
METAOPTIMIZATION ON A DISTRIBUTED SYSTEM FOR DEEP REINFORCEMENT LEARNING

Greg Heinrich¹ Iuri Frosio¹

ABSTRACT

Training intelligent agents through reinforcement learning is a notoriously unstable procedure. Massive parallelization on GPUs and distributed systems has been exploited to generate a large amount of training experiences and consequently reduce instabilities, but the success of training remains strongly influenced by the choice of the hyperparameters. To overcome this issue, we introduce HyperTrick, a new metaoptimization algorithm, and show its effective application to tune hyperparameters in the case of deep reinforcement learning, while learning to play different Atari games on a distributed system. Our analysis provides evidence of the interaction between the identification of the optimal hyperparameters and the learned policy, that is typical of the case of metaoptimization for deep reinforcement learning. When compared with state-of-the-art metaoptimization algorithms, HyperTrick is characterized by a simpler implementation and it allows learning similar policies, while making a more effective use of the computational resources in a distributed system.

1 INTRODUCTION

The Reinforcement Learning (RL) field has been recently revitalized by the advent of Deep Learning (DL), with the development of new training algorithms (Lillicrap et al., 2015; Mnih et al., 2015; 2016) effectively applied in several fields including, among others, gaming, robotics, and finance. Despite these recent successes, training in RL remains an unstable procedure that requires fine hyperparameter tuning.

The causes of instability in RL are copious. High correlation between training data generates instability; the fact that it can be reduced by collecting data from a wide set of agents, acting in parallel but in different environments (Mnih et al., 2016; Babaeizadeh et al., 2016; 2017), has triggered the investigation of GPUs and distributed systems for the simulation of RL environments (Nair et al., 2015; Espeholt et al., 2018; Stooke & Abbeel, 2018). Value-based algorithms assign a numerical value to each state observed by the agent, and develop policies aimed at reaching high value states; training instabilities are associated in this case with Monte Carlo sampling, that requires to play an entire episode before computing an unbiased, high variance estimate of the value of each state. N-steps methods reduce the variance by boosting (Lillicrap et al., 2015; Mnih et al., 2016), at the cost of increasing the bias, whereas identifying the optimal bias-variance trade-off is a non-trivial problem (Buckman

et al., 2018). Instability is also caused by falls into local minima associated to sub-optimal policies, that can be prevented by adding an entropy term to the cost function which favours exploration over exploitation (Lillicrap et al., 2015; Mnih et al., 2015; 2016). Beyond this, far- or near-sighted agents can be more or less prone to fall into local minima when learning different tasks.

Modern RL algorithms, and especially those based on DL, use several hyperparameters to control most of the previously mentioned instability factors. Careful hyperparameter tuning is required to balance between speed, effectiveness, and stability of the training process. The general problem of identifying an optimal set of hyperparameters, while solving the underneath optimization problem, is referred to as *metaoptimization*. In RL it is often solved only after a long sequence of ineffective, time-consuming, trial-and-error, training attempts.

We tackle metaoptimization for RL on a distributed system, to achieve convergence towards a reasonable, learned policy with a minimal effort for the user. When compared to traditional metaoptimization, and in particular when considering a distributed system, the specific case of deep RL shows some peculiarities, like the fact the the choice of the hyperparameters may strongly affect the computational cost of an experiment, and the high instability of the training procedure. Our contribution here is a thorough analysis of metaoptimization for deep RL and a new algorithm particularly suitable for this case. More in detail:

- We introduce HyperTrick, a metaoptimization proce-

¹NVIDIA, Santa Clara, USA. Correspondence to: Greg Heinrich <gheinrich@nvidia.com>, Iuri Frosio <ifrosio@nvidia.com>.

ture that generalizes Successive Halving (Jamieson & Talwalkar, 2016). HyperTrick is implemented on top of MagLev (Farabet, 2018), a recently introduced training and inference framework to manage distributed systems.

- We demonstrate the effectiveness of HyperTrick in deep RL, to learn policies for several Atari games through GA3C (Babaeizadeh et al., 2016; 2017), with minimal effort on the user side for hyperparameter setting.
- We show evidence of the interaction between the optimized set of hyperparameters and the learned policies, which is peculiar of metaoptimization for RL.
- We compare HyperTrick with the state-of-art Hyperband (Li et al., 2016) and show that HyperTrick has a simpler implementation that does not require any support for preemption, it achieves a higher occupancy by effectively releasing and reallocating computational resources during metaoptimization, while reaching similar results in terms of learned policies.

2 RELATED WORK

RL Algorithms: Recent advances have been triggered by the development of novel algorithms for DL agents, but such advances did not come for free. RL is notoriously unstable when the action-value function is estimated by a nonlinear function approximator such as a Deep Neural Network (DNN), because of correlations in the sequence of observations, changes in the policy causing changes in the data distribution during training, and correlations between the action-values and the target values (Mnih et al., 2015). The DQN approach (Mnih et al., 2015) reduces these instabilities through a large replay memory and an iterative update rule that adjusts the action-values towards target values that are only periodically updated. Other learning procedures inspired by DQN achieve faster and more stable convergence: Prioritized DQN (Schaul et al., 2015) gives priority to significant experiences in the replay memory. Double-DQN (van Hasselt et al., 2015) separates the value function estimation and action selection, reducing the DQN tendency to be overly optimistic when evaluating its choices. Dueling Double DQN (Wang et al., 2015) goes a step further by explicitly splitting the computation of the value and advantage functions within the network.

Actor-critic methods, like A3C (Mnih et al., 2016) and its GPU version, GA3C (Babaeizadeh et al., 2016; 2017), outperform the DQN methods. Actor-critic methods alternate policy evaluation and improvement steps; both the actor and the critic are trained during learning. The critic is often modelled by a n-step bootstrapping method, which reduces the variance and stabilizes learning when compared to pure

policy gradient methods. Parallelization allows simulating multiple environments: it increases the diversity of the experiences collected from the agents, reducing the correlation in the observations. An entropy term is also generally included to favour exploration. Nonetheless, the stability and convergence speed of DQN and actor-critic methods strongly depend on the choice of several hyperparameters, such as the learning rate, or the number of steps used for bootstrapping. For instance, 50 different learning rates are tested for each Atari game in (Mnih et al., 2016), to guarantee convergence towards a reasonable policy.

RL on Distributed Systems: Distributed systems are commonly used in RL, with the aim of generating as many experiences as possible, as convergence towards an optimal policy is achieved only if a sufficiently large number of experiences is consumed by the RL agent. For instance, Gorilla DQN (Nair et al., 2015) outperforms DQN by using 100 actors on 31 machines, 100 learners, and a central parameter server with the DNN model. IMPALA (Espeholt et al., 2018) employs hundreds of CPUs and it solves an Atari game in a few minutes; a similar result can be achieved by resorting to a multi-GPU system (Stooke & Abbeel, 2018). Beyond speeding up RL, the aforementioned approaches achieve training stability by dramatically increasing the number of simulated environments, and thus increasing the batch size and the diversity of the collected experiences. Nonetheless, even in this case hyperparameter setting remains critical to guarantee convergence, and the proper configuration has to be identified through a time consuming trial-and-error procedure. For instance, it has been shown that the learning rate and the batch size have to be properly scaled to guarantee the convergence of many deep RL algorithms on large distributed systems, but it has also been observed that some of them (like Rainbow-DQN) do not scale beyond a certain point (Stooke & Abbeel, 2018). Moreover, not all RL algorithms naturally scale to a distributed implementation: for instance GA3C (Babaeizadeh et al., 2016; 2017) can hardly benefit from distribution on a multi-GPU system, as it is limited by the CPU time required to generate experiences and bandwidth to move data from the RAM to the GPU. On the other hand, we can still leverage a distributed system to explore the hyperparameter space and learn an optimal policy; this is the metaoptimization approach described here.

Metaoptimization on Distributed Systems: Metaoptimization consists in finding a set of optimal hyperparameters, while solving an underneath optimization problem that depends on such hyperparameters. Grid and random search are basic metaoptimization methods, based on *parallel search*: a wide exploration of the hyperparameter space is performed by parallel optimization processes with different hyperparameters; the best hyperparameter set is conse-

quently identified, but one limit of this approach is that the optimization processes do not share any information. Other basic metaoptimization procedures, such as hand tuning or Bayesian-Optimization (Shahriari et al., 2016), follow a *sequential search* paradigm, where the results achieved by completed optimization processes drive the selection of new hyperparameters. The search for the optimal hyperparameter set is in this case local, and evidence of the optimal setting generally emerges only after a large number of evaluations - this is a limiting factor, especially when the underneath optimization problem is computationally intensive. When the underneath problem is solved iteratively, partial results can be used as proxy and hyperparameter configurations that are deemed less promising can be abandoned quickly; this scheme is referred to as *Early Stopping*.

