UNDERSTANDING OPTIMIZATION *in* REINFORCEMENT LEARNING

An Empirical Study of
Algorithms and their Hyperparameters

Jan Ole von Hartz

May 2019

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science

to the

Machine Learning Lab
Department of Computer Science
Technical Faculty
University of Freiburg

Work Period

28. 02. 2019 – 28. 05. 2019

Examiner

Prof. Dr. Frank Hutter

Supervisor

Raghu Rajan

Contact

hartzj@cs.uni-freiburg.de

DECLARATION

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids,other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Place, date	Signature

This thesis was formatted using the incredible template created and released into the public domain by Eivind Uggedal, found at github.com/uggedal/thesis.

Hereafter follow the original remarks to his thesis.

This thesis was typeset using the LaTeX typesetting system originally developed by Leslie Lamport, based on TeX created by Donald Knuth.

The body text is set 12/14.5pt on a 26pc measure with Minion Pro designed by Robert Slimbach.
This neohumanistic font was first issued by Adobe Systems in 1989 and have since been revised.
Other fonts include Sans and Typewriter from Donald Knuth's Computer Modern family.

Typographical decisions were based on the recommendations given in *The Elements of Typographic Style* by Bringhurst (2004).

The use of sidenotes instead of footnotes and figures spanning both the textblock and fore-edge margin was inspired by *Beautiful Evidence* by Tufte (2006).

The guidelines found in *The Visual Display of Quantitative Information* by Tufte (2001) were followed when creating diagrams and tables.
Colors used in diagrams and figures were inspired by the *Summer Fields* color scheme found at
http://www.colourlovers.com/palette/399372

ABSTRACT

In machine learning, praxis is by far more delicate than theory: not just different algorithms and random seeds, but also the usage of different optimizers and settings of their hyperparameters yield dramatically different results, to the point of convergence versus non-convergence. Deep reinforcement learning is especially difficult due to the strongly moving loss landscape, with agents not learning the desired behavior because of getting stuck in local optima. While there exists some rough ideas, the exact effects of different optimizers (such as Adam) and their hyperparameters on the returns of the agent, as well as the stability in training and evaluation, are not yet well understood. Using the well known DQN algorithm for discrete and DDPG algorithm for continuous environments, we investigate the influence of different optimizers and settings of their hyperparameters on the agents performance, as well as compare the optimizers' sensitivity to hyperparameters and their stability in training. Using BOBH, a hyperparameter optimization method, we further try to make a step towards hyperparameter agnostic deep reinforcement learning, to try to avoid the pitfalls of hyperparameter settings.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

First and foremost, I would like to thank my parents for their generous and ongoing financial support. Without them, my studies, including this thesis, would not have been possible in the form they were.

The authors acknowledge support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG (bwForCluster NEMO).

TODO

Jan Ole von Hartz
Freiburg, Germany
May 2019

1 INTRODUCTION

According to Sutton and Barto (2018, p. 1) , the field of *reinforcement learning* studies learning "[...] how to map situations to actions—so as to maximize a numerical reward signal". Following the authors, it refers - besides the field of study - to both the problem class itself, as well as a class of solution methods.

The commonly used formalism is that of an *agent* interacting with an *environment* via a fixed set of *actions* and only observing 1. a numerical reward signal and 2. the state of the environment, or an observation which describes part of it. Figure 1.1 summarizes this interaction.

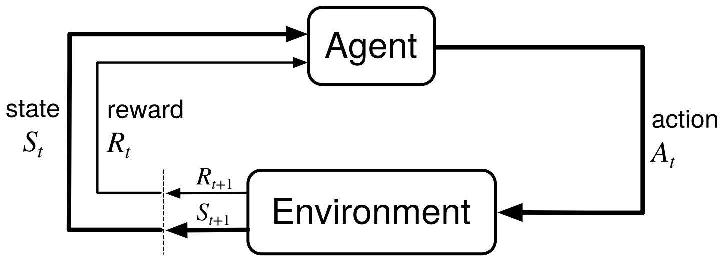


Figure 1.1: The reinforcement learning loop (Sutton and Barto, 2018, Figure 3.1). The agent observes the reward and current state of from the environment and selects an action accordingly, which in turn alters the state of the environment.

