Elastic Hyperparameter Tuning on the Cloud

Lisa Dunlap
UC Berkeley
lisabdunlap@berkeley.edu

Kirthevasan Kandasamy
UC Berkeley
kandasamy@berkeley.edu

Ujval Misra
UC Berkeley
ujval@berkeley.edu

Richard Liaw
UC Berkeley
rliaw@berkeley.edu

Michael Jordan
UC Berkeley
jordan@cs.berkeley.edu

Ion Stoica
UC Berkeley
istoica@berkeley.edu

Joseph E. Gonzalez
UC Berkeley
jegonzal@berkeley.edu

ABSTRACT

Hyperparameter tuning is a necessary step in training and deploying machine learning models. Most prior work on hyperparameter tuning has studied methods for maximizing model accuracy under a time constraint, assuming a *fixed* cluster size. While this is appropriate in data center environments, the increased deployment of machine learning workloads in cloud settings necessitates studying hyperparameter tuning with an elastic cluster size and time and monetary budgets. While recent work has leveraged the elasticity of the cloud to minimize the execution cost of a pre-determined hyperparameter tuning job originally designed for fixed-cluster sizes, they do not aim to maximize accuracy.

In this work, we aim to *maximize accuracy* given time *and cost* constraints. We introduce SEER—Sequential Elimination with Elastic Resources, an algorithm that tests different hyperparameter values in the beginning and maintains varying degrees of parallelism among the promising configurations to ensure that they are trained sufficiently before the deadline. Unlike fixed cluster size methods, it is able to exploit the flexibility in resource allocation the elastic setting has to offer in order to avoid undesirable effects of sublinear scaling. Furthermore, SEER can be easily integrated into existing systems and makes minimal assumptions about the workload. On a suite of benchmarks, we demonstrate that SEER outperforms both existing methods for hyperparameter tuning on a fixed cluster as well as naive extensions of these algorithms to the cloud setting.

ACM Reference Format:
Lisa Dunlap, Kirthevasan Kandasamy, Ujval Misra, Richard Liaw, Michael Jordan, Ion Stoica, and Joseph E. Gonzalez. 2021. Elastic Hyperparameter Tuning on the Cloud. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–5, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472883.3486989

1 INTRODUCTION

The performance of deep learning models depends crucially on the choice of training hyperparameters. These hyperparameters affect the runtime and convergence properties of the entire training process. *Hyperparameter tuning* refers to the task of choosing and evaluating several such hyperparameter configurations in order to find a good set of values for the given learning task. Hyperparameter tuning is computationally intensive and typically requires exhaustive enumeration and evaluation of hundreds of candidate hyperparameter configurations. To evaluate each configuration, the corresponding model must be at least partially trained, which can take hours or even days to complete using multiple parallel accelerators.

Many of the widely used hyperparameter tuning methods [20, 21] are based on a single core idea — sequential elimination. Typically, these methods partially train models with several configurations, eliminate the poor performing candidates and continue training the more promising configurations. The freed resources from the early rounds of elimination can be used to evaluate other candidate configurations or accelerate training of the better performing configurations. The more configurations that are evaluated, the better chance we have of evaluating the optimal configuration. However, by applying more resources to the most

SoCC '21, November 1–5, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3486989

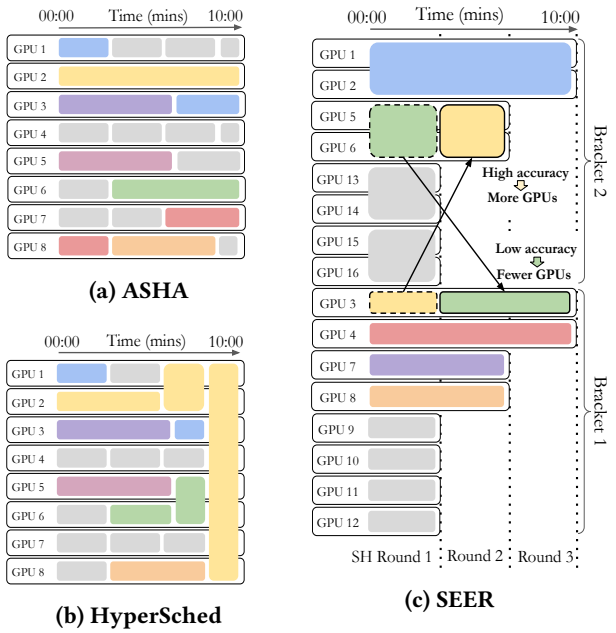


Figure 1: Resource allocation of (a) ASHA [21], (b) Hypersched [22], and (c) SEER (our method) on a 10 minute experiment with a budget of 80 GPU-minutes. Each color represents a distinct promising configuration while grey represents configurations that were eliminated in a short time. The shaded area, which indicates total GPU minutes, is the same for all methods.

promising configurations, we increase the accuracy of the final model even if it is not the optimal configuration.

The hyperparameter tuning process is often time constrained. For example, in settings like click through rate (CTR) prediction where accuracy is critical and data is constantly changing, machine learning engineers wish to ensure that accurate, well-tuned models are released weekly, daily, or even hourly. Similarly, achieving state-of-the-art results in AI research requires rapid iteration of model development and consequently hyperparameter tuning.

Fortunately, hyperparameter tuning exposes multiple degrees of parallelism. Early work in hyperparameter tuning leveraged parallelism (e.g. Fig. 1a) to run multiple concurrent configurations. However, since parallelism is not applied to individual configurations, this approach may not adequately train the best configuration within the allotted time budget. Recent systems work by Liaw et al. [22] explored how to allocate parallelism between different parallel training runs (Fig. 1b) to balance the need to explore different hyperparameter values and adequately train the best identified configuration to maximize accuracy within a deadline. Unfortunately, the degree to which parallelism can be efficiently exploited varies throughout the hyperparameter tuning process. The exploration of multiple concurrent hyperparameter configurations enjoys perfect scaling while the accelerated training of any single configuration often has relatively poor parallel scaling (see Fig. 2). This is usually the result of two factors: first, synchronizing gradient updates at the end of an iteration can lead to bottlenecks which decrease the throughput of the combined workers; second, convergence is faster if gradient updates are performed sequentially (when the loss is computed using the updated model) than if they are done in parallel. This variation in scaling efficiency results in poor utilization of fixed-sized clusters and presents a unique opportunity for cloud computing.

As the majority of machine learning workloads move to the cloud, we are no longer constrained by the assumption of a fixed pool of resources for hyperparameter tuning. Moreover, rather than operating under the constraint of a fixed pool of resources, hyperparameter tuning in the cloud is constrained by a monetary budget that can be spent at a varying rate throughout the hyperparameter tuning process. The elasticity available in the cloud offers a unique opportunity to both reduce the cost of hyperparameter tuning and increase accuracy under a fixed time budget.

The elastic setting, when compared to the fixed cluster setting, allows flexibility in the allocation of resources and consequently allows us to minimize the effects of nonlinear scaling. Since the amount of resources can be scaled up or down, any number of configurations in an initial exploration phase can be executed simultaneously; while this takes the same resource-time as the fixed resource setting, it finishes much sooner. This allows us to allocate more resource-time to the promising configurations without having to parallelize unnecessarily.

In an attempt to adapt fixed-cluster methods to the cloud setting, Misra et al. [24] proposed Rubberband, a system that minimizes the execution cost of a *fixed-cluster* hyperparameter tuning policy (e.g., Successive Halving[14]) while finishing before a given deadline. Rubberband leverages profiling information about the model and cloud environment to simulate the cost and execution time of a given hyperparameter tuning job under different resource allocations, greedily searching for an allocation with the cheapest cost that finishes before a given deadline. While Rubberband enables these methods to be more cost effective, it heavily relies on simulation and does not directly derive a policy for elastic hyperparameter search.

In this paper, we reformulate the hyperparameter tuning problem in terms of time and cost (resource-time) constraints. Rather than elastically scaling the execution of a fixed-cluster policy to reduce cost (i.e., Rubberband), we instead focus on the design of an elastic policy that *given a time deadline and a cost budget, determines the optimal resource-time allocation to maximize accuracy.*

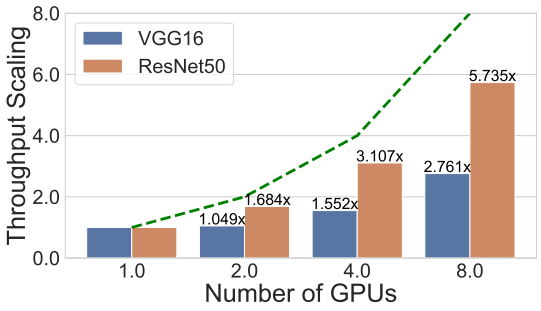


Figure 2: Model throughput of VGG16 and ResNet50 models trained on SVHN VS the number of GPU's per model. While using more GPUs does speed up training, this speed up is non-ideal due to decreased throughput and the non-iterative nature of gradient updates. The dashed green line depicts linear scaling.