Population Based Training (PBT, (Jaderberg et al., 2017)) leverages the benefits of parallel search, sequential search, and early stopping, merging them into a single, metaoptimization procedure that automatically selects hyperparameters during training, while also allowing online adaptation of the hyperparameters to enable non-stationary training regimes and the discovery of complex hyperparameter schedules; it performs online model selection to maximize the time spent on promising models. PBT is naturally implemented on a distributed system, by assigning one or more optimization processes to each node. Hyperband (Li et al., 2016), an extension of Successive Halving (Jamieson & Talwalkar, 2016), is another algorithm that uses adaptive resource allocation and early stopping to solve metaoptimization problems. In Successive Halving, the exploration of the hyperparameter space is performed in multiple phases. Each phase is given a total resource budget B , equally divided between N workers, where each worker is solving the underneath optimization problem using a different set of hyperparameters. The worst half of the workers are terminated at the end of each phase, while the other ones are allowed to run. The main issue of this approach is that, for a fixed B , it is not clear a priori whether it is better to consider many hyperparameter configurations (large N) with a small average training time, or a small number of configurations N with longer average training times. Hyperband addresses the problem of balancing breadth (large N) versus depth (small N) search by calling Successive Halving as a subroutine and considering several possible values of N for a fixed B ; each (B, N) pair is called a *bracket* in Hyperband. Hyperband’s inputs are the maximum amount of resource allocated to a single configuration, R ; and the proportion of configurations discarded in each round of Successive Halving, η . In the first bracket, Hyperband sets N to its smallest value for maximum exploration and runs Successive Halving for the given B/N ratio (under the constraint imposed by R). For any successive bracket, Hyperband reduces N by a factor of approximately η until, in the last bracket, one

final configuration (performing classical random search) is left. In practice, Hyperband performs a grid search over N by running several instances of Successive Halving. Under this scheme, slow learners who may initially under-perform are given a chance to run for longer and may ultimately yield better results.

Unfortunately, any attempt to implement Successive Halving and Hyperband on a distributed system easily reveals some practical issues. These algorithms are well suited for systems with a unique node or a set of equivalent nodes, but in the case of an heterogeneous distributed system, the effective assignment of a fixed budget B/N to any worker is problematic: nodes associated to fast workers may be idle, waiting to synchronize with the slow nodes at the end of each phase. Quite remarkably, this may happen even on a homogeneous system if the hyperparameters affect the computational cost of the underneath optimization problem, like in the case of metaoptimization for RL. Although an asynchronous variation of Successive Halving can be used to partially solve this problem (Li et al., 2018), a second issue is that, if the number of compute nodes is not large enough, some workers need to yield to other workers at the end of each phase and resume execution. This requires explicit support for preemption, and introduces an additional overhead for context switching. These two issues may be alleviated by allowing mapping of one worker to multiple nodes over the course of the training process, which however requires again a non trivial implementation and incurs a context switching cost.

3 HYPERTRICK ON MAGLEV

We propose a metaoptimization procedure based on parallel search and early stopping, which frees computational resources during the process, re-allocates them for new experiments, and does not require support for preemption. We perform our experiments on the MagLev platform, whose architecture is briefly described in the following.

3.1 The MagLev Architecture

MagLev is a platform built over Kubernetes (Kub) that supports the execution of parallel experiments on a distributed system, with homogeneous or heterogenous nodes, each including one or multiple CPUs and GPUs. Metaoptimization can be implemented in different flavors in MagLev by performing a number of parallel optimization experiments, each exploring one point of the hyperparameter space. A hyperparameter optimization service runs in MagLev to this aim (see Fig.1). Each experiment is executed by a *worker*; each compute node can host one or multiple workers (and thus run multiple experiments) at the same time. MagLev allows each experiment to continuously expose information about its status, as well as other metrics. Typically each worker

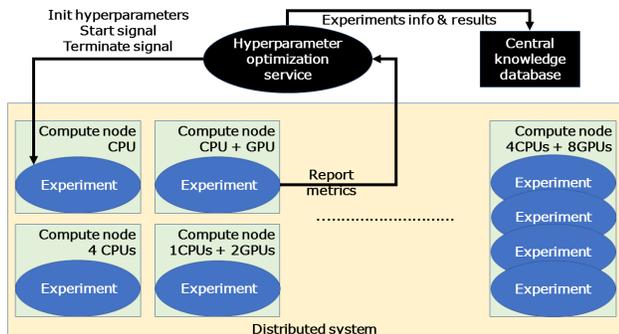


Figure 1: The architecture of the MagLev platform, supporting heterogeneous nodes with CPUs and GPUs. The hyperparameter optimization service randomly initializes the hyperparameters of each experiment, and collects metrics and results into a central knowledge database. Each node runs one or more experiments at the same time. The optimization service can also terminate some of the experiments to re-allocate the compute nodes.

executes an experiment in multiple *phases* and reports a set of metrics to the hyperparameter optimization service at the end of each phase. The hyperparameter optimization service also manages the initial sampling of the hyperparameter space and it is backed by a central knowledge database that collects information about experiments, their hyperparameter configurations, and the reported metrics. The workers periodically query the service to be notified whether to continue running or not. This allows implementing (among other procedures) several metaoptimization algorithms that release and re-allocate computational resources, including our metaoptimization method, illustrated in Section 3.2.

3.2 HyperTrick

We propose HyperTrick, a metaoptimization algorithm partially inspired by Hyperband (Li et al., 2016), Successive Halving (Jamieson & Talwalkar, 2016), and PBT (Jaderberg et al., 2017), aimed at improving the utilization of parallel resources in a distributed system, especially when the hyperparameter configuration affects the computational cost of an experiment, by merely managing the early termination of unpromising workers. We describe here our asynchronous, multithread implementation, which suits well on a distributed system, although a single-thread implementation is also possible.

In HyperTrick, each worker explores one hyperparameter set over a number of phases N_p , where each phase may correspond to a number of training iterations, a given amount of wall-clock time, or any other user-defined arbitrary units of work. Beyond N_p , HyperTrick’s inputs are the initial number of workers, W_0 , and the target eviction rate r , which is the expected ratio of workers terminated after each phase,

although HyperTrick stochastically allows a different ratio of workers to proceed to the next phase. The first step in HyperTrick is to launch a number of experiments equal to the minimum between W_0 and the number of nodes in the distributed system, N (lines 2-3 in Algorithm 1). Differently from Successive Halving and Hyperband, HyperTrick does not employ any synchronization mechanism: each worker runs independently from others. While different workers are running (line 4 in Algorithm 1), they may be in different phases. Workers asynchronously report performance metrics at the end of each phase to the central hyperparameter optimization service (line 5), which stores the statistics (line 6) and then uses the HyperTrick’s rule to decide whether to let a worker continue (lines 7). When a worker is terminated, the compute node is reallocated to a new worker to investigate a new set of hyperparameters, starting from the first phase (lines 8-10 in Algorithm 1). HyperTrick eventually returns the best observed configuration (line 11).

The HyperTrick’s rule to decide whether a worker should continue or not at the end of each phase is the following. Within each phase, HyperTrick first operates in Data Collection Mode (DCM), collecting metrics and letting all workers proceed to the next phase. Once sufficient statistics have been collected, HyperTrick switches to the Worker Selection Mode (WSM) for that phase and starts terminating underperforming workers. Let W_p be the number of workers at phase p . For the given target eviction rate r , the expected value of W_p is given by:

$$E[W_p] = W_0(1 - r)^p. \quad (1)$$

The number of workers required to complete the phase p before switching from DCM to WSM, W_p^{DCM} , is:

$$W_p^{DCM} = W_0(1 - \sqrt{r})(1 - r)^p. \quad (2)$$

Once HyperTrick switches to WSM in phase p , any worker w that reports a metric $m_w(p)$ in the lower \sqrt{r} quantile is terminated. We demonstrate in the following that using Eq. (2) to switch from DCM to WSM leads to the expected target eviction rate in Eq. (1).

Proof. Assumption: the process $m_w(p)$, which returns the metric of a worker w for phase p , is stationary.

Base case $p = 0$: For $p = 0$, the total number of workers is always the initial number of workers W_0 ; Eq. (1) predicts $E[W_0] = W_0(1 - r)^0 = W_0$, thus it holds for $p = 0$.

Inductive hypothesis: Suppose Eq. (1) holds for all values of p up to some k , $k \geq 0$; then, at the beginning of phase k , the expected total number of workers is $E[W_k] = W_0(1 - r)^k$.

