population oscillating between them. In both graphs we can see that while the grey worker was the top performer for most of the first epoch, the orange worker soon becomes and remains the most fit. This is likely due to the slightly more severe decay rate of the grey worker. Since the decay behaves exponentially (note that Graph D is log scale along the y-axis), when the decay factor sinks below a value on the order of .99, the worker's learning rate will tend to be sent too low, too quickly.

Experiment 2

LR Sampling Distribution	Uniform Random between .0001 and .2
LR Decay Ratio Sampling Distribution	Uniform Random between .98 and 1.01
LR Decay Pace	Every 20 iterations
Evolution Pace	Every 120 iterations
LR Perturbation Distribution	No perturbation
LR Decay Ratio Perturbation Distribution	Uniform Random between .9 and 1.1

Table 4.3: Experiment 2 - PBT Settings



Experiment 2 - Graph F (top), Graph G (bottom)

Experiment 2 was an ablation experiment wherein the learning rate was not allowed to evolve directly, rather only through its decay ratio. Considering the learning rate as one dimension in the hyperparameter space, while the least fit workers did copy the learning rates of the most fit, the workers did not explore the dimension in the sense laid out section 4.3. This idea was inspired by the classical optimal control example of an inverted pendulum on a cart. In one experiment the pendulum's output state is a measure of the angle of the pendulum, in another it is only allowed to output the angular velocity of the pendulum. It can be shown that under generic circumstances, with control over the velocity of the cart, both experiments result in stability. Similarly, this ablation experiment results in the learning rates settling into the same feasible channel between .01 and .0001 as Experiment 1. While there is more variablility in the learning rates of the first few training epochs, the aforementioned culling behavior results in convergence of the hyperparameters to value ranges nearly identical to Experiment 1.

Conclusions and Future Work

The experimental results of this research project show that dynamic and adaptive learning rate optimization, such as that deployed in our MagmaDNN PBT implementation, improves the convergence of neural networks. However, the fact that PBT inherently cannot without significant modification effectively optimize network structure can serve as an impetus for further research into alternative hyperparameter optimization schemes such as the random accretative network structure optimizer being developed by Massimiliano Pasini at ORNL. Furthermore the project demonstrated that early stopping hyperparameter tuning algorithms, such as LCM, can compete with standard benchmarks like Random Search.

Another potentially fruitful avenue for future developments may be the implementation of more custom MagmaDNN classes to explore the tuning of convolutional neural network hyperparameters. As well, implementing LCM using the MagmaDNN framework would likely improve its performance and scalability. Finally, the completion of an implementation of MagmaDNN PBT utilizing OpenDIEL's distributed workflow system, would allow the project to be scaled to supercomputer applications, like the electron microscopy research that originally motivated the direction of much of the work done for this project.

References

[1] Goodfellow et al. *Deep Learning.* 2016.
https://www.deeplearningbook.org/

[2] Bergstra and Bengio. *Random Search for Hyperparameter Optimization.* 2012.
http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

[3] Jaderberg et al.. *Population Based Training of Neural Networks.* 2017.
https://arxiv.org/pdf/1711.09846.pdf

[4] Wong et al. *Distributive Interoperable Executive Library (DIEL) for Systems of Multiphysics Simulation.* 2014.
https://cfdlab.utk.edu/openDIEL/PDCAT-DIEL-submitted.pdf

[5] Karpathy. *Convolutional Neural Networks.* 2018.
http://cs231n.github.io/convolutional-networks/

[6] Boyd and Vandenberghe. *Convex Optimization.* 2009.
https://web.stanford.edu/ boyd/cvxbook/bv_cvxbook.pdf

[7] Li et al. *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization.* 2018.
https://arxiv.org/pdf/1603.06560.pdf

[8] Li et al. *Massively Parallel Hyperparameter Tuning.* 2018.
https://arxiv.org/pdf/1810.05934.pdf

Appendix

Supplementary Articles

- Stanley and Miikkulainen. *Evolving Neural Networks through Augmenting Topologies*. 2002
- Young et al. *Evolving Deep Networks Using HPC*. 2017.
- Li et al. *A Generalized Framework for Population Based Training*. 2019.
  https://arxiv.org/pdf/1902.01894.pdf
- Zhang et al. *Tropical Geometry of Deep Neural Networks*. 2018.
  https://arxiv.org/pdf/1805.07091.pdf
- Golovin et al. *Google Vizier: A Service for Black-Box Optimization*. 2017.
  https://storage.googleapis.com/pub-tools-public-publication-data
  /pdf/bcb15507f4b52991a0783013df4222240e942381.pdf
- Curren et al. *Workflow and Direct Communication in the Open Distributive Interoperable Executive Library*. 2015.
  https://www.jics.tennessee.edu/files/images/csure-reu/2015/TANNER-NICK/Report.pdf
- Asemota et al. *Interfaces for openDIEL*. 2018.
  https://www.jics.tennessee.edu/files/images/recsem-reu/2018/opendiel/Report.pdf
- Rodriguez-Mier. *A Tutorial on Differential Evolution with Python*.
  https://pablormier.github.io/2017/09/05/a-tutorial-on-differential-evolution-with-python/
- Golovin et al. *Google Vizier: A Service for Black-Box Optimization*. 2017.
  https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/46180.pdf

PBT psuedocode

Algorithm 1 Population Based Training (PBT)

```
 1: procedure TRAIN($\mathcal{P}$)                                        ▷ initial population $\mathcal{P}$
 2:     for $(\theta, h, p, t) \in \mathcal{P}$ (asynchronously in parallel) do
 3:         while not end of training do
 4:             $\theta \leftarrow \texttt{step}(\theta|h)$        ▷ one step of optimisation using hyperparameters $h$
 5:             $p \leftarrow \texttt{eval}(\theta)$                                  ▷ current model evaluation
 6:             if $\texttt{ready}(p, t, \mathcal{P})$ then
 7:                 $h', \theta' \leftarrow \texttt{exploit}(h, \theta, p, \mathcal{P})$   ▷ use the rest of population to find better solution
 8:                 if $\theta \neq \theta'$ then
 9:                     $h, \theta \leftarrow \texttt{explore}(h', \theta', \mathcal{P})$          ▷ produce new hyperparameters $h$
10:                     $p \leftarrow \texttt{eval}(\theta)$                              ▷ new model evaluation
11:                 end if
12:             end if
13:             update $\mathcal{P}$ with new $(\theta, h, p, t + 1)$                       ▷ update population
14:         end while
15:     end for
16:     return $\theta$ with the highest $p$ in $\mathcal{P}$
17: end procedure
```