We introduce a simple, yet principled, new policy for sequential elimination with elastic resources (SEER) which optimizes the final model accuracy under fixed cost and time budgets. Taking inspiration from prior works, SEER is able to balance exploration and exploitation by executing brackets which run hyperparameter configuration candidates (dubbed *trials*) with different levels of model or data parallelism, enabling SEER to explore a large amount of configurations in the beginning, with the more promising trials getting more resources as the experiment progresses.

We theoretically analyze this new policy and show that it enjoys guarantees similar to those in the fixed-cluster setting [20]. Importantly, unlike previous work [20], we are able to model the sublinear scaling characteristics of a single training job, and prove that SEER is able to do well even without knowledge of the scaling characteristics.

We extend Rubberband [24] with the new SEER policy. This approach leverages the elastic job scaling executor in Rubberband while eliminating the dependence on the simulation and search heuristics needed to adapt prior fixed-resource policies to the elastic setting.

Our contributions are as follows. First, we formalize the setting of identifying and maximally training the best hyperparameter configuration with elastic resources and a fixed budget. Second we develop a new algorithm, SEER, for this setting and explain how SEER can be easily incorporated into the existing Rubberband framework. Third, we evaluate our method on a suite of benchmarks for classification, image segmentation, and natural language processing.

To the best of our knowledge, this is the first work to develop new machine learning methods with the explicit goal of exploiting the flexibility the cloud has to offer in allocating resources.

2 RELATED WORK

Historically, hyperparameter tuning has been viewed as a global optimization problem in the machine learning community. Some examples of such approaches include random search [2], branch-and-bound methods [3, 15], and Bayesian optimization [1, 8, 12, 16, 30].

One popular line of work in hyperparameter tuning algorithms are built on **successive halving** (SH) [14, 17], which eliminates potential configurations in stages, running configurations for a small number of evaluations and allowing the highest performing configurations advance to the next stage. Li et al. [20] designed **Hyperband** which is based on SH. However, the best configuration may not perform well in the early stages, and hence Hyperband runs multiple instances of SH, with each training its configurations to different lengths before elimination. Asynchronous successive halving (**ASHA**) (Fig. 1a), which adapts Hyperband to multiple workers, executes successive halving asynchronously on multiple parallel workers. Finally, Falkner et al. [7] present a Bayesian version of Hyperband (**BOHB**) which assumes a prior on the accuracy over all hyperparameter values and chooses its recommendation based on the posterior.

However, none of the above works explicitly consider how to allocate resources when training the candidates chosen, instead training the chosen models using a single worker for a desired duration. This can be undesirable, especially when operating under tight deadlines, since combining workers can lead to faster training. While it is possible to define a worker to have multiple resources, e.g. by defining multiple GPUs to be one worker , this is a rather blunt tool; all models, including those that are not promising, will be trained using multiple resources, leading to sub-optimal performance.

To our knowledge, the only works which explicitly studies resource allocation for hyperparameter tuning are **HyperSched** [22] (Fig. 1b) and **Rubberband** [24]. HyperSched's algorithm is similar to ASHA; however, as the deadline approaches it avoids testing new configurations and allocates more resources to the promising ones using some simple, yet intuitive, heuristics. While it performs well for the intended setting, as we will show, it can suffer from sublinear scaling and perform poorly when compared to elastic methods. Rubberband [24] is a recent hyperparameter tuning system that aims to minimize the cost of a given hyperparameter tuning job (e.g. a successive halving run) via profiling models and greedily searching for the best allocation of resources. While this work operates in the same setting as ours, Rubberband assumes that the practitioner wants to execute an existing hyperparameter tuning job, while our work aims to design a hyperparameter tuning job that satisfies the time and cost constraints while maximizing accuracy of the final model. That is, Rubberband requires a practitioner to design the job

in the form of training plans, which might require domain expertise. On the other hand, our policy only requires the time and budget constraints as input.

Along with HyperSched and Rubberband, there are multiple frameworks for hyperparameter tuning. Google Vizier [10], Determined AI [21], and Kitlab [9] target hyperparameter tuning in distributed setting. In comparison to our system, Google Vizier and Katlib do not incorporate time and cost constraints, which are key characteristics of our problem. Determined AI supports executing ASHA on pre-emptible instances but their algorithm does not account for resource elasticity. Our framework is built on Tune [23], which is in turn built on Ray [25]; we can therefore leverage the autoscaling features of Ray to rescale a cluster, run distributed hyperparameter tuning experiments, and complete the task within the confines of a given cost and deadline.

Lastly, a new line of work Pollux [27] aims to tune the batch size and learning rate of a model, allocating resources considering cluster-wide performance and fairness. While this system does leverage the ability to provision and deprovision resources in order to optimize for job performance, Pollux does not claim to be a general hyperparameter tuning framework as it focuses solely on optimizing the number of GPU's, batch size, and learning rate.

3 METHOD

In order to address the above issues and take advantage of the elastic setting, we develop SEER (Sequential Elimination with Elastic Resources). SEER is built on the same intuitions as Hyperband, and manages the exploration and exploitation trade-offs by varying the amounts of parallelism; we will explain these connections in depth at the end of this section.

3.1 Overview

The goal of SEER is to find a *final* model with high accuracy. This requires managing the given resources to search for good hyperparameter configurations, and then training them long enough to ensure that the final model is trained to convergence. To achieve this, SEER leverages the intuition that models with different hyperparameters converge at different rates and moreover can converge to different final accuracies (see Fig 3). Therefore, if a trial (a model with a certain set of hyperparameters) converges quickly, many trials should be launched and eliminated after few evaluations, and if the trial converges slowly, few trials should be launched and run for more evaluations. As we make no assumptions about the convergence rate beforehand, SEER generates *multiple brackets*, where each bracket has a *different number trials* and a *different number of resources per trials* (Fig 1c). This allows for both cases to be taken care of: if a trial converges quickly, then the low-performing trials can be quickly eliminated and the high performing trials can be moved to a bracket with more resources per trial. If a trial converges slowly, then the brackets with more resources will train fewer trials for longer, giving them the time to converge before they are eliminated.

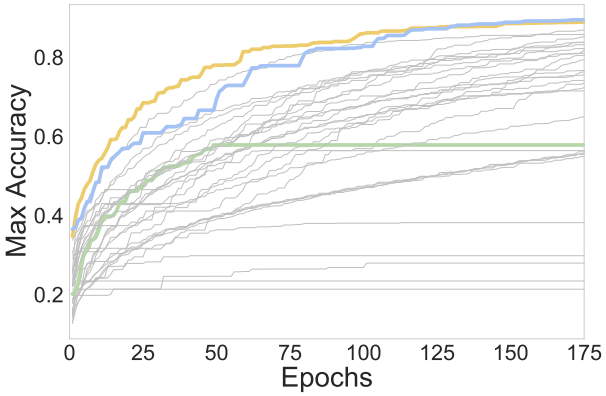


Figure 3: Accuracy curves of running grid search using ResNet18 trained on CIFAR10. Each line represents a model with unique hyperparameters. It is clear that different hyperparameters converge to different accuracies, and some hyperparameters (green line) converge very quickly while other hyperparameters (yellow and blue line) take longer to converge. Thus, SEER reasons that as evaluations come in, more resources should be dedicated to the yellow and blue lines while less resources should be dedicated to the green line.

The SEER algorithm is outlined in Algorithm 1. Boldface upper case letters denote lists (indexing starting at 1). SEER takes in 7 arguments, although only the first 2 are required: a search deadline T (units in minutes), budget B (units in resource-minutes), and optional parameters $p_{\max}, p_{\min}, t_{\min}, \eta$, and ν which we will discuss further in Sec 3.2. There are no restrictions on the amount of resources we can acquire at a given time, but we should pay for the duration we use a resource. Furthermore, resources are integral (e.g. fractional GPUs cannot be assigned to a task), but time and cost are continuous.

At a high level, SEER executes multiple brackets, where it is instructive to think of each bracket as one instantiation of SH. It is important to note that the start and end time of each elimination round (stage) of SH is the same among all brackets. Thus, each bracket finishes in time T, but allocates a different number of resources per trial resulting in varying degrees of parallelism.

To formalize SEER's use of brackets and successive halving, let $\mathcal{X}$ denote the space of hyperparameters. These can be continuous, discrete, ordinal, or a combination of the above. We assume that we can draw random samples from $\mathcal{X}$. Each bracket assigns an amount of resources p to each of the trials

Algorithm 1 SEER

Require: deadline T, cost budget B, $\eta, \nu, p_{\min}, p_{\max}, t_{\min}$ (defaults $\eta = 4, \nu = 2, p_{\min} = 1, p_{\max} = \infty, t_{\min} = 1$).