Inductive step: in phase k , the first W_k^{DCM} workers are allowed to continue unconditionally, whereas the remaining

Algorithm 1: HyperTrick.

```

HyperTrick(W_0, r, N_p, N) //W_0 workers, r target eviction rate, N_p phases, N computational nodes.
  for (i=0; i<min(W_0, N); ++i)
    launch_experiment_thread(i, r, N_p) // launch experiment on i-th node with random hyperparameters
  while(no_more_experiment_running() == false)
    [n, m, p, h] = wait_for_experiment_report() // wait for one experiment with hyperparameters h reporting
    metric m at phase p on node n
    stats = store_statistics(stats, m, p, h) // store metrics m at phase p and hyperparameters h in stats
    if (terminate_experiment_thread_if_needed(n, m, p, stats) == true) // kill experiment with HyperTrick rule
      if (i < W_0 - 1)
        ++i
        launch_experiment_thread(n, r, N_p) // launch experiment on n-th node with random hyperparameters
  return best_of(stats)
    
```

1
2
3
4
5
6
7
8
9
10
11

W_k^{WSM} workers can be terminated by HyperTrick in WSM. We have:

$$\begin{aligned}
 E[W_k^{WSM}] &= E[W_k - W_k^{DCM}] = \\
 W_0(1-r)^k - W_0(1-\sqrt{r})(1-r)^k &= \\
 W_0\sqrt{r}(1-r)^k. & \quad (3)
 \end{aligned}$$

Out of the W_k^{WSM} workers, those that report $m_w(k)$ in the lower \sqrt{r} quantile are terminated. Because m_w is stationary, the probability of a worker being terminated is \sqrt{r} . If W_k^T is the number of workers to terminate then:

$$E[W_k^T] = E[\sqrt{r}W_k^{WSM}] = rW_0(1-r)^k. \quad (4)$$

The expected number of workers at the beginning of the next phase $k+1$ is then equal to:

$$\begin{aligned}
 E[W_{k+1}] &= E[W_k - W_k^T] = E[W_k] - E[W_k^T] = \\
 W_0(1-r)^k - rW_0(1-r)^k &= W_0(1-r)^{k+1} \quad (5)
 \end{aligned}$$

Therefore Eq. (1) holds for $p = k+1$. By induction, it also holds for all $n \in \mathbb{N}$. \square

There are several reasons for different workers to reach the end of a phase in different times: a worker may have been scheduled early or late, running on a fast or slow node, or assigned a more (or less) computationally efficient hyperparameter set. HyperTrick favors early or fast workers in the selection and balances in this way between breadth versus depth search; it takes advantage of the unpredictability of worker scheduling, run time, and reported metrics, to give early workers a higher chance to continue and let them increase the depth of their search, while late workers are discouraged. Across successive phases, HyperTrick requires low performers to be increasingly early, and laggards to perform increasingly well. Since it does not synchronize workers and immediately re-allocates any idle node to a new worker, HyperTrick also achieves an effective utilization of the available computational resources. This is a main advantage of HyperTrick when compared to Successive Halving and Hyperband, whose implicit synchronization mechanism at the end of each phase forces the fast workers to wait for

the slow ones. Idle nodes can be avoided in Successive Halving and Hyperband by allowing a dynamic scheduling of the workers on the nodes, but at the cost of implementing a preemption and yielding mechanism; on the other hand, HyperTrick does not require any preemption management, and does not incur any additional cost associated with context switches. A beneficial side effect of HyperTrick is that, even in the case of occasional failures of the workers, the effect is local to the worker that experienced the failure. The experiment that was running on the worker may be retried, or ignored, without affecting other workers. The price to be paid is the introduction of a potential bias in the final metaoptimization result, due to the random advantage given to the workers that are scheduled early. Our experimental results show that, at least in the case of metaoptimization for RL considered here, this is of little importance in practice.

For higher clarity, Fig. 2 shows HyperTrick with $W_0 = 16$ workers, $N_p = 4$ phases, 6 compute nodes, and a target eviction rate $r = 25\%$, on a toy problem. Accordingly to Eq. (2), the minimum number of workers allowed to continue at the end of the first, second and third phase, are $W_1^{DCM} = 8$, $W_2^{DCM} = 6$, and $W_3^{DCM} = 4$, respectively. Initially, 6 workers $\{W_i\}_{i=0..5}$ run 6 optimization experiments on the 6 available nodes, $\{N_i\}_{i=0..5}$. The fast worker W_0 terminates all 4 phases at time $t = 4$: the node N_0 is released and W_6 starts a new experiment on the same node. At time $t = 4.2$, W_4 is the fifth worker (after W_0, \dots, W_3) to reach the end of the third phase. HyperTrick consequently switches from DCM to WSM for the third phase: from now on, each worker completing this phase will continue only if its score is in the top $\sqrt{r} = 50\%$ of the scores at the end of the third phase. The workers $\{W_i\}_{i=0..3}$ won't be affected by this rule as they already started the fourth phase. Since W_4 reports a low metric at the end of the third phase, it is terminated by HyperTrick and N_4 is reallocated to start W_7 . At time $t = 4.5$, W_5 reaches the end of its third phase; its metric (31) is in the top half, thus W_5 is allowed to proceed to the last phase. At $t = 6$, 6 workers have already reported their metrics for the first phase, W_6 completes it and reports a low metric (8), thus it is terminated; N_0 is reallocated for

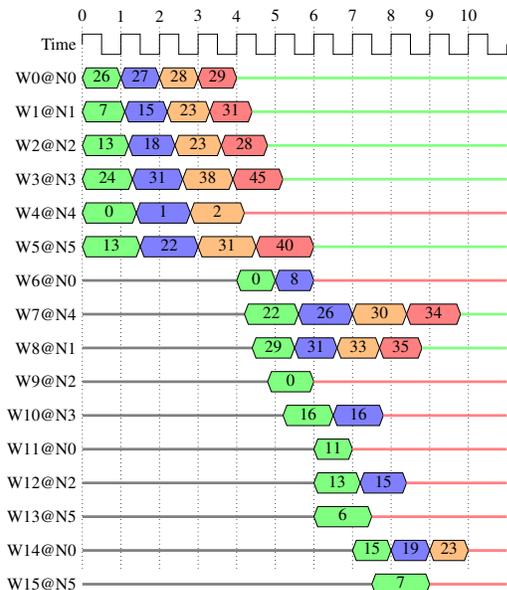


Figure 2: Metaoptimization with HyperTrick ($r = 25\%$, $W_0 = 16$) on a toy problem, $N_p = 4$ phases (in green, blue, orange, and red). The workers $\{W_i\}_{i=0..15}$ are initially scheduled on 6 nodes $\{N_i\}_{i=0..5}$. The cells show the metric $f(p)$ for each worker and phase p ; here, $f(p) = ap + b$, where a, b are random values. Green lines indicate completion of all phases, red lines indicate early termination. The phase execution time is variable: this is common in a heterogeneous systems, or when the hyperparameters affect the computational cost of the underneath optimization problem. The process takes 10 units of time.

W11. Overall, 10 units of time are required to complete the entire metaoptimization process.

Some of the advantages offered by HyperTrick are evident after analyzing Fig. 3, that shows Successive Halving, terminating 25% of the workers at the end of each phase, on the same toy problem. When compared to Hypertrick, Successive Halving takes a longer amount of time (12.1 units) and achieves a lower occupancy of the system, because of the need to synchronize workers. Successive Halving does not allow any slow learner to run to completion (e.g., $W1$ is terminated early, whereas in HyperTrick it runs to completion and achieves a final, above average metric of 31). In this example, Successive Halving requires workers to support preemption, as experiments are stopped and restarted after a while, potentially on a different node; the overhead for context switches is optimistically assumed to be zero here. A simplified implementation of Successive Halving that allocates statically each worker to one node is feasible, but it takes more time (15.3 units of time - see Fig. 8, reported in the Appendix for sake of space).

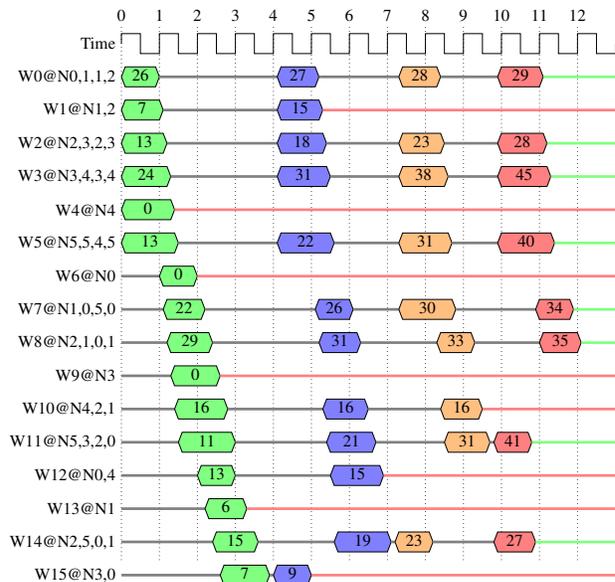


Figure 3: Metaoptimization with a variant of Successive Halving which terminates 25% of workers at the end of every phase, for the same toy problem in Fig. 2. Workers are dispatched to different nodes over the course of the entire process. For example $W1@N1, 2$ indicates that $W1$ executes the first two phases on nodes 1 and 2, respectively. The process takes 12.1 units of time.

