Stage-based Hyper-parameter Optimization for Deep Learning

Ahnjae Shin, Dong-Jin Shin, Sungwoo Cho, Do Yoon Kim,
Eunji Jeong, Gyeong-In Yu, Byung-Gon Chun
Seoul National University
{aj.shin,dongjin.shin}@spl.snu.ac.kr,
{sungwoocho,ddoyoon,ejjeong,gyeongin,bgchun}@snu.ac.kr

Abstract

As deep learning techniques advance more than ever, hyper-parameter optimization is the new major workload in deep learning clusters. Although hyper-parameter optimization is crucial in training deep learning models for high model performance, effectively executing such a computation-heavy workload still remains a challenge. We observe that numerous trials issued from existing hyper-parameter optimization algorithms share common hyper-parameter sequence prefixes, which implies that there are redundant computations from training the same hyper-parameter sequence multiple times. We propose a stage-based execution strategy for efficient execution of hyper-parameter optimization algorithms. Our strategy removes redundancy in the training process by splitting the hyper-parameter sequences of trials into homogeneous stages, and generating a tree of stages by merging the common prefixes. Our preliminary experiment results show that applying stage-based execution to hyper-parameter optimization algorithms outperforms the original trial-based method, saving required GPU-hours and end-to-end training time by up to 6.60 times and 4.13 times, respectively.

1 Introduction

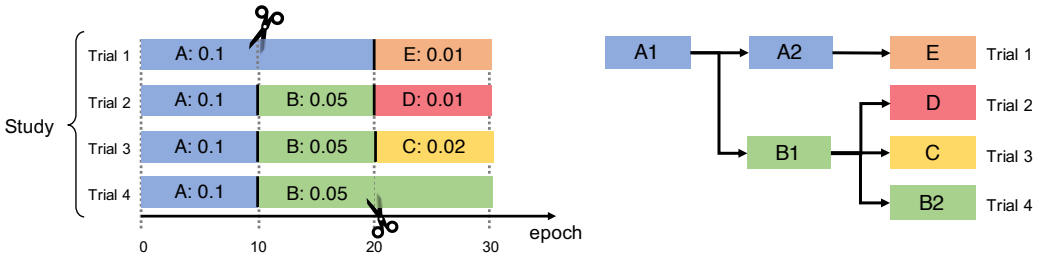
Deep learning (DL) models have made great leaps in various areas including image classification [12, 21, 7], object detection [25], and speech recognition [11, 3]. However, such benefits come at a cost; training DL models requires heavy datasets and long computations which may take up to a week [29] even on a hundred of GPUs [29]. This cost becomes more significant when we take hyper-parameter optimization into account. Investigating the hyper-parameter search space often requires hundreds to thousands of trainings with different hyper-parameter settings [22]. Consequently, naively running hyper-parameter optimization requires an exceedingly large number of GPUs, and it is crucial to explore the hyper-parameter search space as efficiently as possible.

A hyper-parameter optimization job trains and evaluates the target DL model multiple times, each with a different configuration. Each training sub-procedure is identified by its unique configuration. We use the term study to refer to the job and the term trial to refer to the training sub-procedure[1]. As an example, Figure 1(a) depicts a study with four trials. Each trial has different learning-rate values. The first trial trains a DL model with 0.1 learning-rate and switches to 0.01.

Training modern DL models to reach state-of-the-art accuracy requires changing hyper-parameter values in the midst of training, as they target minimizing high-dimensional, non-convex loss functions. Hence, a hyper-parameter configuration can be regarded as a sequence of values. Examples include learning-rate [12, 15, 26, 27, 10, 17, 30, 13, 4], drop-out ratio [5], optimizer [29], momentum [31],

[1]The terms study and trial come from *Vizier*[9].

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.



(a) study with four trials

(b) stage-tree

Figure 1: Figure 1a represents a study with four different trials each with a different learning-rate sequence. Trial 1 has sequence A->E, and trial 2 has sequence A->B->D. The trials have common prefixes. Trials 2 and 3 share subsequence A in common. If we split subsequence A of trial 1 into two parts (A1 and A2), the former part is identical to the subsequence trials 2, 3, and 4 have. We can merge the common subsequences and represent the trials as a tree, as shown in Figure 1b. Note that even if E and D have the same value, they cannot be merged and should be considered as different subsequences, because they do not share the same prefix (A1->A2 for E, A1->B1 for D).

batch size [28], image augmentation parameters [14], training image input size [18], input sequence length [8], and network architecture parameters [18].