$K, t_1, \mathbf{N}, \mathbf{P} \leftarrow$ COMPUTE-BRACKET-PARAMETERS$(T, B, \eta, p_{\max}, t_{\min})$.
$S_1 \leftarrow$ randomly sample $\sum_{i=1}^{\text{len}(\mathbf{P})} \mathbf{N}[i]$ trials from $\mathcal{X}$. Order them arbitrarily.
for $k = 1, 2, \ldots, K$ **do**
  $s = 1$ # Starting index for current bracket.
  for $i = 1, 2, \ldots, \text{len}(\mathbf{P})$ **do** concurrently in parallel # Each round here takes time $t_1\eta^{k-1}$.
    $S_k^i \leftarrow$ the s^{th} to $(s + \lfloor \mathbf{N}[i]/\eta^{k-1} \rfloor)^{\text{th}}$ trials in S_k. # Trials in the i^{th} bracket at round k.
    $s \leftarrow \lfloor \mathbf{N}[i]/\eta^{k-1} \rfloor$
    Train all trials in S_k^i concurrently in parallel for time $t_1\eta^{k-1}$ with $\mathbf{P}[i]$ resources each, then evaluate.
    Remove the lowest $\lfloor \mathbf{N}[i]/\eta^k \rfloor$ performing trials of S_k^i from S_k # SH elimination.
  end for # Better trials to be moved to a bracket with more resources.
  $S_{k+1} \leftarrow$ Order the trials in S_k in descending order of loss.
end for
return The last element in S_{K+1}.

Algorithm 2 Compute number of SH rounds (K), time (t_1), number of trials ($\mathbf{N}$), and number of resources per trial ($\mathbf{P}$)

procedure COMPUTE-BRACKET-PARAMETERS$(T, B, \eta, \nu, p_{\min}, p_{\max}, t_{\min})$
  Let $R_\star$ be the largest $R > 0$ which satisfies the following two conditions,
    $\frac{R\eta}{(\eta-1)}\left(1 - \eta^{-\lceil \log_\eta R \rceil}\right) \le \frac{T}{t_{\min}}, \qquad p_{\min} R \lceil \log_\eta R \rceil \le \frac{B}{t_{\min}}.$ # This can be done using binary search
  $K \leftarrow \lceil \log_\eta R_\star \rceil$ # Number of SHA rounds per bracket
  $t_1 \leftarrow t_{\min} R_\star \eta^{-(K-1)}$ # Duration of the first round
  $B_0 \leftarrow p_{\min} t_{\min} R_\star \lceil \log_\eta R_\star \rceil$. # Budget of the first round
  $\mathbf{B}, \mathbf{P} \leftarrow$ COMPUTE-BRACKET-BUDGETS$(B, B_0, \nu, p_{\min}, p_{\max})$. # Budget, resources-per-trial per bracket.
  for $i = 1, 2, \ldots, \text{len}(\mathbf{P})$ **do**
    $\mathbf{N}[i] \leftarrow \lfloor \frac{\mathbf{B}[i]}{K t_1 \mathbf{P}[i]} \rfloor$. # Number of trials launched in the i^{th} bracket.
  end for
return $K, t_1, \mathbf{N}, \mathbf{P}$
end procedure

Algorithm 3 Compute bracket budget ($\mathbf{B}$) and bracket parallelism ($\mathbf{P}$)

procedure COMPUTE-BRACKET-BUDGETS$(B, B_0, \nu, p_{\min}, p_{\max})$
  Let $q_\star$ be the largest positive integer $q > 0$ satisfying $q\nu^{q-1} \le B/B_0$. # $q_\star$ = number of brackets
  if $p_{\min}\nu^{q_\star - 1} < p_{\max}$ **then** # The max number of resources-per-trial is less than $p_{\max}$
    $\mathbf{P} = [p_{\min}, p_{\min}\nu, p_{\min}\nu^2, \ldots, p_{\min}\nu^{q_\star - 1}, \min(p_{\max}, p_{\min}\nu^{q_\star})]$.
    $\mathbf{B}[i] \leftarrow B_0 \nu^{q_\star - 1}$ for $i \le q_\star, \qquad \mathbf{B}[q_\star + 1] \leftarrow (B - B_0 q_\star \nu^{q_\star - 1})$
  else # The max number of resources-per-trial is more than $p_{\max}$, so remove brackets and add trials
    Let $q'_\star$ be the largest integer $q \ge 0$ satisfying $p_{\min}\nu^q < p_{\max}$. # $q'_\star$ = neq number of brackets
    $\mathbf{P} \leftarrow [p_{\min}, p_{\min}\nu, p_{\min}\nu^2, \ldots, p_{\min}\nu^{q'_\star}, p_{\max}], \qquad \mathbf{B} \leftarrow (B/\text{len}(\mathbf{P})) * \mathbf{1}_{\text{len}(\mathbf{P})}$.
  end if
return $\mathbf{B}, \mathbf{P}$
end procedure

in that bracket for some time t, at which point they will be evaluated and a subset of them will be eliminated before moving to the next stage of successive halving. In the case of a typical machine learning setting, when p resources are assigned to a hyperparameter $x \in \mathcal{X}$ for time t, a trial initiated with hyperparameters x is trained for t timesteps using p resources, at which point it is evaluated (e.g. evaluate the trial on the validation set). For our experiments, we interpret training a trial with p resources as training this model in a

data-parallel fashion over p GPU's, but one could instead use pipeline-parallelism or a different parallelization strategy. Similarly, while our experiments use validation accuracy as our evaluation metric, one could pick any metric they want to minimize or maximize (e.g. minimizing loss).

The parameters for each bracket is computed using the COMPUTE-BRACKET-PARAMETERS (Algorithm 2) subroutine which returns $K, t_1, \mathbf{N}, \mathbf{P}$. The algorithm proceeds in K stages and trials can be transferred from one bracket to another at the end of a stage depending on how well it performs. For this, it maintains an ordered set of trials in S_k, for $k = 1, \ldots, K+1$. First, S_1 is sampled randomly from $\mathcal{X}$ and ordered arbitrarily. In the k^{th} stage, we evaluate the trials in S_k in different brackets. At the end of the stage, we order the trials according to their loss values to produce S_{k+1}. In order to prioritize the promising trials, in the next stage, the number of trials evaluated per bracket is reduced to $1/\eta$ of its value in the previous stage, and moreover trials that yields a lower loss value are assigned to brackets with higher parallelism (line 3). This also ensures that only $(1/\eta)$ fraction of the trials are carried forward from one stage to another.

Since trials are trained for more epochs in brackets with more resources, a natural question that arises is whether the trials in the bracket with less resources will ever outperform trials in the bracket with more resources. While the trials in the highest bracket will get more evaluations, since we sample more trials in lower brackets (see Fig1(c)), we are more likely to sample a good trial which can quickly yield high accuracy. Such trials will be moved to a higher bracket in the next stage. In our experiments, we found that the optimal trial was equally likely to have been initially sampled from low or high brackets.

3.2 Optional Arguments

Along with the required inputs of deadline T and budget B, SEER takes in 5 other optional parameters $\eta, p_{\max}, p_{\min}, t_{\min}$, and ν which can be set if the user has additional knowledge of their workload. Unless otherwise stated, we use the default parameters in our experiments.

The parameter η (> 1) is a standard parameter in successive halving algorithms that dictate how aggressively we eliminate poorly performing parameters and ν (≥ 1) dictates how aggressively we increase the number of parallel resources assigned to a trial. When η is large, we eliminate parameters more aggressively, whereas when ν is small we increase the degree of parallelism aggressively.

Next, $p_{\min}$ and $p_{\max}$ are hard constraints on the minimum and maximum number of resources that can be assigned to a configuration. A case in which a user may want to set $p_{min} > 1$ would be if training a trial for one epoch (aka one evaluation) on one machine takes a very long time, so the user would set p_{min} higher to shorten the experiment time. On the flip side, a user may want to set $p_{max} < \infty$ if they knew that a trial's throughput does not increase when given more than a certain number of machines [13], so allocating any more would simply be wasting resource-time.

Lastly, t_{min} specifies the minimum training time before we choose to eliminate a configuration. In practice, this should be set to roughly how long it takes to train a trial for 1-2 epochs, unless the user has more information as to the convergence rate of their trials (if all trials are slower to converge, this may want to be set higher). We do note that often it is not known how long it takes to train a trial for 1-2 epochs, but this can be easily profiled and we have observed in our experiments that this parameter can be set slightly more or less than the actual training time with minimal effects on the results.