1.1 FOCUS

In this thesis, we focus on the analysis of adaptive gradient-based optimization methods (Adam and AdamW) in deep reinforcement learning using well-established value-based (DQN) and actor-critic (DDPG) methods in a popular and computationally simple control environment (Cartpole), a popular and more complex set of video game environments (Atari Arcade Environment) and a set of more challenging physical control tasks (Mujoco)[1]. We deem it important that our groundwork is generalized to more optimization methods, learning algorithms and environments, but we had to limit the scope of this thesis due to constraints on the volume of work.

1. For an introduction of and reference to all these methods and environments, see chapter Chapter 4 (p. 13).

1.2 MOTIVATION

Recent Progress In recent years, *reinforcement learning* has not only seen a surge of research, but has also produced a number of remarkable results, such as agents learning to control a helicopter (Ng et al., 2006), locomotion tasks (Heess et al., 2017), in-hand manipulation of objects (OpenAI et al., 2018) and playing games[2] such backgammon (Tesauro, 1995), Atari video games (Mnih et al., 2013) and Go (Silver et al., 2016, 2017). Just during the creation of this thesis, a team of agents trained via reinforcement learning won a tournament against the 2018 world champions in the complex multiplayer online battle arena computer game DOTA 2 (Statt, 2019). All this is in part due to the increasing application of so called *deep reinforcement learning* techniques, the use of *deep neural networks* in reinforcement learning.

2. Often on what is called a "superhuman" level, generating a lot of media coverage for the field.

New Problems While allowing for great scientific progress, the use of deep neural networks also introduces many new and opaque problems, some of which we try to tackle in this thesis. For one, solving machine learning problems usually involves the selection of a as well suited as possible function from a class of functions by minimizing some loss function defined over the output of said function for a set of data points. Hence, the training of a deep neural network has in its core a *numerical optimization problem*: finding a suited parametrization of the network to achieve the best possible performance (the lowest possible loss) on unseen data by using a given data set as an approximation of the general data distribution and optimizing the networks performance on it. This is usually done via *gradient-based methods*, such *gradient descent* and variations of it.

Trial-and-Error Reinforcement learning however introduces additional problems; unlike in *supervised learning*, the training data is not given, but must be produced by the agent by engaging with the environment in a trial-and-error fashion. Since the deep neural networks are used by the agent to e.g. approximate a function that estimates the reward of a certain action in a certain state and these estimations change drastically over time, the numerical optimization of these networks is very unstable.

Local Optima A systematic problem of gradient-based methods is that unlike analytical methods they are generally only suited to find local optima,[3] not global optima. In reinforcement learning the loss landscape is usually very rough and provides plenty of local optima to get stuck in. An illustrative example was presented by Henderson et al. (2018a): in an environment, in which an agent was supposed to learn a swimming motion, curling up constituted a local optimum, in which one agent would get stuck.[4]

3. Global optima are only found for convex functions.

4. See youtu.be/lKpUQYjgm8o

Methods and Hyperparameters Furthermore, it is not well understood, how different (adaptive) gradient-based methods, such as the popular Adam algorithm and AdamW, a variation of it, compare in reinforcement learning, especially since they are also somewhat sensitive to the settings of their hyperparameters. A common call thus is for the development of hyperparameter agnostic algorithms - algorithms that adapt their hyperparameters themselves during runtime (Henderson et al., 2018a).

Performance Estimation The automatic hyperparameter optimization introduces further challenges, since it requires for a cost-effective estimation of the performance of a given configuration - something that is not given in reinforcement learning, where the training of an agent is usually very costly.[5]

5. Agents are most often trained on millions of simulated time steps using large computer clusters.

1.3 OBJECTIVE

Performance Estimation We will shine some light on these opaque problems of optimization in deep reinforcement learning by first investigating ways of reliable and cost-effective performance estimation of reinforcement learning agents as a foundation.