4 GA3C

In this Section we introduce some basic concepts of RL and summarize the main characteristics of GA3C (Babaeizadeh et al., 2016; 2017), to help the reader interpret the experimental results presented in the next section.

4.1 Reinforcement Learning and REINFORCE

In RL, an agent observes a state s_t at time t and selects an action a_t , following a policy π , that maps s_t to a_t . The agent receives then a feedback from the environment in the form of a reward r_t . The goal of RL is to find a policy π that maximizes the sum of the expected rewards.

In policy-based, model-free methods, a DNN can be used to compute $\pi(a_t|s_t; \theta)$, where θ is the set of DNN weights. Algorithms from the REINFORCE (Williams, 1992) family use gradient ascent on $E[R_t]$, where $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$ is the accumulated reward from time t , discounted by the factor $\gamma \in (0, 1]$. Small values of γ generate short-sighted agents that prefer immediate rewards, whereas large γ values create agents with a long term strategy, but more difficult to train. The vanilla REINFORCE updates θ uses the gradient $\nabla_{\theta} \log \pi(a_t|s_t; \theta) R_t$, an unbiased estimator of $\nabla_{\theta} E[R_t]$; its variance is reduced by subtracting a learned *baseline* (a function of the state $b_t(s_t)$) and using the gradi-

ent $\nabla_{\theta} \log \pi(a_t | s_t; \theta) [R_t - b_t(s_t)]$ instead. One common baseline is the value function $V^{\pi}(s_t) = E[R_t | s_t]$, which is the expected return for the policy π starting from s_t . The policy π and the baseline b_t can be viewed as *actor* and *critic* in an actor-critic architecture (Sutton & Barto, 1998).

4.2 A3C and GA3C

A3C (Mnih et al., 2016), a successful RL actor-critic algorithm, uses a DNN to compute both the policy and value function. The DNN trained to play Atari games in (Mnih et al., 2016) has two convolutional layers, one fully connected layer, and ReLU activations. The DNN outputs are: a softmax layer for the policy function approximation $\pi(a_t | s_t; \theta)$, and a linear layer for $V(s_t; \theta)$. Multiple agents play concurrently and optimize the DNN through asynchronous gradient descent, with the DNN weights stored in a central parameter server. After each update, the central server propagates new weights to the agents. The variance of the critic $V(s_t; \theta)$ is reduced (at the price of an increased bias) by N-step bootstrapping: the agents send updates to the server after every t_{max} actions, or when a terminal state is reached. The cost function for the policy is:

$$\log \pi(a_t | s_t; \theta) \left[\tilde{R}_t - V(s_t; \theta) \right] + \beta H[\pi(s_t; \theta)], \quad (6)$$

where θ_t are the DNN weights θ at time t , $\tilde{R}_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_t)$ is the bootstrapped discounted reward in the time interval from t to $t+k$ and k is upper-bounded by t_{max} , and $H[\pi(s_t; \theta)]$ is an entropy term to favor exploration, whose importance is modulated by β . In the original A3C paper, t_{max} is empirically set to 5, which experimentally achieves convergence for most of the Atari games in a reasonable amount of time; nonetheless, the optimal bias-variance trade-off for the critic may be achieved for other values of t_{max} , depending on the specific game. The cost function for the estimated value function is:

$$\left[\tilde{R}_t - V(s_t; \theta) \right]^2, \quad (7)$$

which uses again the bootstrapped estimate \tilde{R}_t . The server collects gradients ∇_{θ} from both of the cost functions and uses the standard non-centered RMSProp (Tieleman & Hinton, 2012) to optimize them. The gradients of the two cost functions can be either shared or separated between agent threads, but the shared implementation is known to be more robust (Mnih et al., 2016).

A3C (Mnih et al., 2016) uses 16 agents on a 16 core CPU and takes four days to learn an Atari game (Brockman et al., 2016). Its GPU version, GA3C, is only slightly different from A3C (see details in (Babaeizadeh et al., 2016; 2017)), but it reaches convergence in about one fourth of the time and it is therefore the one adopted here for our experiments.

5 RESULTS AND DISCUSSION

5.1 Experimental setup

We implement HyperTrick (with settings reported in Table 1) on Maglev to learn to play four different Atari games (Boxing, Pong, Ms-Pacman, and Centipede) in the OpenAI Gym environment (Greg et al., 2016), through GA3C. Our system has 300 compute nodes, each including a dual 20-core Intel Xeon E5-2698 v4 2.2 GHz CPU and 8 Nvidia V100 GPUs. HyperTrick optimizes three hyperparameters: learning rate, γ , and t_{max} . A large learning rate can lead to training instabilities, whereas for a small one the convergence is slow and the capability to evade from local minima is limited. The discount factor γ determines the short/farsightedness of the agent; a small γ leads to agents that easily learn a sub-optimal policy which maximizes immediate returns, but lack from a long term strategy; a large γ generates agents that weigh future rewards more, but training is more difficult. The hyperparameter t_{max} affects both the convergence properties and the computational cost of GA3C: a large t_{max} leads to high variance estimates of \tilde{R}_t and consequently high variance updates of the cost functions in Eqs. (6) and (7); increasing t_{max} also increases the batch size, which leads to a better utilization of the GPU in GA3C, but decreases the number of policy updates per second, since a large number of frames have to be played to populate the batch. Decreasing t_{max} reduces the variance of \tilde{R}_t , but increases the bias; it also reduces the batch size, eventually leading to a higher number of biased updates per second.

In our experiments we run HyperTrick on a population of 100 workers, whose hyperparameters are randomly picked. The learning rate is sampled from a random log uniform distribution over the [1e-5, 1e-2] interval; t_{max} is sampled from a random quantized log uniform distribution over the [2, 100] interval, with an increment of 1 to pick integer values; γ is sampled uniformly from the {0.9, 0.95, 0.99, 0.995, 0.999, 0.9995, 0.9999} set.

5.2 Metaoptimization

5.2.1 Metaoptimization Results

Table 1 reports the scores achieved by HyperTrick, and compares it with the score in (Babaeizadeh et al., 2017) for GA3C and an “optimal” hyperparameter configuration, identified by a trial-and-error, for the same Atari games. HyperTrick consistently achieves comparable scores, demonstrating its effectiveness to identify an optimal policy, without any significant user intervention to set the hyperparameters.

5.2.2 Worker Selection Analysis

The first row of Fig. 4 shows the distribution of the metrics (Atari game scores) during our experiments with Hyper-

RL Metaoptimization on a Distributed System

Game	Episodes per Phase	N_p	r	α (min[α], E[α])	Score (GA3C)	Score (HyperTrick)
Boxing	2500	10	25%	48.2% (18.87%, 37.75%)	92	98
Centipede	2500	10	25%	52.2% (18.87%, 37.75%)	7386	8707
Ms Pacman	2500	10	25%	46.1% (18.87%, 37.75%)	1978	2112
Pong	2500	5	25%	59.1% (30.51%, 61.02%)	18	18

Table 1: HyperTrick parameters (episodes per phase; number of phases, N_p ; target eviction rate, r) for metaoptimization on four Atari games. The fifth column indicates the measured (α), minimum (min[α]) and expected (E[α]) worker completion rate achieved by HyperTrick; 100% corresponds to running all the workers to completion. The last two columns report the score achieved with a standard implementation of GA3C (Babaeizadeh et al., 2017) and that obtained with HyperTrick.

Trick. The number of active workers drops after each phase. Few workers stop before reaching the end of a phase (small drops in Fig. 4): they crash or hang for different reasons, but do not affect the output of HyperTrick. As expected, workers with a low metric are gradually eliminated and the fraction of those achieving a high score increases during metaoptimization. More in detail, computationally efficient workers (*e.g.* those with a t_{max} that maximizes the frame generation rate) generally continue to the next phase, independently from their score, because they reach the end of each phase early, with HyperTrick in DCM. Computationally demanding workers arrive late at the end of a phase, and they continue only if their score is sufficiently high - in other words, only if they are sample efficient. Fig. 4 also shows the learning curves for the entire set of workers; we notice that unstable training processes (that are common for sub-optimal choices of the hyperparameters, *e.g.* a large learning rate) have a high chance of being eliminated, possibly because of the noise in the reported score at the end of each phase. In practice, computationally efficient workers are allowed to explore a given hyperparameters configuration in depth, while workers with a higher computational cost must be sample efficient and stable to pass to the next phase. Workers that are computationally and sample inefficient at the same time are terminated soon.