3.3 Computing Bracket Parameters

The computation of the bracket parameters $K, t_1, \mathbf{N}, \mathbf{P}$ as outlined in Algorithm 2 leverages some of the intuitions regarding non-ideal scaling alluded to in Section 1. To illustrate this, assume, for now, that the deadline T is small, but the budget B is large; more concretely, when computing $R_\star$ in line 2, the first inequality is tight, but $p_{\min} R_\star \lceil \log_\eta R_\star \rceil < M$. Loosely speaking, in this case, the algorithm has a large budget, and it must decide if it should train a small number of trials (N) using more resources per configuration (p) or if it should train a large number of trials using fewer resources per trial. If the problem is such that training each trial can take a long time to converge, then we should prefer the former. On the other hand, if convergence is fast, then we should prefer the latter. In the absence of any prior knowledge, there is no way to say *a priori* what the optimal trade-off would be. Therefore SEER hedges its bets by dividing up its total cost budget and executing different brackets with different (N, p) values.

When determining the appropriate p values as stated above, it starts with small values of p. This accounts for the fact that, like most parallel systems, model training exhibits diminishing returns when more resources are used in parallel. By starting with small p and stretching out the cost for as long as possible, we ensure that the cost was spent as efficiently as possible. However, parallelism can speed up training, and hence SEER increases the amount of parallelism with the cost budget B. Specifically, we start new brackets with successively higher amount of parallelism when B increases to certain thresholds dependent on ν.

Finally, suppose that the specified budget is "too small" for the given deadline; i.e. in line 2, the second inequality is tight, but $\frac{R\eta}{(\eta-1)} \left(1 - \eta^{-\lceil \log_\eta R \rceil}\right) < T$. Then, the algorithm creates a single bracket with $\mathbf{P}[1] = p_{\min}$ and $\mathbf{N}[1] = B/(K t_1 p_{\min})$, which exhausts the budget and finishes sooner than the

deadline. Alternatively, had we attempted to 'stretch out' the budget to fill up the deadline, it could result in worse performance, as we may have to reduce the number of trial sampled to stay within the budget. In the previous case, when the inequality for T was tight, increasing the budget could indeed result in better performance since the additional budget is used to both increase the amount of resources spent on existing brackets and create new brackets with more resources per-trial. However, in this case, the budget creates a bottleneck and increasing the deadline further does not alter the behavior or output of the algorithm.

3.4 Example

To better illustrate this method, let's consider the plan produced by SEER for a $T = 10$ minute experiment with a budget of $T = 80$ GPU-minutes and $\eta = 2$ as shown in Fig 1c.

Here, we obtain $R_* = 5.714, K = 3, t_1 = 1.42$, and $B_0 = 17.142$, meaning that there will be 3 elimination rounds, with the first round taking 1.42 minutes and requiring 17.142 GPU-minutes. Then, using Algorithm 3 we obtain the number of brackets $q_* = 2$, the total budget per bracket ($\mathbf{B} = [17.142, 34.283, 11.432]$), and the parallelism of each bracket ($\mathbf{P} = [1, 2]$).

From here we can Algorithm 2 to obtain the number of trials initialized for each bracket $\mathbf{N} = [8, 4]$. Now we have everything we need to run SEER: first, 12 hyperparameter configurations will be randomly selected from a predefined search space. For round 1 of SH (from $t = 0$ to $t = t_{min} = 1.42$ minutes), 8 trials will be trained with 1 GPU each in bracket 1 and the remaining 4 trials will be trained with 2 GPU's in bracket 2. At the end of round 1, the 4 lowest performing trials in bracket 1 will be eliminated and the 2 lowest performing trials from bracket 2 will be eliminated. At the start of round 2, the top 2 performing trials will be assigned to bracket 2 and the rest will be assigned to bracket 1. From time $t = t_{min} = 1.42$ to $t = t_{min}\eta^{k-1} = 2.84$ minutes, 4 trials will be trained with 1 GPU and 2 trials will be trained with 2 GPU's. After elimination, the highest performing trial at the beginning of round 3 will be trained with 2 GPU's and the 2nd and 3rd highest performing trials will be trained with 1 GPU from time $t = 4.26$ to $t = 9.94$ minutes. Lastly, the highest performing trial after round 3 will be returned by SEER. This is visually represented by Fig 1c.

3.5 Comparison with Prior work

SEER uses similar intuitions to SH, Hyperband, ASHA, and Hypersched. However, SEER is catered to our elastic and deadline-aware setting. For instance, SEER's strategy of running multiple brackets with different amounts of parallelism is similar in spirit to Hyperband's stratgy of running multiple SH instances where each hyperparameter configuration is trained for different lengths of time (or iterations). However, while the latter strategy works for any time algorithms, it may not be suitable when there is a deadline. Additionally, in SEER, trials are transferred from one bracket to another depending on how well they perform, whereas in Hyperband, each SH instance is run independently. We design a naive elastic variant of Hyperband and show that it performs worse than SEER.

Furthermore, the generation of a hyperparameter tuning job is an important distinction from Rubberband [24], which also takes into account time and cost budgets. Unlike the other previous works, Rubberband assumes that the user already has an experiment they want to run, say a successive halving experiment, and the way in which compute resources are allocated to execute said plan will not affect the final accuracy. Thus, Rubberband does not aim to maximize accuracy but rather to minimize cost of a job with a set accuracy. In contrast, our work aims to replace the profiling and simulation step of Rubberband that converts a fixed cluster policy to the elastic setting, and run the job generated by SEER on Rubberband's executor.

4 THEORETICAL ANALYSIS

We now present our theoretical analysis. Our proofs are given in Appendix A, with some technical lemmas skipped due to space constraints. Our first result simply verifies that SEER does not exceed the time and cost budgets.

Fact 1. *Algorithm 3 completes in time at most T and expends resource-time at most B.*

Our main theoretical results bound the difference between the configuration returned by SEER and the optimal configuration in $\mathcal{X}$. Intuitively, we show that SEER does well using varying levels of parallelism. For this, we will begin with some assumptions on the problem.

Let $\ell : \mathcal{X} \times \mathbb{R}_+ \to [0, 1]$ denote the loss functions defined over $\mathcal{X}$, where, for example, $\ell(x, t)$ denotes the validation loss when we train the model with hyperparameter $x \in \mathcal{X}$ for time $t \in \mathbb{R}_+$ using a *single* resource. Let $\ell^\infty : \mathcal{X} \to [0, 1]$ denote the terminal losses over $\mathcal{X}$. That is, $\ell^\infty(x) = \lim_{t\to\infty} \ell(x, t)$ is the final loss when we train the model to completion with hyperparameter x. We will assume that this limit exists for all $x \in \mathcal{X}$, and therefore ℓ^∞ is well defined. Let $\ell^\star = \inf_{x\in\mathcal{X}} \ell^\infty(x)$ be the optimal loss value in the given space $\mathcal{X}$. Next, we will define $\gamma : \mathbb{R}_+ \to \mathbb{R}_+$ to be the pointwise smallest monotonically decreasing function satisfying $\sup_{x\in\mathcal{X}} |\ell(x, t) - \ell^\infty(x)| \le \gamma(t)$. Intuitively, if γ is small it means all configurations in $\mathcal{X}$ converge to their terminal values fast. We will denote the inverse of γ by $\gamma^{-1}(y) = \inf\{t \in \mathbb{R}_+; \gamma(t) \le y\}$. Note that both γ and γ^{-1} are decreasing functions.

Next, recall that each instance of SH in SEER randomly samples initial hyperparameter configurations from some distribution P, with $\textbf{supp}\,(P) = \mathcal{X}$. Let F denote the induced CDF of terminal losses when configurations are sampled from P, i.e. $F(y) = \mathbb{P}_{X\sim P}(\ell^{\infty}(X) \leq y)$. Clearly, $\textbf{supp}\,(F) \subset [\ell^{\star}, 1)$. We will denote the inverse of F by $F^{-1}(z) = \inf\{y : F(y) \leq z\}$. The assumptions above are consistent with similar assumptions on this problem in previous work (e.g. [14, 20]), except we state all our quantities in terms of (continuous) time, instead of epochs.

However, in a departure from prior work, we will model the scaling effects of using multiple resources for evaluating a single model via the function $\lambda : \mathbb{N}_{+} \to \mathbb{R}$, which has the following interpretation: the loss after training $x \in \mathcal{X}$ for time t is $\ell(x, \lambda(p)t)$. Clearly $\lambda(1) = 1$ as per our definition of ℓ above. λ is an increasing function, which captures the fact that more resources can help you train faster. The sublinear scaling of using multiple resources for the same job can be modeled via the assumption that $\lambda(p)/p$ is decreasing with p; i.e. the per resource efficiency decreases with more resources. For instance, this implies that $\lambda(p) \leq p$, meaning that training for time t with p resources is worse than training for time pt with a single resource.

Observe that many of the above assumptions are stated in abstract terms. Moreover, the algorithm is agnostic to quantities such as the loss functions ℓ, the convergence rate γ, and the scaling characteristics λ; this is by design—in practice, it is usually not possible to know them ahead of time for an arbitrary hyperparameter tuning job. Therefore, while sequential elimination strategies such as SH are known to be optimal for stochastic best arm identification [4], our goal here is less ambitious. We simply wish to *(i)* demonstrate that the algorithm will behave reasonably under suitable assumptions on the problem, and *(ii)* understand the effects of the deadline, budget, and parallelism in the elastic setting.