Comparison of Optimization Methods We will then compare different popular optimization methods in reinforcement learning to evaluate their respective performance and gain an understanding of their different characteristics, that is 1. the final performance of their produced network configuration[6] 2. the stability of these optima[7] 3. their sensitivity towards their hyperparameters and 4. their stability in training as measured by their inner state.

6. Behold the problem of **local optima** in reinforcement learning.

7. Connecting back to the problem of instable optimization due to **trial-and-error** training.

Towards Hyperparameter Agnosticism Using a hyperparameter optimization method we will automatically find well performing hyperparameter settings for the optimizers to make a step towards hyperparameter agnosticism and evaluate the practicality of this approach.

1.4 CONTRIBUTION

Cost-effective performance estimation is not only a preliminary for automatic hyperparameter setting, but will also help researchers and practitioners, especially with low computational budgets, to make early informed decisions and efficiently use their resources.

The analysis of the different optimization methods does not only deepen our understanding of the behavior of adaptive gradient-based methods in the strongly changing loss landscapes of deep reinforcement learning, but also helps with the selection of suitable algorithms and their hyperparameters.

The automatic optimization of these hyperparameters will help to advance towards the goal of a fully closed machine learning pipeline.

Generally, our results will help practitioners and researchers to focus on their core interest without the large burden that comes with the selection of optimization methods and their hyperparameters.

1.5 OUTLINE

This thesis consists of three parts.

The first one introduces necessary formalisms, concepts and establishes a consistent notation. It will do so in a logical order; starting with notation and basic theoretical notions, continuing with the realization and use of these concepts in the used algorithms, and closing with the tangible tools and computational libraries that were used.[8]

8. This approach is motivated in the beginning of chapter Chapter 5 (p. 29); Henderson et al. (2018a) showed the huge influence of the code base on the final performance in deep reinforcement learning.

The second part presents the problem of performance estimation in deep reinforcement learning and the experiments we conducted.

The third part presents our comparison of the different optimizers in a practical reinforcement learning environment.

Afterwards, we will conclude our results and point out which future works needs to be done.

PART I

BACKGROUND

KEY CONCEPTS AND NOTATION

2

2.1 NOTATION

Whenever possible we distinguish between

- scalars: x
- vectors: $\vec{x}$
- matrices: X
- sets: $\mathcal{X}$
- (time) series $(\chi_n)_{n=0,1,\dots}$

For a vector $\vec{x}$ the i-th element is denoted as x_i. For a matrix X, $\vec{x}_i$ denotes the i-th column and x_{ij} the j-th value of this column. The value of a time series $(\chi_n)_{n=0,1,\dots}$ at point t is denoted as χ_t.

2.2 KEY CONCEPTS

To better understand all details addressed in this thesis, we recommend the gentle reader to be familiar with some key concepts of *deep learning* - machine learning using *deep neural networks*. The gist however, should be accessible to most readers with a basic mathematical background. We will next briefly reiterate the most important concepts to accomplish a common ground in notation. Unfamiliar readers can revise these concepts, e.g. with Goodfellow et al. (2016). Proficient readers can safely skip this chapter. Necessary concepts from *reinforcement learning* and *deep reinforcement learning* will be introduced in the next chapter. Debutants can deepen their understanding, e.g. in Sutton and Barto (2018).

Supervised Learning Given some function class $\mathcal{G} : \mathbf{R}^d \to \mathbf{R}^o$ and a matrix of training data $X \in \mathbf{R}^{d \times n}$ with targets $Y \in \mathbf{R}^{\mathbf{o} \times \mathbf{n}}$, that are sampled from some generating data distribution X_{Gen}, Y_{Gen}, an algorithm tries to find the function $g \in \mathcal{G}$ that best describes the relation of data points $\vec{x}_i \in X_{Gen}$ with the matching targets $\vec{y}_i \in Y_{Gen}$. It does so by using the training data and targets as an approximation of the general data distribution and minimizing some **loss function** $L : \mathbf{R}^{o \times 2} \to \mathbf{R}$ on the set of pairs of predictions $g(\vec{x}_i)$ on the training data, and training targets $\vec{y}_i$. One commonly used loss functions is the **mean squared**