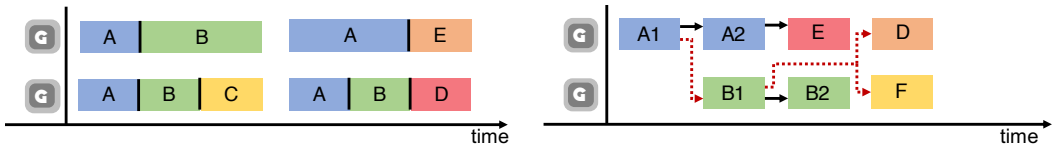
Existing approaches for hyper-parameter optimization systems [23, 9, 6, 20] simply execute multiple trials sequentially, or support launching multiple trials in parallel to utilize multiple GPUs and machines. However, we observe that such trial-based execution strategy does not exploit an important characteristic of hyper-parameter optimization: a hyper-parameter configuration is a *sequence*, not a single *value*. As in Figure 1a, a trial is a sequence of homogeneous *stages*, where a stage is a span of a trial with constant hyper-parameter values. For instance, the second trial is composed of three stages, and the last trial is composed of two stages. If the hyper-parameter optimization system knows where each stage starts and ends, we can train duplicate stages only once, and reuse the computed result multiple times. Note that, continuous-valued hyper-parameter sequences are also eligible for merging. For instance, trials that use learning rate warmup[10], or cyclic learning rate strategy[26] have potentially identical prefixes. When applying warmup, the learning rate linearly increases for a few steps and decreases afterward. Consequently, two different sequences can have the warmup period as a common prefix, each with a different decaying strategy. The same logic applies to cyclic learning rates. Each cycle need not be identical to each other; usually, later cycles have smaller ranges. Hence, two different sequences may have identical cycles. Therefore, this observation motivates a new execution strategy for hyper-parameter optimization, which manages its workload based on stages, not trials. Such fine-grained execution can cut down GPU resource usage and lead to shorter end-to-end training time.

In this paper, we present a stage-based execution strategy that seeks higher computational and resource efficiency. With the strategy, we can exploit the characteristics of trials or studies by inspecting their stages, while trial-based execution treats trials as black boxes. This introduces the opportunity to remove redundant computations across trials, and thereby improves the efficiency of hyper-parameter optimization.

Our experiments with three studies show that stage-based execution can reduce GPU-hours and end-to-end training time compared to the trial-based execution. When tuning learning rate, it can save end-to-end training time up to 2.98 times, and GPU-hours up to 5.73 times. When tuning batch size, it can save end-to-end training time up to 4.13 times and GPU-hours up to 6.60 times.

2 Stage-based Execution

We propose a stage-based execution strategy for using GPU resources efficiently. In this section, we describe how a stage-based system represents trials internally, and how it executes stages.



(a) trial-based execution

(b) stage-based execution

Figure 2: We can execute the trials in Figure 1a in two ways: trial-based, and stage-based. Trial-based execution, shown on the left, executes each trial independently on a GPU. As there are two GPUs, each trial is assigned one GPU. Stage-based execution takes advantage of the stage-tree in Figure 1b. The edges of the tree are shown in the figure as red dotted or black solid lines.

2.1 Stage-tree

Given a search space composed of multiple trials as in Figure 1a, instead of directly executing each trial, we express the search space as a stage-tree, as in Figure 1b. In the example, there were four trials which were split and merged into seven stages. In Figure 1b, each node is a stage; it has homogeneous hyper-parameter values with a fixed range of iterations[2]. As there were four trials, there are corresponding four leaf stages in the stage-tree. Therefore, every path from a root node to a leaf node corresponds to a trial with a unique hyper-parameter sequence.

The stage-tree is an internal representation, and therefore is not exposed to the user. Users submit a study composed of trials and they are automatically split into homogeneous configurations by the system. Then, to add a trial into the stage-tree, the system traverses the stage-tree built so far to find the longest common prefix between the given trial and the current stage-tree. In case the number of training steps are different for two identical configurations, it splits the larger sequence to maximize the length of common prefix. Then, it appends the remaining unmatched subsequence of the new trial to the tree. Note that in this case each homogeneous value corresponds to a list of stages, as in Figure 1b. The first homogeneous value (0.1) of Trial 1 and Trial 2 is identical, but as one sequence is longer than the other, it splits the longer one into stages A1 and A2.

2.2 Stage Execution

To use GPU resources more efficiently and reduce end-to-end time, we employ stage-based execution based on the stage-tree built from trials. With trial-based execution, we treat each trial as a black box and simply launch the trial on an idle GPU, as shown in Figure 2a. On the other hand, when we use the stage-based execution strategy, each stage becomes a unit of scheduling, thus avoiding redundant computation.