To simplify the exposition we will assume that $p_{\min} = 1$ and $p_{\max} = \infty$. Suppose that the configuration returned by the algorithm is $\widehat{x}$ and that this was trained using p resources. Denote the final loss after training for time T by $\widehat{\ell} = \ell(\widehat{x}, T\lambda(p))$. We are interested in bounding the error $\widehat{\ell} - \ell^{\star}$. We will state our theorem in the case where $T = \frac{R\eta}{(\eta-1)}\left(1 - \eta^{-\lceil \log_{\eta} R\rceil}\right)$. That is, the deadline constraint for $R_{\star}$ in line 2 is tight. Let $q_{\star}$ be as defined in line 2. Under this, we see that $R_{\star}$ is an increasing function of T and $q_{\star}$ is an increasing function of B. Our main result, stated in terms of $R_{\star}$ and $q_{\star}$ demonstrates that as both T and B increases, the loss of the returned model is close to the optimal value $\ell^{\star}$ with high probability.

Theorem 2. *Consider an execution of Algorithm 3 with parameters η and ν. Moreover, let $R_{\star}$ and $q_{\star}$ be as defined in lines 2 and 2. For $\delta > 0$, define $u_{\star}(\delta) = \log(2q_{\star}/\delta)/R_{\star}$. Let $\delta \in (0, 1)$ be such that the following holds,*

$$\min_{j \leq q_{\star}} \left(\frac{2\eta}{\lambda(\nu^{q_{\star}-j})} \int_{u_{\star}(\delta)}^{1} \gamma^{-1}\left(\frac{F^{-1}(t) - \ell^{\star}}{4}\right) \mathrm{d}t + \frac{20\eta^2}{3R_{\star}\nu^{q_{\star}-j}\lambda(\nu^{j})} \log\left(\frac{2q_{\star}}{\delta}\right) \gamma^{-1}\left(\frac{F^{-1}(u_{\star}(\delta)) - \ell^{\star}}{4}\right) \right) \leq 1. \tag{1}$$

Let $\widehat{\ell}$ be the trained loss of the best model returned by the algorithm. Then with probability at least $1-\delta$, we have $\widehat{\ell} - \ell^{\star} \leq 4(F^{-1}(u_{\star}(\delta)) - \ell^{\star})$.

We first observe that there always exists some $\delta' \in (0, 1)$ such that for all $\delta > \delta'$, the given condition in the theorem is satisfied, However δ' may be large if, intuitively speaking, either F is a heavy-tailed distribution (i.e. an optimum is hard to find via randomly sampling from the space) or γ converges slowly to 0 (i.e. the model class converges slowly). In contrast, when F and γ have smaller tails, then we can also find small δ such that the condition is true, and consequently, we will have a small bound on the error $4(F^{-1}(u_{\star}(\delta) - \ell^{\star})$.

To illustrate this, observe that as $u_{\star}$ has just log dependence on $q*/\delta$. Moreover, as it decreases with $R_{\star}$, it also does so with T. Therefore, the bound $4(F^{-1}(u_{\star}(\delta) - \ell^{\star})$ decreases with the deadline T. Next, fix the deadline T, and assume that we increase B. Then, $q_{\star}$ increases and therefore both terms in the LHS of (1) decrease; this follows from the fact that λ is an increasing function. Hence, for a given $j \leq q_{\star}$, the expression in the LHS becomes smaller with large B, which means it is likely to be smaller than 1 for small δ values. This improves the probability that the final bound will hold. Additionally, when $q_{\star}$ increases there will be more terms to account for in the minimisation, which increases the chances that one of them will be smaller than 1. This reflects the fact that when the budget increases, it is used to both, increase the number of configurations in existing SH instantiations and create more instantiations with higher parallelism.

On the flip side, when $R_{\star}$ is large, $u_{\star}$ is small which increases the value of both terms in the LHS. However, provided that γ is decaying fast enough, the effect of these terms can be negligible–see Section 5.3.2 in Li et al. [20]. Moreover, while large $q_{\star}$ does affect the expression (1) and the final bound negatively, it only does so by log factors.

Next, let us turn to the effects of parallelism on the result, for which we will fix $q_{\star}$ and $R_{\star}$. Since λ is an increasing function of the number of resources, the first term in the LHS of (1) decreases with a large number of resources. However, in the second term, since $\lambda(p)/p$ is decreasing due to sublinear scaling, the coefficient $1/(\nu^{q_{\star}-j}\lambda(\nu^{j}))$ is small for instances j with fewer resources. The optimal j will depend on γ and F^{-1}. This captures the fact that the optimal level of parallelism depends on the problem: if training converges slowly, it might be better to sample few trials and train them for long, and vice versa if training converges fast.

It must be stated that an algorithm which accounts for many of the problem specific quantities such as γ, F, and λ will able to achieve better guarantees and better quantify the dependence of the error $\widehat{\ell} - \ell^\star$ in terms of these quantities. That said, it is worth noting that even though SEER is agnostic to these quantities, the above theorem demonstrates that it behaves sensibly: as B and T increases so do the chances of the returned model being close to the optimal.

5 SYSTEM DESIGN & IMPLEMENTATION

As stated previously, we employ Rubberband's executor to handle the placement of trials and scaling of the cluster. The executor contains three main components: the trial scheduler, placement controller, and cluster manager. Each component leverages Tune [23], an open source framework for distributed hyperparameter tuning and model training, as well as the distributed framework Ray [25], which is able to launch training jobs in parallel as well as scale a cluster up or down.

If the current cluster needs to change or resources need to be reallocated (e.g. at the end of a stage), the scheduler requests the cluster manager to provision new nodes or deprovision existing ones. To reallocate workers, the placement controller will convert the resource quantity allocated to each trial into physical resource assignments for its workers. Parallel workers of a trial should be either colocated on a single machine or packed onto a minimal set of nodes. By colocating workers, the distributed training algorithm will avoid incurring unnecessary network overheads.

We modify Rubberband's existing scheduler to take in plans generated by SEER (along with all baselines in Section 6) and use Rubberband's cluster manager and placement controller for execution.

Rubberband's current scheduler takes in an *allocation policy*, which defines the hyperparameter tuning job to execute. This allocation policy is comprised of *stages*, which specify how many trials to run and how many epochs/evaluations to run them for. This allocation policy is then used by the simulator to determine the allocation of resources to each trial within a stage, with all resources being shared equally. In contrast, SEER needs an allocation policy which already specifies how to allocate resources and which allows trials within each stage to be given a different number of resources. Thus our allocation plan is a series of *skewed stages*, each with start and stop time, a list of trials, and their respective number of resources. At the end of a stage k, the top $1/\eta$ fraction of trials are chosen to continue onto stage $k+1$. As indicated in Algorithm 3, the best performing trials get the most resources. When proceeding from one stage to the next, the configurations that do not progress to the next stage are eliminated. The cluster manager will determine which (if any) instances need to be terminated and the placement controller will re-allocate the remaining resources to the surviving trials such that worker colocation is maximized.

Finally, we mention that while prior systems for hyperparameter tuning [21, 23] use the number of epochs to define a stage in their design, we use wall clock time because instances charge by the amount of time used, and estimating the time it takes for a single epoch can be difficult, especially in the case of stragglers. While our approach avoids wasting time waiting for stragglers, it is also possible that the time for a stage may run out before any configurations has finished a single epoch. In this case, the user can specify a minimum amount of time per stage via the parameter $t_{\min}$ to ensure all configurations have finished at least one epoch before they are eliminated or promoted.

6 EXPERIMENTS

We present our experimental evaluation in this Section.

6.1 Setup

All experiments are run on AWS p3.8xlarge instances, each of which provides 4 NVIDIA Tesla V100 GPUs. We utilize a single r5.8xlarge instance to coordinate experiments and host model checkpoints. In practice, the price of the CPU instance is negligible in comparison to that of GPU instances, and is therefore ignored for the purposes of this evaluation. To ensure a fair comparison across all benchmarks, we also provide results only for experiments where there were no node failures.

Unless otherwise stated, we use a fixed batch size of 2048 throughout the entire experiment. Since scaling up the batch size with the number of GPUs has been shown to have unpredictable performance, we set a large batch size and use *gradient accumulation* to ensure that the batch size does not change during learning.

6.2 Search Spaces

Our search space is taken from previous works [20–22]. Tab. 1 depicts the search space for standard image classification benchmarks used for the image classification benchmarks in Sec. 6. For the segmentation benchmarks, the search space is the same except all the learning rate parameters are divided by 10. For the text classification task, all the learning rate parameters are multiplied by 3 and we add in a parameter for the embedding size with choices of 8, 16, 32, 64, and 128.