Fig. 5 shows the active workers in the hyperparameter space in different phases. In general, each game has one optimal configuration of learning rate, γ , and t_{max} , which is effectively identified by HyperTrick. Workers with large t_{max} and low scores are terminated in the first phases of the metaoptimization process: these are the computationally intensive, slow learners, that often reach the end of a phase in WSM mode.

5.2.3 Worker Completion Rate

We define the *worker completion rate*, α , as the fraction of phases completed by a metaoptimization algorithm. In the case of Grid Search, with no early stopping, $\alpha = 100\%$. A low value of α indicates that most of the workers have been terminated early; for the same result obtained on the underneath optimization problem by two metaoptimization

algorithms, the one with the lowest α identifies promising hyperparameters configurations more effectively. Based on the expected (Eq. (1)) and minimum (Eq. (2)) number of workers in each phase, we compute the minimum and expected α for HyperTrick as:

$$\min[\alpha] = \frac{\sum_{p=0}^{N_p-1} W_p^{DCM}}{N_p W_0} = \frac{\sum_{p=0}^{N_p-1} (1 - \sqrt{r})(1 - r)^p}{N_p} = (1 - \sqrt{r})[1 - (1 - r)^{N_p}]/(r N_p), \quad (8)$$

$$E[\alpha] = \frac{\sum_{p=0}^{N_p-1} (1 - r)^p}{N_p} = \frac{1 - (1 - r)^{N_p}}{r N_p}. \quad (9)$$

Table 1 shows that, for HyperTrick, α is experimentally close to its expectation for Pong, and slightly higher in other cases, suggesting that the experimental eviction rate is generally lower than the target r . The visual inspection of the learning curves in Fig. 4 gives a possible interpretation of this phenomenon: these curves are mostly regular and monotonic for Pong, and more irregular for the other games. In this last case, HyperTrick performs a noisy worker selection in WSM, which may increase the chance of slow or sample inefficient workers to pass the first selections, just to be eliminated in the next phases. On the other hand, the minimum completion rate $\min[\alpha]$ is hardly achievable in practice, as in this case HyperTrick in WSM should terminate all workers. This represents an upper bound for α , which could be approached by using any a-priori knowledge about hyperparameters and their effect on the learning curve. This may lead to optimal scheduling planning in HyperTrick - since this topic goes beyond the scope of our paper, we leave it for future investigation. Notice that E[α] for HyperTrick is also the exact completion rate for a vanilla implementation of Successive Halving, assuming no overhead for context switching; HyperTrick achieves an experimentally higher α in our tests, but it is expected to complete the metaoptimization earlier and to achieve a higher system occupancy (*c.f.* Figs. 2 and 3), since it does not synchronize the workers. Experimental evidence of this is provided in Section 5.2.4.

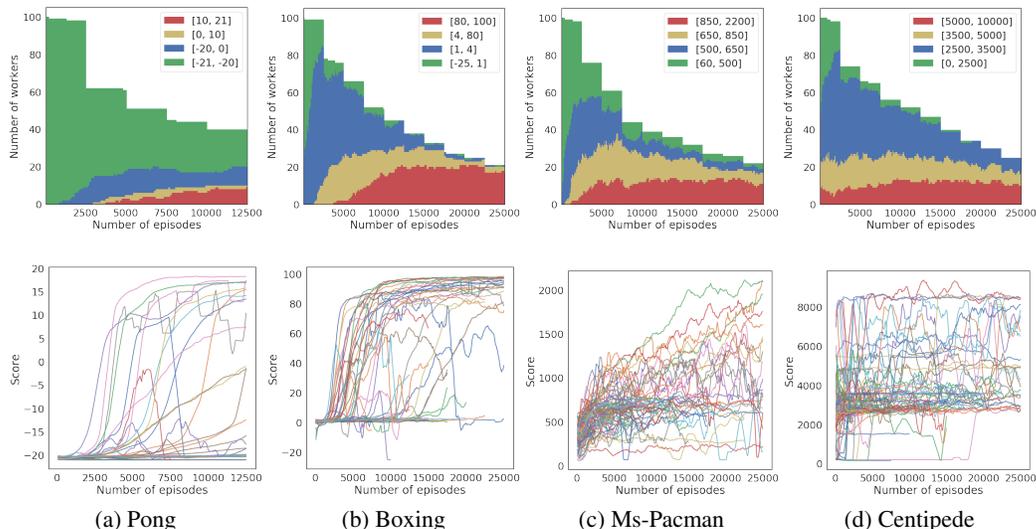


Figure 4: The first row shows the distribution of the score during metaoptimization with HyperTrick, for four different Atari games learned through GA3C. The lower row shows the learning curves for the entire population of workers.

5.2.4 Comparison Against Hyperband

We compare HyperTrick and Hyperband (Li et al., 2016), a recently proposed metaoptimization algorithm which calls Successive Halving multiple times as a sub-routine, in the attempt of automatically finding the optimal balance between breadth and depth of the search. For Hyperband, we set $\eta = 3$ (as in (Li et al., 2016)) and $R = 27$, which leads to a total of 4 brackets $s = \{3, 2, 1, 0\}$ and $27 + 9 + 6 + 4 = 46$ configurations of hyperparameters explored by Hyperband (first row in Table 2), initialized randomly. We define a unit of computational resource ($r_{i,s}$ in Table 2) as a set of 500 training episodes, such that the maximum number of training episodes in one phase of Successive Halving is equal to 13,500. Since each bracket in Hyperband represents an independent instance of Successive Halving, we run the four brackets in parallel on Maglev. Experiments are restarted from the first iteration in each phase of Successive Halving. For this configuration of Hyperband, we can compute the worker completion rate for each bracket (reported in Table 2) as $\alpha_s = n_{0,s}R / \sum_i (n_{i,s}r_{i,s})$; for the entire Hyperband algorithm we have $\alpha = \sum_s (n_{0,s}R) / \sum_s \sum_i (n_{i,s}r_{i,s}) = 32.61\%$. We run Hyperband on 46 nodes and guarantee that all workers start at the same time, with no delay. For a fair comparison, we run HyperTrick on the same 46 configurations of hyperparameters, on the same nodes, and $N_p = 27$ phases; we compute the target eviction rate of HyperTrick to guarantee that the overall compute time is similar for the two metaoptimization algorithms. Since both the algorithms analyze the same number of hyperparameter configurations, this is achieved by setting the expected worker completion rate of HyperTrick equal to that of Hy-

	$s = 3$	$s = 2$	$s = 1$	$s = 0$
i	$n_{i,3} r_{i,3}$	$n_{i,2} r_{i,2}$	$n_{i,1} r_{i,1}$	$n_{i,0} r_{i,0}$
0	27 1	9 3	6 9	4 27
1	9 3	3 9	2 27	—
2	3 9	1 27	—	—
3	1 27	—	—	—
α_s	14.81%	33.33%	66.67%	100%

Table 2: The Hyperband configuration used in testing ($\eta = 3$, $R = 27$) leads to the definition of 4 brackets $s = \{0, 1, 2, 3\}$, each $(n_{i,s}, r_{i,s})$ corresponding to an instance of Successive Halving; $n_{i,s}$ indicates the number of experiments running in the i -th phase of Successive Halving; whereas $r_{i,s}$ indicates the computational resources allocated for each experiment. In the specific case considered here, $r_{i,s} = 1$ corresponds to 500 training episodes while running GA3C. The worker completion rate α_s is also indicated for each individual bracket, whereas it is equal to 32.61% for the entire Hyperband algorithm.

perband, i.e. $E[\alpha] = 32.61\%$, and iteratively solving Eq.(9) for r , to get $r = 10.82\%$.