Figure 2b shows an example of stage-based execution. We can see that stage-based execution improves both GPU-hours and end-to-end time compared to trial-based execution. In the figure the edges indicate a parent-child relationship between the two connected nodes in the stage-tree. Each child node starts execution by initializing its weights using the checkpoint of its parent node. Therefore, each edge represents a data dependency relationship; every child starts off from where its parent has ended. For the red dotted lines, initializing weights from a checkpoint is necessary. However for the black solid lines, since the connected nodes run consecutively in the same GPU, checkpoint loading is unnecessary. For example, executing A1 and A2 consecutively does not require checkpoint loading, but executing E and D in series in the same GPU needs loading of checkpoint from B1 before running D.

We keep track of stages in the tree that have been executed and are currently being executed. The children of stages that have been executed become candidates for execution. When deciding the next stage to execute among the candidates, we consider the priorities of trials that many hyper-parameter optimization algorithms specify. For example, existing algorithms such as the Asynchronous Successive Halving Algorithm (ASHA)[22] not only specify what configuration to run, but also in what order. Given the next stage to execute, we decide how many GPUs the stage needs to be executed. In order to do so, we profile the GPU usage characteristics of stages and estimate the resource requirements of new stages based on the history of previously executed stages. If the

[2]Number of iterations can be number of epochs or steps depending on user code.

stage fails because our prediction turns out to be wrong, we then re-launch the stage by increasing the number of GPUs until we find a sufficient number of GPUs to execute the stage.

Before training each stage, the system spawns multiple workers, each responsible of one or more GPUs, and run inside a containerized environment. Each container exclusively holds multiple GPUs from the same node, and runs one worker process. Note that trials from a single study share a common environment, hence it is perfectly valid to reuse both the container and the worker process. As a result, we can make the system not launch new containers or workers for stages with the same resource requirements. When running a stage with different resource requirements, we need to merge or split existing workers. For example, if all existing workers only have one GPU in control, and the next stage requires two GPUs, the system destroys two free workers (and containers) and creates a new worker with two GPUs. To avoid communication overhead in distributed training, the system tries to select and merge workers from the same node.

3 Discussion

3.1 Just-fit Resource Allocation

By splitting trials into stages, besides the advantage of reducing computation, the system is able to provide high resource utilization by allocating the right amount of resource. DL training jobs require varying amounts of resources according to the hyper-parameters, and trials require different amounts of resources through its life cycle with respect to its hyper-parameter sequence.

For instance, when training a model using dynamic batch size [28] or optimization algorithms [19], the memory requirements of a trial vary. In such situations, trial-based execution should allocate the maximum amount of resources the trial may use, which incurs lots of idle resources. On the contrary, we can avoid such inefficiency by allocating GPU resources for each stage. Since each stage has a homogeneous behavior, we are able to allocate just the right amount of resources to each stage.

3.2 Multi-study Optimization

Multiple studies with an identical model and dataset can run on the same DL cluster [2]. In this case, if we use stage-based systems, we can expose the search space history explored by previous study to other studies. Then, trials from one study may exploit the search space that other studies have already explored by reusing the results. Hyper-parameter optimization algorithms that require prior information to work properly can benefit from using the search space of previous runs. To support multi-study optimization in our system, we can use an executor layer. There are a set of studies and a set of executors, and the system maps each study to an executor. Studies that have the same dataset and model are routed to the same executor. A router component will send submitted trials to its corresponding executor based on which study it came from. This way, we can give each executor its stage-tree and resource pool, as merging will not occur between executors.

3.3 Supporting Continuous Search Space

For our stage-based execution system to perform well, the trials sampled by hyper-parameter optimization algorithms should have overlapping prefixes. If the user declares hyper-parameter search space as a set of discrete values (e.g., grid search), we have overlapping prefixes more or less depending on the sampling algorithm. However, when using random search over a continuous domain, sampled values are hardly ever identical. In other words, the possibility of benefiting from overlapping prefixes is close to zero. This is a typical case in random search or bayesian optimization over a continuous domain, where our system may provide only minimal gain.