error $L(\vec{x}_j, \vec{y}_j) = \frac{1}{n}\sum_{i=0}^{n}(x_{ij} - y_{ij})^2$, where $n = |a| = |b|$. The learning is usually done by defining $\mathcal{G}$ as a set of functions $g_{\vec{\theta}}$ with parameters $\vec{\theta}$ and numerically optimizing these parameters.

(Mini-) Batch Since the computation of the loss on the whole available training data at once is not only costly but also not necessarily effective (Keskar et al., 2016), while the computation on a single data point leads to very noisy estimates of the gradients in numerical optimization, a common approach is to divide the training data into, or to sample *(mini-) batches* from, the training data and compute the numerical updates on these instead of on the full data set.

Overfitting is what happens when a machine learning model fits the training distributions more closely than justified by the underlying generating data distribution, leading to a worse generalization performance.

Gradient Descent is a numerical optimization method, suited to find local minima of some differentiable function $f : \mathbf{R}^n \to \mathbf{R}$ and global minima if f is *convex*[1]. Starting at some point $\vec{x}_0$ and with some *learning rate* α, gradient descent iteratively computes updates $\vec{x}_t = \vec{x}_{t-1} - \alpha \cdot \nabla f(\vec{x}_{t-1})$ until some stopping criterion (e.g. convergence) is reached.

1. Convexity is usually not given, but local optima give decent approximations and might overfit less.

Stochastic Gradient Descent is a stochastic approximation of gradient descent, commonly applied in machine learning. The gradients of the loss function for the parameter updates are computed on a randomly sampled minibatch, often leading to faster convergence.

(Deep) Neural Networks are non-linear function approximators commonly used in recent machine learning research. They are constructed by stacking layers of nodes (neurons) that each 1. compute some linear combination of the values of the nodes of the previous layer and 2. apply some non-linear activation function. The parameters of the network (the weights and biases of the linear functions) are denoted by $\vec{\theta}$. A prediction for some data vector is made by feeding the latter into the first layer (the input layer) of the network and computing the activation of the last layer (the output layer). This is called the *forward pass*. The value of the output layer of the network f with parameters $\vec{\theta}$ for some input $\vec{x}$ is denoted as $f(\vec{x}, \vec{\theta})$. Supervised training of the network is commonly achieved by calculating the average derivative $\frac{1}{n}\sum_{i=0}^{n}\nabla_{\vec{\theta}}L(\vec{y}_i, f(\vec{x}_i, \vec{\theta}))$ of some loss function L, with respect to the network parameters $\vec{\theta}$ on some (mini-) batch X with size n and targets Y (called *backward pass*), and applying updates to $\vec{\theta}$ via *(stochastic) gradient descent*.

Observability In artificial intelligence research, an environment is called *fully observable*, if an agent that interacts with it, has full access to all relevant information about its inner state, otherwise *partly observable*.

3 REINFORCEMENT LEARNING

As introduced, problems in reinforcement learning are usually framed as an *agent* interacting with an *environment* via a fixed set of *actions* and only observing 1. a numerical reward signal and 2. the state of the environment or an observation which describes part of it. This chapter introduces the mathematical formalisms and the theoretical ideas behind their practical application.

3.1 THE MATHEMATICAL MODEL

In large parts of the current research, as well as in this thesis, instances of this problem class are formalized using *Markov decision processes* (Howard, 1960). Hence, we now define this and related notions.