Experimental results are summarized in Table 3 and illustrated in Fig. 6. Hyperband and HyperTrick identify the same optimal configuration of hyperparameters for Pacman and Boxing; the slightly better score reported for Hyperband in Table 3 is due to the non-deterministic nature of the training and evaluation procedures. In the case of Pong and Centipede the hyperparameter configurations are different, but the final score similar, possibly because multiple hyperparameter configurations can lead to similar results in this

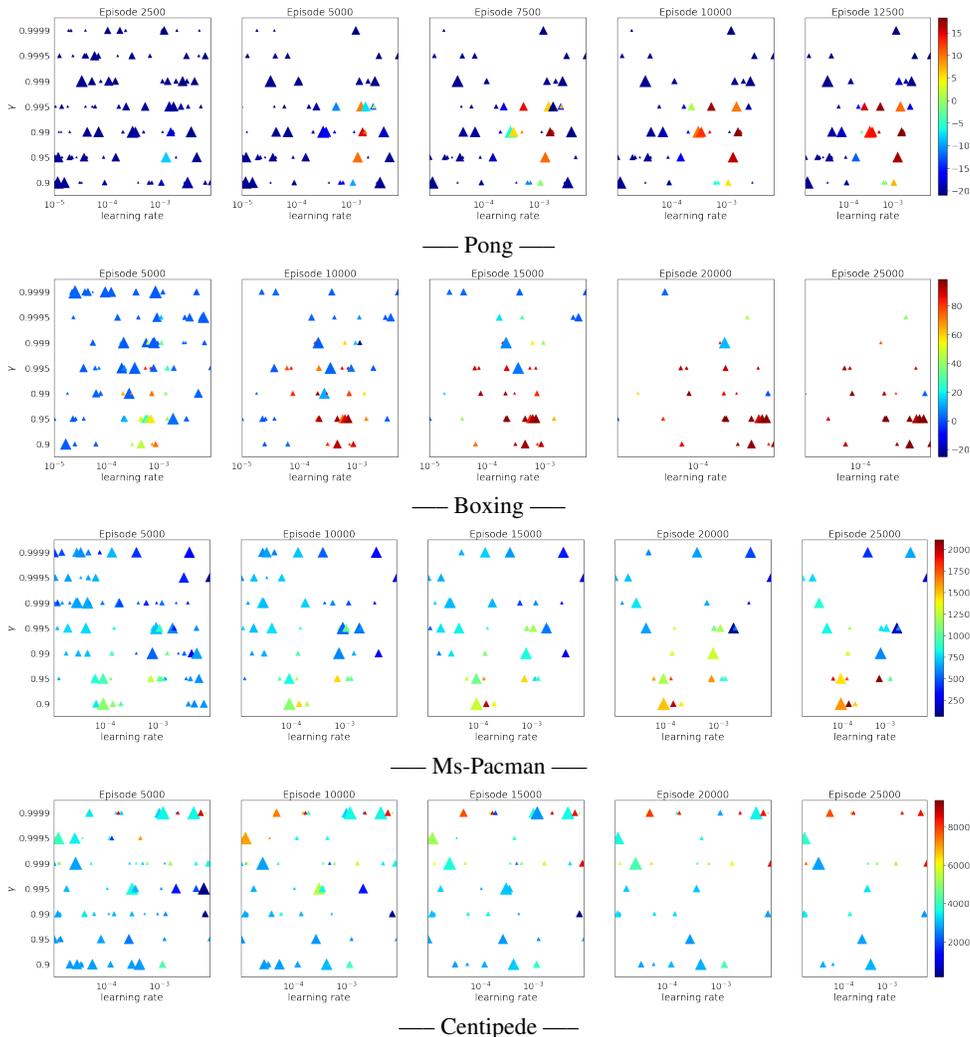


Figure 5: Selection of the optimal hyperparameters through HyperTrick. Each row represents a different Atari game learned with GA3C. Each triangle is a worker with a different hyperparameters configuration (learning rate on the x axis, discount γ on the y axis, the size of the triangle is proportional to t_{max}). Workers are terminated by HyperTrick from left to right.

case (see also Fig. 5). Despite the fact that HyperTrick and Hyperband have the same expected α , and therefore execute (on average) the same overall number of operations, HyperTrick generally terminates the metaoptimization procedure in a shorter amount of time. The last row of Fig. 6 highlights that HyperTrick achieves a higher occupancy of the computational nodes in the distributed system, which explains the overall shorter time; this is a direct effect of the lack of synchronization in HyperTrick, which immediately reallocates a node for a new experiment when a worker is terminated, whereas Successive Halving in each bracket of Hyperband pays an additional overhead due to phase synchronization, that leads some of the workers to remain idle (middle row of Fig. 6). Table 3 also highlights that HyperTrick generally identifies the best configuration in a significantly shorter amount of wall time, compared to Hyperband. The only ex-

ception to this is the case of Centipede in Table 7, but Fig. 6 reveals that the best configuration identified by HyperTrick is associated to a late, slight oscillation of the maximum score, while both HyperTrick and Hyperband effectively identify a close-to-optimal solution in a similar amount of time. Another practical issue (actually observed during our experiments) in HyperBand is that a single point of failure may jeopardize the entire Successive Halving bracket, since workers need to wait for each other. This is not the case with HyperTrick due to the absence of synchronization points.

5.3 RL Training Results

Based on the results stored in the knowledge database in MagLev, we investigate the selection of the optimal hyperparameters for each game and how they affect the learned

RL Metaoptimization on a Distributed System

Game	Method	Best Score	Total Wall Time	Time To Best Score	Best Config		
					LR	γ	T_{max}
Boxing	HyperBand	96	51h	29h	$3.3e^{-4}$	0.99	13
	HyperTrick	95	38h	13h	$3.3e^{-4}$	0.99	13
Centipede	HyperBand	8521	42h	2h	$5.4e^{-3}$	0.9995	72
	HyperTrick	8667	38h	29h	$1.2e^{-4}$	0.9999	33
Pacman	HyperBand	2456	31h	26h	$1.6e^{-4}$	0.95	73
	HyperTrick	2243	27h	16h	$1.6e^{-4}$	0.95	73
Pong	HyperBand	17.5	48h	47h	2.0^{-3}	0.95	64
	HyperTrick	17.8	39h	22h	$5.9e^{-4}$	0.995	6

Table 3: HyperBand vs HyperTrick results on four Atari games.

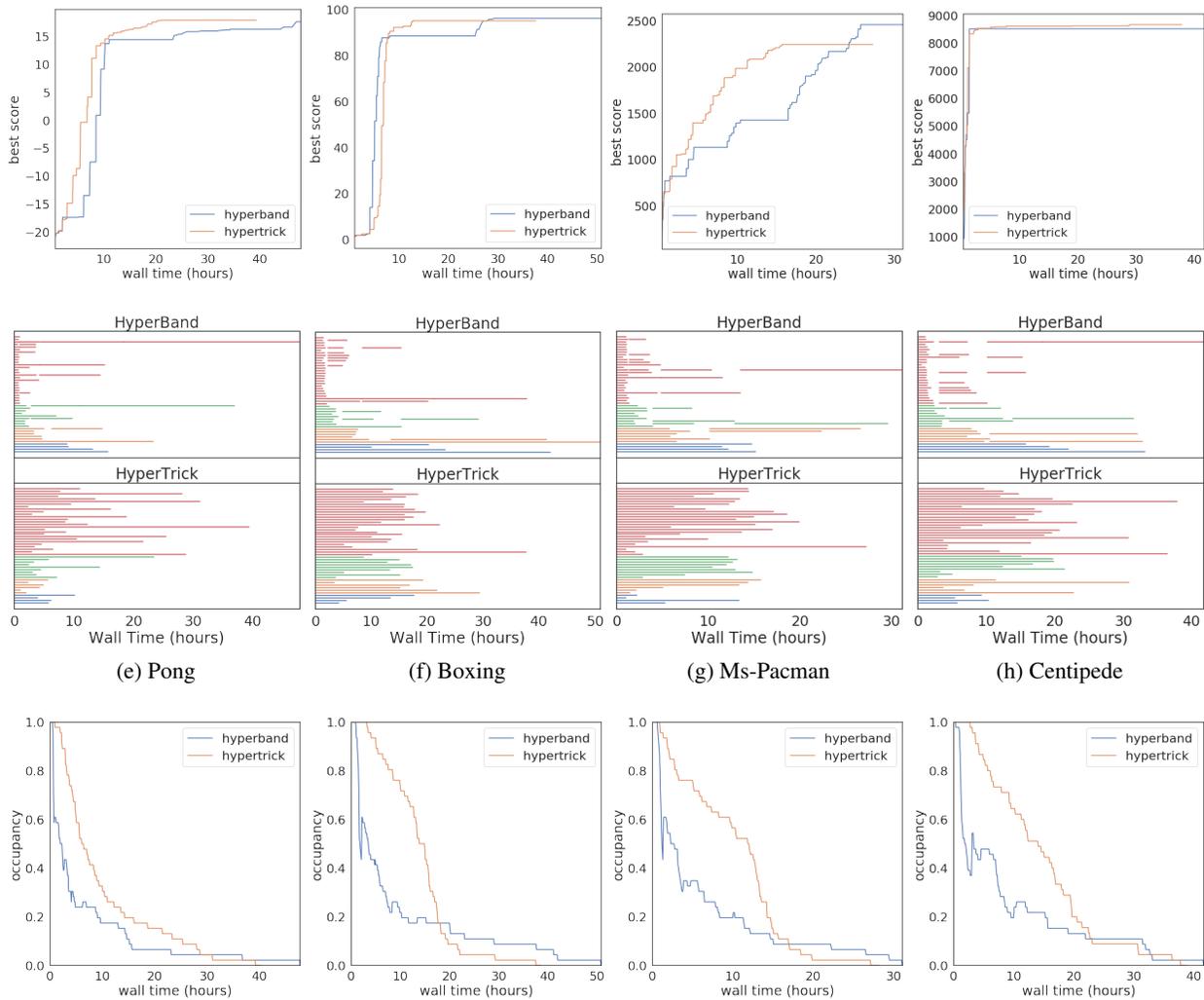


Figure 6: The first row shows the score of the best agent as a function of the wall time on four different Atari games for Hyperband and HyperTrick; the performances of the two metaoptimization algorithms are comparable in these graphs. The middle row shows the execution timelines for each worker; each color corresponds to a bracket of Hyperband - the same color is used in HyperTrick for an easy comparison. Workers may be idle in the Hyperband case because of the synchronization required at the end of each phase in Successive Halving. The last row shows the occupancy of the nodes of the distributed system as a function of the wall time; the occupancy is generally higher for HyperTrick.