In cases when there are no overlaps among trials, our system has identical behavior with previous systems. For example, when running algorithms like random search or bayesian optimization, sampled trials can have very low overlaps, and such algorithms do not hinder system performance. Rather, our system gives optimization algorithms and users an opportunity to reduce computation in training many trials. The user can further modify the search space in a way that maximizes prefix overlaps, as more overlaps will allow the system to explore the space better within the same time or financial budget. One may think these constraints improves efficiency at the cost of the final accuracy. However, improving the time- or resource- efficiency of hyper-parameter optimization is also important for

Hyper-parameter	Values
initial learning rate	0.5, 0.2
decay rate	0.2, 0.1
decay epoch 1	40, 60, 80
decay epoch 2	40, 60, 80
decay epoch 3	40, 60, 80
batch size	128
optimizer	SGD
momentum	0.9
weight decay	1e-4

(a) Resnet

Hyper-parameter	Values
initial batch size	128
increase rate	5
increase epoch 1	30, 60
increase epoch 2	30, 60
increase epoch 3	30, 60
learning rate	0.1
optimizer	SGD
momentum	0.9
weight decay	5e-4

(b) WideResnet

Table 1: Hyper-parameter space

improving the quality of the final model. Since we can try training with more hyper-parameter configurations within the same budget, the final model may actually become better than before, and being able to continue training from existing checkpoints will help in finding network weights that perform high validation accuracy. Indeed, algorithms like random search or bayesian optimization are designed with the same philosophy in mind; they selectively choose hyper-parameters to explore the search space efficiently.

4 Experimental Evaluation

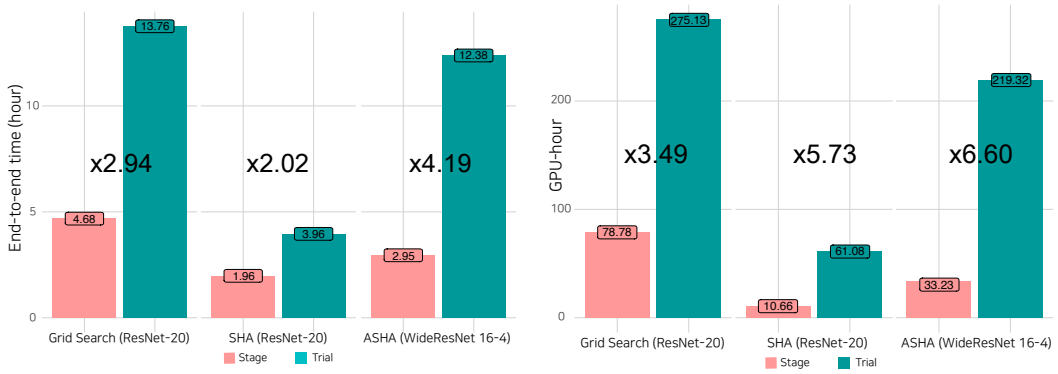
To evaluate our stage-based execution strategy, we implemented a prototype system in Python. We use Docker [24] to launch containers and gRPC [1] as a messaging interface between processes. Estimation of GPU requirements for each stage is done by a simple linear regression model, yet the estimation falls back to one GPU at cold start.

We present three experiments that show how stage-based execution benefits in hyper-parameter optimization. We apply both trial-based and stage-based execution strategies to optimize hyper-parameters and measure the required GPU-hours and end-to-end time. End-to-end time refers to the elapsed time from the experiment's start to end, while GPU-hours signifies the sum of active execution time of all GPUs (e.g., executing a job on 2 GPUs for 10 hours results in 20 GPU-hours). The first two experiments involve tuning the learning rate, and the last experiment involves tuning the batch size. All three cases have reached target accuracy.

The first two cases is training the ResNet-20 [12] model on the CIFAR-10 dataset [21]. Since the CIFAR-10 dataset does not have a fixed *train-validation-test* split, we used a 40K/10K/10K split; 40K images are used to train the model, 10K images are used to tune the hyper-parameters, and the remaining 10K images are used to report the final test error on the model with the lowest validation error. For each case, we used grid search and the Successive Halving Algorithm (SHA) [16] to optimize the hyper-parameters. SHA cuts down running trials by a factor of three at epoch 16 and 64 with their validation accuracy.

As the dimension of hyper-parameter sequences is too large to search, optimizing the hyper-parameter sequence directly is not practiced. Instead, it is common to parameterize the sequence and set the parameters as hyper-parameters. For example, the learning rate is often modeled as a step-decay function and the epoch to decay the learning rate is optimized instead [32, 22]. In addition, to take full advantage of computation reuse, we approximated the continuous search space of hyper-parameters into a discrete search space. The hyper-parameters used are shown in Table 1(a). We only tune the learning rate, and there are three decay periods. Each trial decreases the learning rate three times. Each decay period represents the number of epochs until the next decay. The trial-based and stage-based setups both explored 108 trials that originated from the same hyper-parameter search space. All trials are given maximum 200 epochs of training, unless early stopped.