3.1.1 *Markov Decision Processes and Policies*

Definition 1 (Markov Process) *Let $\tau = 0, 1, \dots$ be a (possibly infinite) series of discrete time steps and $t \in \tau$. A* Markov process [1] *is a 2-tuple $(\mathcal{S}, \mathcal{P})$, where $\mathcal{S}$ is a finite* [2] *set of states and $\mathcal{P}$ is a state transition function $\mathcal{P}(s, s') = \mathbf{P}[s_t = s' \mid s_{t-1} = s]$, that fulfills the* Markov property, *i.e. for all states and at all times $\mathbf{P}[s_t \mid s_{t-1}, s_{t-2}, \dots, s_0] = \mathbf{P}[s_t \mid s_{t-1}]$.* [3]

1. *Also often called a* Markov chain.
2. *We do not need the notion of infinite Markov processes here.*
3. *Intuitively: "the future is independent of the past, given the present".*

Definition 2 (Markov Decision Process) *A* Markov decision process *is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $(\mathcal{S}, \mathcal{P})$ is a* Markov process *with transition function $\mathcal{P}_a(s, s') = \mathbf{P}[s_t = s' \mid s_{t-1} = s, a_{t-1} = a]$ (where $a \in \mathcal{A}$), $\mathcal{A}$ is a finite set of actions, $\gamma \in [0, 1]$ is a discount factor and $\mathcal{R}$ is the expected immediate reward $\mathcal{R}_a(s, s') = \mathsf{E}[R(s_{t-1}, a_{t-1}) \mid s_{t-1} = s, s_t = s', a_{t-1} = a]$, with R being the reward received at point t, defined as a function mapping s_{t-1} and a_{t-1} to some real number.* [4]

4. *Also often defined as a function of only s_{t-1} or even s_{t-1}, a_{t-1} and s_t, depending on the problem at hand.*

Definition 3 (Policy) *A* policy *π is a mapping from a state space $\mathcal{S}$ to a set of probability distributions over an action space $\mathcal{A}$, given some state $s \in \mathcal{S}$: $s \mapsto p(a \mid s)$.*

Reinforcement learning problems are then often modeled as interpreting a *Markov decision process M* as an environment in which an Agent A can take actions $a \in \mathcal{A}_M$ and observe the ensuing reward $r \in \mathbf{R}$ and state $s \in \mathcal{S}_{\mathcal{M}}$. The agent's goal then usually is to find an optimal policy $\pi^\star$ to maximize some cumulative function, usually the total discounted reward $\mathrm{R} = \sum_{t=0}^{h-1} \gamma^t \cdot R(s_t, a_t)$, where $h \in \mathbf{N} \cup \{\infty\}$ is called the *horizon* of the problem. [5]

5. The horizon can be infinite.

Definition 4 (Trajectory) *Let $h \in \mathbf{N} \cup \{\infty\}$ be a (possibly infinite) Horizon, $\tau = 0, 1, \dots, H$ a series of discrete time steps, $\mathcal{A}$ a set of actions, $\mathcal{S}$ a set of states and $\mathcal{R} \subseteq \mathbf{R}$ a set of reward signals. A trajectory then is a series $\rho = s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_{h-2}, a_{h-2}, r_{h-2}, s_{h-1}$, with $s_i \in \mathcal{S}$, $a_i \in \mathcal{A}$ and $r_i \in \mathcal{R}$ for all $i \in \tau$. A trajectory is called infinite, if h is infinite.* [6]

6. Practically we will only concern ourselves with finite trajectories.

The notion of the trajectory is literally used in reinforcement learning to describe the interaction of an agent with an environment over time.

Another related notion is that of the ***episode***. As the trajectory of an agent in an environment is often closed by the environment reaching some terminal state [7], an *episode* describes a trajectory from an initial state to a terminal state.

7. E.g. "game over" in video games, which are popular training grounds in reinforcement learning

3.1.2 *Value Functions and the Bellman Equation*

Notably, a policy implies a transition distribution over the state space of a Markov decision process M. This property allows to easily evaluate a given policy π on M, by reducing the problem of evaluation to the evaluation of a Markov process, like the following definition illustrates.

Definition 5 (State-Value Function) *The* state-value function *(often called V-function) $V^\pi(s)$ of a state $s \in \mathcal{S}$, given a policy π, is the expected return gained when starting in s and following π henceforth:*

$$V^\pi(s) := \mathsf{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t \cdot R(s_t, a_t) \mid s_0 = s \right]$$

.