policy, for the experimental setup described in Section 5.1. Fig. 7 shows the distribution of the scores as a function of learning rate, γ , and t_{max} . Since these scores are those reported by HyperTrick, not all the instances of GA3C run to completion.

The first row in Fig. 7 highlights the importance of selecting a proper learning rate: for instance, in the case of Pong, a learning rate in the $[1.5 \cdot 10^{-4}, 3 \cdot 10^{-3}]$ interval is needed to solve the game; the situation is similar, although with different numerical intervals, for Boxing and Ms Pacman, whereas for Centipede a scattered set of learning rates generate the best scores. As expected, agents learned with different learning rates do not show significant differences in the policy; the learning rate only affects the stability and rate of convergence towards the optimal policy.

The learning rate alone is clearly not sufficient to determine the success of RL training: the second row of Fig. 7 highlights the importance of the discount factor γ . A common choice is to set $\gamma = 0.99$ (Lillicrap et al., 2015; Mnih et al., 2016; Babaeizadeh et al., 2017), but γ values in a larger interval may be effectively employed for Pong and Boxing, whereas respectively smaller and larger γ values potentially lead to better results for Ms-Pacman and Centipede. Generally speaking, different games are learned at best for different intervals of γ values. This has been already noticed for instance in (François-Lavet et al., 2015), where γ is even modulated during training. It can be explained noticing that the temporal dynamics of the rewards are different in each game: for instance, a reward is immediately generated when the adversary scores a goal in Pong; when one of the two players hits the other one in Boxing; or when a pill is eaten in Ms-Pacman; these examples justify the adoption of small γ values for these games. On the other hand, rewards in Centipede are delayed from the moment in which the player fires and the instant in which a target is hit, which may justify the preference for larger γ values.

Quite remarkably, γ does have a significant effect on the learned policy. Agents that achieve similar scores on a game, but trained with different γ , adopt clearly different strategies. For instance, short-sighted (small γ) agents in Pong mostly learn to not lose the game, moving somewhat erratically to catch the ball; these agents tend to engage in long rallies, waiting for the opponent to commit a mistake. On the other hand, agents trained with a large γ learn to hit the ball on the edge of the racket, to give it a spin and effectively score a goal some frames later. In Boxing, short-sighted agents perform better as they kick the opponent as fast as possible, whereas looking for more long-term reward seems to merely give the opponent a chance to strike back. In Centipede, all agents learn that firing as fast as possible is desirable. High values of γ encourage agents to hide towards the edge of the screen where aliens are less likely

to attack, whereas slightly lower values of γ force agents to stick to the center of the screen, where they are more exposed to attacks. In Ms-Pacman, all the best agents are short-sighted ($\gamma < 0.9$); they learn to navigate the maze to eat close pills and escape from ghosts, but tend not to move towards far, isolated pills when few of them are left in the maze. Overall, metaoptimization can effectively identify an optimal value for γ , but the learned policy is affected by this choice. This interaction between one hyperparameter and the solution of the underneath optimization problem is easily justified in the context of RL, remembering that the discount factor γ affects the definition of optimality of the learned policy; it is anyway a peculiar aspect of metaoptimization on RL problems, that has to be taken into consideration by future researchers working in this direction.

The last hyperparameter analyzed here is t_{max} , which is set by default to $t_{max} = 5$ in A3C (Mnih et al., 2016). The third row in Fig. 7 suggests that higher values of t_{max} (but not smaller ones) can indeed lead to convergence, although it's not evident whether an optimal interval for this hyperparameter can be identified. Most likely, t_{max} affects at the same time the computational cost of the learning procedure, by changing the size of the batch, and the noise level of the updates, by affecting the bias-variance trade-off in the estimate of the value function. The relation between t_{max} and the outcome of the RL procedure is consequently complex, and probably influenced by the other hyperparameters - thus metaoptimization helps to automatically identify the best value of t_{max} in the absence of any other intuition.

The information stored by MagLev in the central knowledge database can also be used to perform *a posteriori* analyses that reveal more quantitative information about the RL procedure. An example of this is reported in the Appendix, where we show how to train a regressor to estimate the relation between the hyperparameters and the final score achieved by the RL procedure, and consequently quantify the contribution of each hyperparameter to the success of the RL training procedure.

6 CONCLUSION

HyperTrick, the asynchronous metaoptimization algorithm proposed here, is particularly suitable for the case of distributed systems, when the selection of the hyperparameters affect the computational cost of the underneath experiments. We demonstrate that HyperTrick allows effective metaoptimization for deep RL problems. HyperTrick does not require any complex synchronization mechanisms or pre-emption management: it frees and reallocates computational resources more efficiently than algorithms based on the Successive Halving principle, like HyperBand. When compared experimentally with those algorithms, HyperTrick achieves a higher occupancy of the nodes in the distributed system,

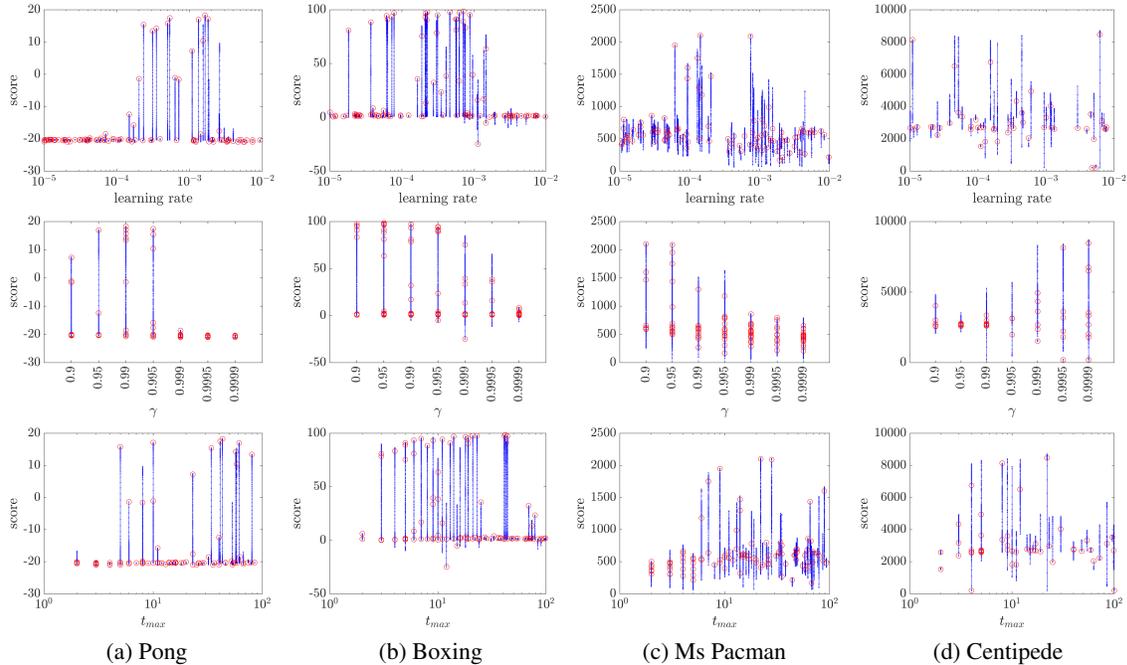


Figure 7: The final (red circles) and during training (blue lines) scores for 100 workers during metaoptimization through HyperTrick, as a function of the learning rate, discount γ , and t_{max} , for four Atari games learned with GA3C.

completes the metaoptimization procedure in a shorter time, and finds the optimal solution earlier. By adopting a stochastic process for the selection of the promising workers, HyperTrick gives early workers a higher chance to continue and increase the depth of their search, while late workers are discouraged. HyperTrick achieves in this way a partial balance between breadth and depth search in metaoptimization. A promising direction to achieve an even better balance is the integration of HyperTrick and Hyperband, where multiple instances of HyperTrick with different N_p and r may run in parallel. Furthermore, the additional resources released by HyperTrick may be employed to further improve the metaoptimization process, for instance by the integration of evolutionary strategies, *e.g.* by mixing the hyperparameters of fast learners, or reinitializing terminated agents with new sets of promising hyperparameters. We leave these and other possible improvements for future investigation. Our experiments finally highlight that, in the case of RL, hyperparameter selection and the learned policy can be connected, as in the case of the discounting factor γ - this is an additional source of complexity that has to be taken into account by future researchers working in this field.