The experiments were conducted on 4 machines with total 20 NVIDIA GeForce TITAN Xp GPU cards and 2 18-core Intel Xeon E5-2695 @ 2.10 GHz processors with 256 GB RAM. For both



(a) End-to-end time

(b) GPU-hour

Figure 3: Experiment results

trial-based and stage-based experiments, we measure the end-to-end time and GPU-hours (total GPU resource usage) as shown in Figure 3.

The reported test error of ResNet-20 on CIFAR-10 is 8.75% [12]. Our best model's test error reaches 8.24% with only 4/5 of original training data. As shown in Figure 3, for grid search, stage-based system is 2.94 times faster and uses 3.49 times less resource. The SHA has a diminishing effect in reducing end-to-end time. This is because most of the GPUs were idle in stage-based execution. In fact, the GPU resource usage is 5.73 times smaller than trial-based execution.

The third case involves just-fit resource allocation, which was discussed in Section 3.1. We have trained the *WideResnet 16-4* model on the CIFAR-10 dataset. We only tuned batch size; initial value, which epoch to increase, and by how much as in Table 1(b). Total 64 trials were run for both trial-based and stage-based settings. We use the same *train-validation-test* split on CIFAR-10. Grid search and the ASHA[22] algorithm was used to tune the model. Parameters $R = 216, r = 15, eta = 3, s = 0$ are used to run the ASHA algorithm. The results show the validation accuracy reaches 94.8%, where the paper that introduces this strategy[28] has reached 94.4%. In addition, by assigning resources per stage, stage-based approach reduces resource spendings by 6.6 times.

5 Conclusion

In this paper, we have proposed the stage-based execution strategy that splits trials into smaller homogeneous units and removes computational redundancy in the hyper-parameter optimization process. Applying this strategy to hyper-parameter optimization, we are able to reduce end-to-end training time and GPU-hours by up to 4.19 times and 6.6 times, respectively.

As a future work, we plan to evaluate this execution strategy in various state-of-the-art models and datasets using various hyper-parameters. In addition, in this work, we evaluated only discrete-valued sequences. We will expand our research to continuous-valued sequences as well as hyper-parameters such as data augmentation or network architecture parameters. Furthermore, we plan to develop a new hyper-parameter optimization algorithm that can maximize the use of this strategy.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics), the ICT R&D program of MSIT/IITP (No.2017-0-01772, Development of QA systems for Video Story Understanding to pass the Video Turing Test), and Samsung Advanced Institute of Technology.

References

[1] gRPC. `http://grpc.io/`.

[2] *Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads*. USENIX Association, 2019.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[4] Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. In *International Conference on Learning Representations*, 2018.

[5] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2015.

[6] Henggang Cui, Gregory R. Ganger, and Phillip B. Gibbons. Mltuner: System support for automatic machine learning tuning, 2018.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[9] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley, editors. *Google Vizier: A Service for Black-Box Optimization*, 2017.

[10] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[11] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[13] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent.

[14] Daniel Ho, Eric Liang, Xi Chen, Ion Stoica, and Pieter Abbeel. Population based augmentation: Efficient learning of augmentation policy schedules. In *International Conference on Machine Learning*, pages 2731–2741, 2019.

[15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[16] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[17] Diederik P Kingma JLB. Adam: A method for stochastic optimization. *Proc. of ICLR*, 2015.

[18] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation, 2017.

[19] Nitish Shirish Keskar and Richard Socher. Improving generalization performance by switching from adam to SGD. *CoRR*, abs/1712.07628, 2017.

[20] Jinwoong Kim, Minkyu Kim, Heungseok Park, Ernar Kusdavletov, Dongjun Lee, Adrian Kim, Ji-Hoon Kim, Jung-Woo Ha, and Nako Sung. Chopt : Automated hyperparameter optimization framework for cloud-based machine learning platforms, 2018.

[21] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[22] Ang Li, Ola Spyra, Sagi Perel, Valentin Dalibard, Max Jaderberg, Chenjie Gu, David Budden, Tim Harley, and Pramod Gupta. Massively parallel hyperparameter tuning. In *NIPS Workshop on Machine Learning Systems (LearningSys)*, 2018.

[23] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[24] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[25] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, June 2016.

[26] Leslie N. Smith. Cyclical learning rates for training neural networks. *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, Mar 2017.

[27] Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates, 2017.

[28] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018.

[29] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv:1609.08144 [cs]*, September 2016. arXiv: 1609.08144.

[30] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[31] Jian Zhang, Ioannis Mitliagkas, and Christopher Ré. Yellowfin and the art of momentum tuning. *arXiv preprint arXiv:1706.03471*, 2017.

[32] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2017.