Definition 6 (State-Action-Value Function) *The* state-action-value function *(often called Q-function) $Q^\pi(s, a)$ of a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, given a policy π, is the expected return gained when starting in s, taking action a and following π thereafter:*

$$Q^\pi(s, a) := \mathsf{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t \cdot R(s_t, a_t) \mid s_0 = s, a_0 = a \right]$$

.

Theorem 1 (Bellmann Equation) *The* state-action-value function *fulfills the following recursive property:* [8]

8. A similar property holds for the action-value function.

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s, s') \sum_{a' \in \mathcal{A}} \pi(a' \mid s') \cdot Q^\pi(s', a')$$

Value Iteration This famous property allows to, given the reward function R, iteratively compute the Q-function or V-function for a finite state space, a finite action space and a given policy. Thereto, one solely has to initialize all values as zero and then repeatedly apply the Bellman equation until the values converge (Bellman, 1957).

Policy Iteration Remember, that the goal for the agent is to learn the optimal policy $\pi^\star$, which corresponds to an optimal *state-action-value function* $Q^\star$. The optimal policy $\pi^\star$ is the one that maximizes the corresponding Q-function $Q^\star$. As the Bellman equation illustrates, maximizing Q under policy π is achieved by greedily selecting the action a in state s that maximizes

$$R(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s,s') \sum_{a' \in \mathcal{A}} \pi(a' \mid s') \cdot Q^\pi(s',a')$$

yielding an improved policy π':

$$\pi'(s,a) := \begin{cases} 1 & \text{, if } a = \arg\max_{\hat{a} \in \mathcal{A}} Q(s,\hat{a}) \\ 0 & , else \end{cases}$$

Hence, just as $Q^\star$ can iteratively be computed, so can $\pi^\star$ be as well, e.g. via *dynamic programming* (Howard, 1960).

3.2 APPLICATION OF THE MODEL

These lessons learned in the exact solution of the optimal policy via dynamic programming, can now also be applied to approximately solving this problem, as usually done in current reinforcement learning research. For this, we present two approaches that the algorithms introduced later make use of: *value base methods* and *actor-critic methods*.

3.2.1 *Value Based and Actor-Critic Methods*

Valued Based Methods To learn an approximation to the optimal policy $\pi^\star$ in an environment M, an agent A can thus estimate $Q^\star$ via arbitrarily initializing its estimate $\hat{Q}$ and then, using the reward signals provided by the environment in response to its actions, iteratively improve its estimates of $Q^\star$ via *value iteration*.[9] This technique is at the heart of so called *value based methods*, such as *Q-learning* (Watkins and Dayan, 1992).

9. In order to get a decent estimate of $Q^\star$, the agent has to engage in a trade-off between taking the greedily selected action and exploring new actions, as will be discussed in § 4.2.1 (p. 18).

Actor-Critic Methods As the selection of the policy often happens in a greedy manner,[10] all possible actions must be considered, which is only feasible for action spaces that are discrete and finite. For other action spaces, agents can, for example, separately estimate both the optimal V-function and the optimal policy (instead of producing the current policy via greedy selection from the current estimation of Q), which is at the heart of so called *actor-critic methods*.

10. Greedy selection connotes selecting the option that at the current point in time has to best value.

Other Methods Another issue with continuous action spaces is that the run time of the iterative update of the Q-function is at least linearly dependent on the size of the action space (see Theorem 1 (p. 10)) . This can be solved by approximating the updates of the Q-function via *Monte Carlo sampling* from the action space, which we will not discuss. Another common approach is to artificially discretize a continuous action space.

3.2.2 *Practicality of the Markov Property*

As noted by Arulkumaran et al. (2017), the underlying assumption of the *Markov Property* does often not hold in practical applications, since it requires full observability of the states. While algorithms that use *partly observable Markov decision processes* exist, full observability is often simply assumed as an approximation of the actual partly observable environment. An example of such an approximation can be found in Mnih et al. (2013): using the video output of an Atari game simulator, the authors preprocessed the observations by combining four consecutive video frames into a single observation to approximate state features such as the velocity of in-game objects.