6.3 Baselines

We do not compare against Rubberband's simulator as Rubberband does not have the objective of maximizing accuracy, and instead convert some fixed-cluster algorithms into elastic algorithms to fairly compare how to SEER. We compare SEER

Hyperparameter	Space
Learning Rate	1e-4, 5e-4, 1e-3, 5e-3, 0.01, 0.05, 0.1, 0.5, 1
Weight Decay	0.0001, 0.0005, 0.001, 0.005
Momentum	0.9, 0.95, 0.99, 0.997

Table 1: Standard search space for vision models

to 4 fixed-cluster algorithms: Random, ASHA, HyperSched, and BOHB; as well as 2 elastic algorithms: Elastic Hyperband (E-Hyperband) and Elastic grid searc (E-Grid Search).
Random: This chooses a random configuration in the search space and trains it for the entire duration using a fixed cluster size so that the resource-time is equal to the given budget. This is a natural baseline which indicates whether or not hyperparameter tuning is necessary for a problem.

ASHA, HyperSched, BOHB: These are methods from prior work which are based on successive halving. We use the same parameter η for these methods as we did for SEER and E-Hyperband. They also require specifying a minimum and maximum resource allocation (which in this case is number of epochs) as specified in Section 3. We set them on a per-experiment basis using the guidelines provided in the respective papers. Additionally, HyperSched requires specifying a scaling function which maps the number of resources to throughput. For the vision benchmarks, we use the functions given in their paper, and for the NLP and segmentation tasks, we use the default in their implementation.

It is worth mentioning, that the above methods cannot be naturally executed in an elastic environment as their policies explicitly depend on the number of workers available.

E-Hyperband: This adapts Hyperband–which is traditionally run sequentially–to the elastic setting by running all brackets in parallel, and calculating its input to produce a plan that satisfies the time and monetary constraints provided. Hyperband defines its brackets in terms of R, the maximum amount of resources to give to a single configuration. Since the concept of a resource is generic, we could define R as the largest r which is less than the deadline but also so that the cost is less than the given budget.

E-Grid Search: This is a simple search technique with a set exploration and exploitation phase: given a deadline T (minutes), budget B (GPU-minutes), and max/min parallelism $p_{\min}/p_{\max}$, $T/2$ minutes are allocated for exploitation of the top trial using $p_{\max}$ resources, and $T/2$ minutes are allocated for exploration, with each trial getting $p_{\min}$ resources and the number of trials chosen based on B. Unless otherwise stated, we set $p_{\min} = 1, p_{\max} = 4$ for both the elastic grid search baseline and SEER.

6.4 Benchmarks

We evaluate the above methods on five datasets on three tasks, image classification, image segmentation, and text classification.

Image Classification: We evaluate 3 different image classification models/datasets: VGG16 [29] on SVHN [26], ResNet18 [11] on CIFAR10 [18], and (3) ResNet50 [11] on TinyImagenet [19]. For our TinyImagenet experiments, we begin with a model pretrained on ImageNet. We use an SGD optmizer and a plateau learning rate scheduler, where the learning rate halves after the model has trained for 5 epochs without any accuracy increase.

Image Segmentation: We evaluate FCN ResNet50 [28] on the PASCAL VOC2012 [6] segmentation dataset. Since this is a fairly large dataset, for E-Hyperband, E-Grid Seach, and SEER, we set $p_{\max} = 8$ and change $t_{\min}$ to 5 minutes. We use an SGD optimizer with a ploy learning rate scheduler as described above, and a batch size of 4.

Text Classification We evaluate BERT [5] on the Muli-Genre Natural Language Inferene corpus (MNLI) dataset from the General Language Understanding Evaluation (GLUE) benchmark [31]. Given a premise sentence and a hypothesis sentence, the task is to predict whether the premise entails the hypothesis (entail- ment), contradicts the hypothesis (contradiction), or neither (neutral). We evaluate on both the matched (in-domain) and mismatched (cross-domain) sections. We use a model pretrained on lowercase english text from the Huggingface repository [32]. For Hyperband, E-Grid Seach, and SEER, we set $p_{max} = 8$ and changed t_{min} from 1 minute to 10 minutes.

Results. As shown in Tables 2, 4, and 3, SEER outperforms the other methods on all benchmarks. One interesting observation is that the naive E-Grid Search baseline often outperforms other sophisticated fixed-cluster baselines. As mentioned before, this suggests that the effects of non-ideal scaling can be significant in practice and highlights the benefits of cloud-specific algorithms. ASHA performs poorly since it spends too much time exploring configurations even close to the deadline, indicating that carefully designed resource allocation strategies can make a difference in deadline-aware settings.

6.5 Different Budgets and Deadlines

Figure 4 shows the performance of SEER and other methods when we vary the deadline for a fixed budget and when we vary the budget for a fixed deadline. SEER generally outperforms other methods, with the largest performance gains coming from the short deadline, high budget settings. This is because HyperSched is able to sufficiently explore trials in a larger deadline. Figure 4b shows that when the budget is fixed GPU minutes, SEER outperforms the baselines, but

Method	Model	Dataset	Deadline	GPU minutes	Accuracy	Std-Error
Random	VGG16	SVHN	15	4×15	0.19	0.028
ASHA	VGG16	SVHN	15	4×15	0.819	0.053
HyperSched	VGG16	SVHN	15	4×15	0.927	0.021
BOHB	VGG16	SVHN	15	4×15	0.458	0.086
E-Hyperband	VGG16	SVHN	15	4×15	0.921	0.015
E-Grid Search	VGG16	SVHN	15	4×15	0.944	0.010
SEER	VGG16	SVHN	15	4×15	**0.956**	**0.005**
Random	ResNet18	CIFAR10	60	16×60	0.226	0.101
ASHA	ResNet18	CIFAR10	60	16×60	0.896	0.006
HyperSched	ResNet18	CIFAR10	60	16×60	0.932	0.005
BOHB	ResNet18	CIFAR10	60	16×60	0.864	0.000
E-Hyperband	ResNet18	CIFAR10	60	16×60	0.914	0.005
E-Grid Search	ResNet18	CIFAR10	60	16×60	0.904	0.001
SEER	ResNet18	CIFAR10	60	16×60	**0.935**	**0.001**
Random	ResNet50	TinyImagenet	60	16×60	0.091	0.064
ASHA	ResNet50	TinyImagenet	60	16×60	0.212	0.068
HyperSched	ResNet50	TinyImagenet	60	16×60	0.581	0.019
BOHB	ResNet50	TinyImagenet	60	16×60	0.110	0.055
E-Hyperband	ResNet50	TinyImagenet	60	16×60	0.630	0.003
E-Grid Search	ResNet50	TinyImagenet	60	16×60	0.632	0.049
SEER	ResNet50	TinyImagenet	60	16×60	**0.675**	**0.001**

Table 2: Results from training various models and datasets for image classification. Accuracy is averaged across 3 runs. We have separated the elastic and inelastic methods with a dashed line. If the deadline is t and the budget is $n \times t$ GPU-minutes, it means an inelastic algorithm would have used n resources for the entire duration of t minutes.

Method	M/MM Accuracy	Std-Error
Random	0.651 / 0.657	0.078 / 0.098
ASHA	0.837 / 0.831	0.002 / 0.001
HyperSched	0.834 / 0.837	0.001 / 0.001
BOHB	0.814 / 0.817	0.001 / 0.000
E-Hyperband	0.836 / 0.831	0.002 / 0.001
E-Grid Search	0.833 / 0.815	0.001 / 0.002
SEER	**0.839 / 0.840**	**0.001 / 0.002**

Table 3: Results from fine-tuning BERT on MNLI dataset with a deadline of 120 minutes and a budget of 16×120 GPU-minutes. Accuracy is recorded for both the matched and mismatched sections.

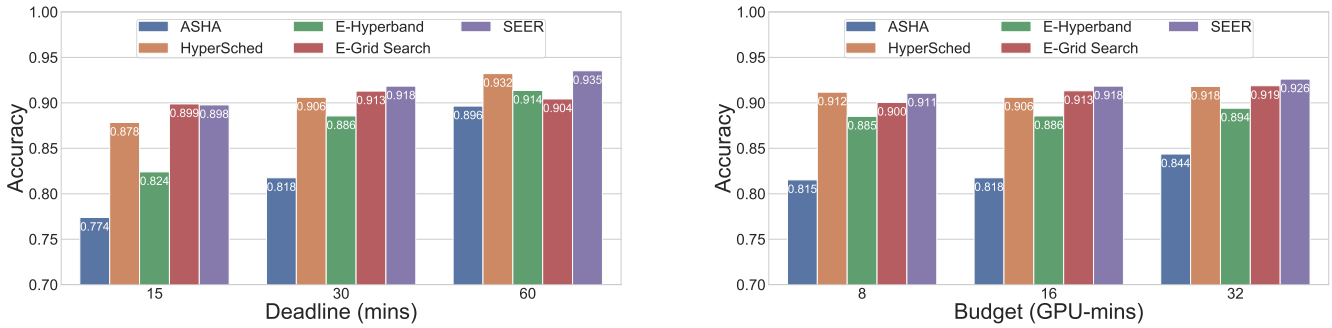
Method	Mean-IOU	Std-Error
Random	0.413	0.078
ASHA	0.519	0.005
HyperSched	0.524	0.061
BOHB	0.474	0.078
E-Hyperband	0.503	0.003
E-Grid Search	0.524	0.006
SEER	**0.541**	**0.008**

Table 4: Results from training FCN ResNet50 on the Pascal VOC2012 segmentation dataset with a deadline of 180 minutes and a budget of 16×180 GPU-minutes.