REFERENCES

Kubernetes website. available at <https://kubernetes.io/>.

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., and Kautz, J. GA3C: GPU-based A3C for deep reinforcement learning. *NIPS Workshop*, 2016.

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., and Kautz, J. Reinforcement learning through asynchronous advantage actor-critic on a gpu. In *ICLR*, 2017.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

Buckman, J., Hafner, D., Tucker, G., Brevdo, E., and Lee, H. Sample-efficient reinforcement learning with stochastic ensemble value expansion. *CoRR*, abs/1807.01675, 2018.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018. URL <http://arxiv.org/abs/1802.01561>.

Farabet, C. Maglev: Efficient training for safe autonomous vehicles, 2018. URL <https://blogs.nvidia.com/blog/2018/09/13/how-maglev-speeds-autonomous-vehicles-to-superhuman-levels-of-safety>.

- François-Lavet, V., Fonteneau, R., and Ernst, D. How to discount deep reinforcement learning: Towards new dynamic strategies. *CoRR*, abs/1512.02011, 2015. URL <http://arxiv.org/abs/1512.02011>.
- Greg, B., Vicki, C., Ludwig, P., Jonas, S., John, S., Jie, T., and Wojciech, Z. Openai gym. 2016.
- Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., and Kavukcuoglu, K. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017. URL <http://arxiv.org/abs/1711.09846>.
- Jamieson, K. G. and Talwalkar, A. Non-stochastic best arm identification and hyperparameter optimization. In *AISTATS*, 2016.
- Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016. URL <http://arxiv.org/abs/1603.06560>.
- Li, L., Jamieson, K., Rostamizadeh, A., Gonina, K., Hardt, M., Recht, B., and Talwalkar, A. Massively parallel hyperparameter tuning. 2018.
- Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. 2015.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. URL <http://dx.doi.org/10.1038/nature14236>.
- Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.
- Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., Maria, A. D., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., and Silver, D. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015. URL <http://arxiv.org/abs/1507.04296>.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015. URL <http://arxiv.org/abs/1511.05952>.
- Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and de Freitas, N. Taking the human out of the loop: A review of bayesian optimization. In *IEEE*, 104(1): 148–175, 2016.
- Stooke, A. and Abbeel, P. Accelerated methods for deep reinforcement learning. *CoRR*, abs/1803.02811, 2018. URL <http://arxiv.org/abs/1803.02811>.
- Sutton, R. S. and Barto, A. G. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.
- Tieleman, T. and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4 (2), 2012.
- van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL <http://arxiv.org/abs/1509.06461>.
- Wang, Z., de Freitas, N., and Lanctot, M. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. URL <http://arxiv.org/abs/1511.06581>.
- Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

7 APPENDIX

7.1 Metaoptimization with Static Allocation of the Workers to the Nodes

For completeness, we report here simulations on the same metaoptimization toy problem introduced in Fig. 2, in the case of Successive Halving, with a static assignment of any worker to one node of the distributed system, and in the case of Grid Search (with no early stopping).

An implementation of Successive Halving with a static association between workers and nodes (Fig. 8) is possible, although a mechanism to manage preemption is needed even in this case, for two reasons: fast or early workers must stop and wait for all the other workers to complete a phase; in the case the number of nodes is smaller than the number of workers, the same node may have to run the same phase for more than one worker. At least for the toy problem considered here, such implementation is extremely inefficient in terms of time consumption, as it takes 15.3 units of time (to be compared against 12.1 units of time for Successive Halving with dynamic allocation of the workers to the nodes in Fig. 3, and 10 units of time for HyperTrick in Fig. 2).

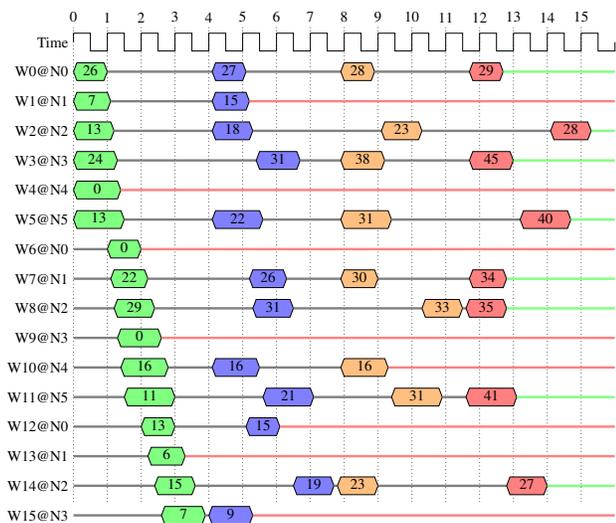


Figure 8: Metaoptimization with a variant of Successive Halving, which terminates 25% of workers at the end of every phase, for the same toy problem in Fig. 2. Each worker is statically assigned to a single node for the entire process. The process lasts for 15.3 units of time.

If preemption cannot be implemented, the metaoptimization scheme boils down to a full Grid Search, shown in Fig. 9. On the toy problem considered here, this is the slowest metaoptimization scheme (15.6 units of time). Since all the workers run the underneath optimization experiments to the end, the worker completion rate is in this case equal to $\alpha = 100\%$.

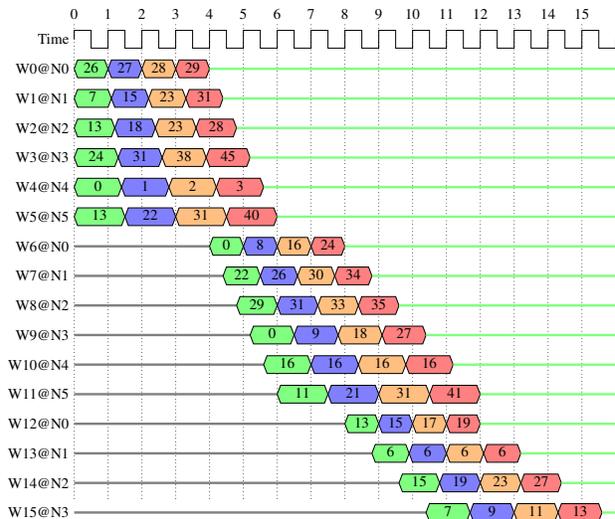


Figure 9: Metaoptimization with Grid Search (no early stopping) on the same toy problem in Fig. 2. No preemption mechanism is needed in this case. The process lasts 15.6 units of time.

7.2 Estimate the Importance of the Hyperparameters

The information stored by MagLev in the central knowledge database can be used to perform *a posteriori* analyses that reveal quantitative information about the RL procedure. An example is illustrated in the following, where we estimate the importance of the learning rate, γ and t_{max} to determine the final score of a game.

Game	learning rate	γ	t_{max}
Boxing	54%	24%	22%
Centipede	34%	35%	31%
Ms Pacman	43%	37%	20%
Pong	55%	31%	14%

Table 4: Importance of the learning rate, γ and t_{max} for every game, as estimated by a Random Forest regressor trained to map a hyperparameter configuration to a score.

Although learning an accurate mapping function between the hyperparameters and the final score is complex, we show that even an approximated function can provide valuable insights about the role played by each hyperparameter. To show this, we employ a Random Forest regressor trained to map a hyperparameter configuration to the score achieved in HyperTrick. Notice that this score is not necessarily the final score after completing all the phases, as a worker can be terminated early. For each game we use Scikit Learn (Pedregosa et al., 2011) to train 100 Random Forest regressors using various configurations of the Random Forest parameters; we use then 10-fold cross validation to identify the best, non overfitting, regressor. The feature importances for

each game, extracted through the Scikit API, are reported in Table 4.

Centipede arguably features the noisiest learning curves and unsurprisingly its regressor gives identical importance to all the hyperparameters. Conversely, for Pong γ and the learning rate dominate, and t_{max} appears to be less important. These results are in line with the intuition one can build from Fig. 5 and the observations reported in the main paper; they confirm the general small influence of t_{max} on the final score, and the importance of a proper selection of the learning rate.