3.3 SCOPE OF THIS THESIS

In this thesis we will examine *deep reinforcement learning* algorithms based on *value based methods* and *actor-critic methods*.

We will now introduce the used algorithms and explain how they relate to the introduced formalisms and methods in order of increasing abstraction level.

4 ALGORITHMS AND METHODS

4.1 NUMERICAL OPTIMIZATION

Deep reinforcement learning usually involves the *numerical optimization* of one or more *deep neural networks* as function approximators with regards to some loss function. Numerical optimization in deep reinforcement learning usually makes use of *gradient-based methods*, since gradients can be easily computed for deep neural networks, while analytical solutions are infeasible and higher order moments very expensive to calculate. However, naive *stochastic gradient descent* is generally not the preferred method, as more advanced first-order methods that estimate higher order moments of the target function can yield significantly faster convergence without adding a lot of computational burden. We will now introduce the used optimization methods.

4.1.1 *Adam: Adaptive Moment Estimation*

Method

Adam (Kingma and Ba, 2014) is a first-order gradient-base optimization method that "[...] computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients" (Kingma and Ba, 2014, p. 1). It does so by calculating exponential moving averages of the gradient and squared gradient of each parameter and using them to determine the individual learning rates. The individual steps of Adam can be seen in Algorithm 1 (p. 14). Until reaching some stopping criterion (e.g. convergence), in each time step Adam computes the current gradients, which in machine learning is usually done on some minibatch, updates its moment estimates and uses them to update the parameters it optimizes.

Popularity

Kingma and Ba (2014) have not only shown theoretical convergence properties of Adam, but also demonstrated for a range of machine learning tasks, including deep neural networks, that Adam performs en par with - or even outperforms - similar methods, such as RMSProp (Tieleman and Hinton, 2012) in terms of convergence of the training cost. Adam has hence quickly gained in popularity. According to a survey

```
**Hyperparameters:** step size $\alpha \in \mathbf{R}$, decay rates $\beta_1, \beta_2 \in \mathbf{R}$,
                     division stabilization constant $\epsilon > 0$
**Given:** initial parameter vector $\vec{\theta}_0 \in \mathbf{R}$,
           objective function series $\left(f_n\left(\cdot, \vec{\theta}\right)\right)_{n=0,1,\dots}$
**Optional:** schedule multiplier series $(\eta_n)_{n=0,1,\dots}$
**Init:** $t \leftarrow 0$; $\vec{m}_0, \vec{v}_0 \leftarrow \vec{0}$;  (time step and moment vectors)
1  while stopping criterion not met do
2      $t \leftarrow t + 1$;
3      $\vec{g}_t \leftarrow \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1})$;       (computed on minibatch $\vec{m}$)
4      $\vec{m}_t \leftarrow \beta_1 \cdot \vec{m}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t$;       (moment estimation)
5      $\vec{v}_t \leftarrow \beta_2 \cdot \vec{v}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t^2$;
6      $\hat{\vec{m}}_t \leftarrow \vec{m}_t / (1 - \beta_1^t)$;                 (zero-bias correction)
7      $\hat{\vec{v}}_t \leftarrow \vec{v}_t / (1 - \beta_2^t)$;
8      $\vec{\theta}_t \leftarrow \vec{\theta}_{t-1} - \eta_t \cdot \alpha \cdot \hat{\vec{m}}_t / (\sqrt{\hat{\vec{v}}_t} + \epsilon)$;     (parameter update)
9  end
10 return $\theta_t$
```

Algorithm 1: Adam. Until some predefined stopping criterion (e.g. convergence) is met, the algorithm repeats the following steps; computation of the gradients on a minibatch, update of the moment estimates and bias-correction, update of the parameters. The bias correction is introduced due to the zero-initialization of the moment estimates, see Kingma and Ba (2014). Although not proposed by Kingma and Ba (2014), Adam can also be combined with learning rate scheduling, see e.g. Loshchilov and Hutter (2017), introducing an additional factor η_t in the parameter update.

conducted by Karpathy (2017) of 28,303 machine learning papers published on arxiv.org between 2012 and 2017, Adam was used in about one of four papers, making it the most popular optimization method by far.