HyperSched outperforms SEER and ASHA on the smallest budget 8×30, because SEER only evaluates 13 trials and is less likely to find an optimal configuration than HyperSched which evaluates more trials.

7 CONCLUSION

Limitations: In this section we list the potential limitations of our work. Similar to HyperSched with a similar problem formulation, ASHA will outperform SEER given a very large deadline since it tests more configurations and has enough time to train them to convergence. Furthermore, HyperSched will match or outperform SEER given a small budget, because SEER tests much fewer configurations and the wasted resources due to sub-linear model scaling is minimized. As mentioned in Sec. 6.4 and 6.5, SEER may also perform poorly when the the deadline is very small (~10-15 minutes) because of cluster resizing overheads.



(a) CIFAR10 Different Deadlines **(b) CIFAR10 Different Budgets**

Figure 4: Results of tuning ResNet18 on CIFAR10 with (a) a fixed budget of 16×deadline **GPU-minutes for deadlines of 15, 30, and 60 minutes and (b) a fixed deadline of 30 minutes and budgets of** 8×30, 16×30, **and** 32×30 **GPU-minutes. HyperSched outperforms SEER on a budget of** 8×30 **GPU-minutes because SEER evaluates very few trials and the effects of sublinear scaling are minimized.**

Summary: In this work, we formalize the problem of elastic hyperparameter tuning in terms of time and cost constraints to find a model with high accuracy. We introduce SEER, which produces a resource allocation plan to evaluate several hyperparameters, and train the promising one to completion based on the stipulated time and cost budgets. It leverages elasticity to sufficiently explore configurations while maintaining different levels of parallelism to identify the best trade-off for the problem. The proposed method outperforms fixed-cluster methods and naive elastic heuristics on a variety of deep learning benchmarks.

The cloud introduces new opportunities and challenges for machine learning. While the existing literature has focused on system design for the cloud, this work demonstrates that more algorithmic work is needed to fully realize the benefits of the cloud for emerging machine learning workloads.

A PROOFS OF THEORETICAL RESULTS

In order to prove Theorem 2, we will require two intermediate results. The first of these is a technical result taken from Li et al. [20] which shows that if we draw many samples from $\mathcal{X}$, the small terminal losses of the samples should be close to the optimal loss in $\mathcal{X}$. Its proof is similar to the above paper, and therefore we skip it due to space constraints.. Recall the definitions of P, F and γ from Section 4.

Lemma 3 (Adapted from Lemma 2, Li et al. [20]). *Let $\delta \in (0,1)$. Suppose we draw $N = MN'$ i.i.d. samples $x_1, \ldots, x_N$ from P such that, $M, N \geq 1$ are integers and $\ell^\infty(x_1) \leq \ell^\infty(x_2) \leq \cdots \leq \ell^\infty(x_N)$. Denote $u_{N'} = \frac{\log(2M/\delta)}{N'}$. Define,*

$$\mathrm{H}(N,\delta) = 2N \int_{u_{N'}}^{1} \gamma^{-1}\left(\frac{F^{-1}(t) - \ell^\star}{4}\right) \mathrm{d}t + \frac{10}{3}\log\left(\frac{2}{\delta}\right)\gamma^{-1}\left(\frac{F^{-1}(q_N) - \ell^\star}{4}\right).$$

Then, with probability at least $1-\delta$ $\ell^\infty(x_M) \leq F^{-1}(u_{N'})$, and

$$\sum_{i=1}^{N} \gamma^{-1}\left(\max\left\{F^{-1}(u_{N'}) - \ell^\star, \frac{\ell^\infty(x_i) - \ell^\infty(x_M)}{4}\right\}\right) \leq \mathrm{H}(N,\delta).$$

Our next result states that a SH procedure can achieve low loss if enough resource time is allocated to it.

Lemma 4. *Consider a set of N configurations $\{x_1, x_2, \ldots, x_N\}$ such that $\ell(x_1) \leq \ell(x_2) \leq \ldots \ell(x_N)$. Assume that we executed successive halving with parameter η for K ($\leq \log_\eta(N)$) stages using p workers for each configuration. Let $M = \lceil N/\eta^K \rceil$. Define B_{SH} as follows,*

$$B_{\mathrm{SH}}(\epsilon) = \frac{K\eta p}{\lambda(p)}\gamma^{-1}\left(\max\left\{\frac{\epsilon}{4}, \frac{\ell^\infty(x_i) - \ell^\infty(x_1)}{4}\right\}\right)$$

Let $\widehat{\ell}$ denote the smallest observed loss among all surviving configurations at the end of the K^{th} stage. If $B \geq B_{\mathrm{SH}}$, then $\widehat{\ell} \leq \ell_M + 3\epsilon/4$.

Proof. For brevity, denote $\ell^\infty(x_j) = \ell_j$ and $\ell(x_j, t) = \ell_{j,t}$. Denote the time taken for round k by t_k, and let $\bar{t}_k = \sum_{k' \leq k} t_{k'}$. Let the arms surviving at the end of round k by S_k. Let S_{K+1} denote the arms surviving after the K^{th} stage. Recall, that SH chooses the best arm at the end of the K^{th} stage.

First, consider the time taken for the k^{th} round,

$$t_k = \frac{B/K}{p|S_k|} \geq \frac{\eta}{|S_k|\lambda(p)} \sum_{j=1}^{N} \gamma^{-1}\left(\max\left\{\frac{\epsilon}{4}, \frac{\ell_j - \ell_M}{4}\right\}\right)$$
$$\geq \frac{\eta}{|S_k|\lambda(p)} \max_{i=1,\ldots,N} i\gamma^{-1}\left(\max\left\{\frac{\epsilon}{4}, \frac{\ell_j - \ell^{\infty}(x_M)}{4}\right\}\right)$$
$$\geq \frac{1}{\lambda(p)}\gamma^{-1}\left(\max\left\{\frac{\epsilon}{4}, \frac{\ell_{\lceil N/\eta^k\rceil} - \ell_M}{4}\right\}\right) \quad (2)$$

First, assume that at least one of the first M arms, say x_m, survived after K stages. By (2), we have $\bar{t}_k \geq \gamma^{-1}(\epsilon/4)/\lambda(p)$, and therefore, $|\ell_{m,\lambda(p)\bar{t}_K} - \ell_m| \leq \gamma(\bar{t}_k\lambda(p)) \leq \epsilon/4$. For the model with the best loss we therefore have,

$$\widehat{\ell} - \ell_M \leq \ell_{m,\bar{t}_K} - \ell_m + \ell_m - \ell_M \leq \epsilon/4.$$

Here, we have used the fact that $\widehat{\ell} \leq \ell_{m,\bar{t}_K}$ and that $\ell_m \leq \ell_M$. This proves the result if one of the first M arms survives the K rounds. In the remainder of the proof we will prove the result assuming that none of them do.

First define $\tau_j = \frac{1}{\lambda(p)}\gamma^{-1}\left(\frac{\ell_j - \ell_M}{2}\right)$. If $t > \tau_j$, for any $i = 1, \ldots, N$ after time t of training with p workers, we have $|\ell_{i,t\lambda(p)} - \ell_i| \leq \gamma(t\lambda(p)) \leq (\ell_i - \ell_M)/2$. Applying this to M and j we have, $\ell_{j,t\lambda(p)} - \ell_{M,t\lambda(p)} = \ell_{j,t\lambda(p)} - \ell_j + \ell_j - \ell_M + \ell_M - \ell_{M,t\lambda(p)} = (\ell_{j,t\lambda(p)} - \ell_j) + (\ell_M - \ell_{M,t\lambda(p)}) + \ell_j - \ell_M \geq 0$. In particular, this implies that if $\ell_{j,t\lambda(p)} < \ell_{M,t\lambda(p)}$, then $t < \tau_j$.

If none of the first M arms survived the K rounds, this means there must exist $k \leq K$ and $m \leq M$ such that $x_m \in S_k$, but $x_m \notin S_{k+1}$. Therefore,

$$x_m \in S_k \wedge x_m \in S_{k+1} \implies \sum_{j\in S_k} \mathbb{1}(\ell_{j,t_k\lambda(p)} < \ell_{M,t_k\lambda(p)}) \geq \left\lfloor \frac{N}{\eta^k} \right\rfloor$$
$$\implies \sum_{j\in S_k} \mathbb{1}(t_k < \tau_j) \geq \left\lfloor \frac{N}{\eta^k} \right\rfloor \implies t_k < \tau_{\lceil N/\eta^k\rceil}.$$

Combining the above result with (2), we have, for k as defined above, the following two conclusions.

$$\frac{\epsilon}{4} > \frac{\ell_{\lceil N/\eta^k\rceil} - \ell_M}{2}, \qquad \bar{t}_k > \frac{1}{\lambda(p)}\gamma^{-1}(\epsilon/4).$$

Let $q = \min\{j \in [N]; (\ell_j - \ell_M)/2 \geq \epsilon/4\}$. The first of the above two conclusions implies that $q \geq \lceil N/\eta^k\rceil$. Additionally, the second conclusion implies that for all $j \geq q$,

$$\tau_j = \frac{1}{\lambda(p)}\gamma^{-1}\left(\frac{\ell_j - \ell_M}{2}\right) \leq \frac{1}{\lambda(p)}\gamma^{-1}(\epsilon/4) = t_k.$$

as γ^{-1} is decreasing. Therefore, for all $j \geq q$, we have $\ell_{i,t_k\lambda(p)} > \ell_{M,t_k\lambda(p)} > \ell_{m,t_k\lambda(p)}$. This means that all arms $i \geq q$ will have been eliminated before or at the same time as m. And hence, for all remaining arms, by the definition of q, we have $\ell_i - \ell_M \leq \epsilon/2$. Since this is true for all configurations surviving at stage $k+1$, it is also true for configurations at the very end, including, in particular, $\widehat{x}$. Finally, we note that since $\bar{t}_K \geq t_K \geq \gamma^{-1}(\epsilon/4)$, we have, $\widehat{\ell} - \ell_M \leq \widehat{\ell} - \ell^{\infty}(\widehat{x}) + \ell^{\infty}(\widehat{x}) - \ell_M \leq \epsilon/4 + \epsilon/2 \leq 3\epsilon/4$. □

The next result combines the above two results to provide a guarantee on a given SH instance in Algorithm 3.

Lemma 5. *Consider a SH instance in Algorithm 3 with parameter η, $N = M\eta^{K-1}$ arms, and p workers for each configuration, which has been executed for K stages with the time for first stage t_1. Let $u = \log(2M/\delta)/\eta^{K-1}$. If,*

$$1 \geq \frac{2\eta}{\lambda(p)} \int_u^1 \gamma^{-1}\left(\frac{F^{-1}(t) - \ell^{\star}}{4}\right) \mathrm{d}t +$$
$$\frac{20}{3N\lambda(p)} \log\left(\frac{2}{\delta}\right) \gamma^{-1}\left(\frac{F^{-1}(u) - \ell^{\star}}{4}, \right),$$

Then, $\widehat{\ell} - \ell^{\star} \leq 4(F^{-1}(u) - \ell^{\star})$ with probability at least $1 - \delta$.

Proof. Assume that the two events specified in Lemma 3 hold, which they do with probability $\geq 1 - \delta$. By starting with given condition we can show (steps skipped due to space constraints), $NKt_1p \geq B_{\text{SH}}(4(F^{-1}(u) - \ell^{\star}))$. We can now apply Lemma 4 to obtain $\widehat{\ell} - \ell^{\infty}(x_M) \leq 3(F^{-1}(u) - \ell^{\star})$, where x_M is the M^{th} configuration when the N configurations are ordered according to their terminal losses. Applying the first event of Lemma 3, we have $\widehat{\ell} - \ell^{\star} \leq \widehat{\ell} - \ell^{\infty}(x_M) + \ell^{\infty}(x_M) - \ell^{\star} \leq 4(F^{-1}(u) - \ell^{\star})$. □

We are now ready to prove Theorem 2.

Proof of Theorem 2. Recall that Algorithm 3 executes at least $q_{\star}$ instantiations where $q_{\star}$ is as defined in line 2. Here, the i^{th} stage will have $\mathbf{N}[i] = \eta^{K-1}\nu^{q_{\star}-i}$ configurations and use $\mathbf{P}[i] = \nu^i$ workers per configuration.

We will now apply Lemma 5 with $\delta \leftarrow \delta/q_{\star}$ for each of $q_{\star}$ instances. Now assume that the condition stated in the theorem holds for a given $\delta \in (0, 1)$. This means, for some $j \leq q_{\star}$ the expression inside the minimum is smaller than 1. Let $u_{\star} = \log(2/\delta)/R_{\star}$ and $u = \log(2Mq_{\star}/\delta)/\eta^{K-1}$. By noting that $u_{\star} \geq u$ and that γ^{-1} is a decreasing function, we can show (steps skipped due to space constraints),

$$1 \geq \frac{2\eta}{\lambda(\mathbf{P}[i])} \int_u^1 \gamma^{-1}\left(\frac{F^{-1}(t) - \ell^{\star}}{4}\right) \mathrm{d}t +$$
$$\frac{20}{3\mathbf{N}[i]\lambda(\mathbf{P}[i])} \log\left(\frac{2q_{\star}}{\delta}\right) \gamma^{-1}\left(\frac{F^{-1}(u) - \ell^{\star}}{4}, \right).$$

That is, the condition in Lemma 5 holds for j. Applying its conclusion yields, $\widehat{\ell} - \ell^{\star} \leq 4(F^{-1}(\log(2q_{\star}/\delta)/R_{\star}) - \ell^{\star})$. □

REFERENCES

[1] Maximilian Balandat, Brian Karrer, Daniel R Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2019. Botorch: Programmable bayesian optimization in pytorch. *arXiv preprint arXiv:1910.06403* (2019).

[2] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems*.

[3] Sébastien Bubeck and Nicolò Cesa-Bianchi. 2012. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Foundations and Trends in Machine Learning* (2012).

[4] Alexandra Carpentier and Andrea Locatelli. 2016. Tight (lower) bounds for the fixed budget best arm identification bandit problem. In *Conference on Learning Theory*. 590–604.

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

[6] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2010. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision* 88, 2 (June 2010), 303–338.

[7] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. *CoRR* abs/1807.01774 (2018). arXiv:1807.01774 http://arxiv.org/abs/1807.01774

[8] Matthias Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. 2015. Efficient and robust automated machine learning. *Advances in Neural Information Processing Systems* 28 (01 2015), 2944–2952.

[9] Johnu George, Ce Gao, Richard Liu, Hou Gang Liu, Yuan Tang, Ramdoot Pydipaty, and Amit Kumar Saha. 2020. A Scalable and Cloud-Native Hyperparameter Tuning System. arXiv:2006.02085 [cs.DC]

[10] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1487–1495.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[12] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-based Optimization for General Algorithm Configuration. In *LION*.

[13] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *USENIX Symposium on Network Design and Implementation (NDSI 21)*. USENIX Association.

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

[15] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. 1993. Lipschitzian Optimization Without the Lipschitz Constant. *J. Optim. Theory Appl.* (1993).

[16] Kirthevasan Kandasamy, Karun Raju Vysyaraju, Willie Neiswanger, Biswajit Paria, Christopher R Collins, Jeff Schneider, Barnabas Poczos, and Eric P Xing. 2020. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *Journal of Machine Learning Research* 21, 81 (2020), 1–27.

[17] Zohar Karnin, Tomer Koren, and Oren Somekh. 2013. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*. 1238–1246.

[18] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[19] Ya Le and Xuan Yang. 2015. Tiny ImageNet Visual Recognition Challenge.

[20] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.

[21] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2018. Massively Parallel Hyperparameter Tuning. In *Proceedings of Workshop on ML Systems in The Thirty-second Annual Conference on Neural Information Processing Systems (NIPS)*.

[22] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. 2019. HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. ACM, New York, NY, USA, 61–73. https://doi.org/10.1145/3357223.3362719

[23] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv preprint arXiv:1807.05118* (2018).

[24] Richard Liaw, Ujval Misra, Lisa Dunlap, Romil Bhardwaj, Alexey Tumanov, Joey E. Gonzalez, and Ion Stoica. 2021. Rubberband: Cloud Based Hyperparameter Tuning. *EuroSys* (2021).

[25] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[26] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. 2011. Reading Digits in Natural Images with Unsupervised Feature Learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

[27] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 1–18. https://www.usenix.org/conference/osdi21/presentation/qiao

[28] Bing Shuai, Ting Liu, and Gang Wang. 2016. Improving Fully Convolution Network for Semantic Segmentation. *CoRR* abs/1611.08986 (2016). arXiv:1611.08986 http://arxiv.org/abs/1611.08986

[29] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition.

[30] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*.

[31] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. (2018). arXiv preprint 1804.07461.

[32] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *CoRR* abs/1910.03771 (2019). arXiv:1910.03771 http://arxiv.org/abs/1910.03771