Limitations

In contrast to its advantages and popularity, Wilson et al. (2017) found that Adam and other adaptive methods tend to generalize worse than stochastic gradient descent and provided examples on which adaptivity leads to overfitting. E.g. Reddi et al. (2018) have since proposed extensions of Adam to combat its shortcomings.

Open Questions

Despite these ongoing developments, the exact properties of Adam and the influence of its hyperparameters are not yet well understood. Heusel et al. (2017, p. 4) sow the seeds by investigating Adam for *generative adversarial networks* and characterizing it as a "heavy ball with friction"

- preferring flat minima over steeper ones. Henderson et al. (2018b, p. 1) compared different optimizers in a reinforcement learning setting and found "[...] that adaptive optimizers have a narrow window of effective learning rates, diverging in other cases, and that the effectiveness of momentum varies depending on the properties of the environment". Figure 4.1 illustrates this fact; the performance is very sensitive towards the learning rate, the exact effect however depends on the agent and environment.

These findings seem contradictory to us, as flat minima are typically thought to be more robust than steeper ones (Hochreiter and Schmidhuber, 1997; Keskar et al., 2016) and Adam is advertised to be more robust against settings of its learning rate exactly due to the individual adaptation (Kingma and Ba, 2014). Dinh et al. (2017) also challenged the idea of flat minima generalizing better.

Also, recall from § 1.2 (p. 2) the problem of local minima in which agents could get stuck, not learning the desired behavior. As the relation between the deployed optimizer and the found minima is not well understood yet, so is the relation between the optimizer and the learned behavior of the agent.

We therefore want to continue in this line of research by investigating Adam's detailed behavior in deep reinforcement learning and explore whether hyperparameter optimization methods can solve the problems found by Henderson et al. (2018b).

Loshchilov and Hutter (2017) found another reason for the poor generalization of Adam. In practice gradient descent is often used in conjunction with *L_2-regularization* or *weight decay* (Krogh and Hertz, 1992), two popular methods to regularize network parameters to avoid overfitting. While both are equivalent for stochastic gradient descent, they are not for Adam. And while naive implementations often just apply L_2-regularization, Adam benefits far more from weight decay.

4.1.2 AdamW: Adam with Weight Decay

Method

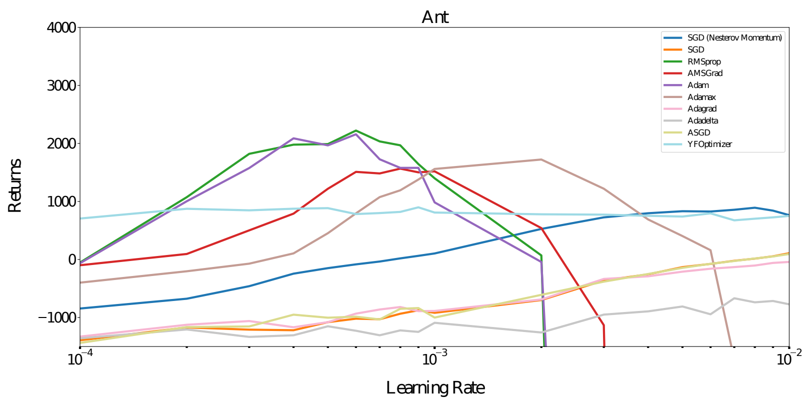
While in vanilla stochastic gradient descent the update rule of the optimized parameters $\vec{\theta}$ with learning rate α for time step t on minibatch $\vec{m}$ is defined as

$$\vec{\theta}_t = \vec{\theta}_{t-1} - \alpha \cdot \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1}) \tag{4.1}$$

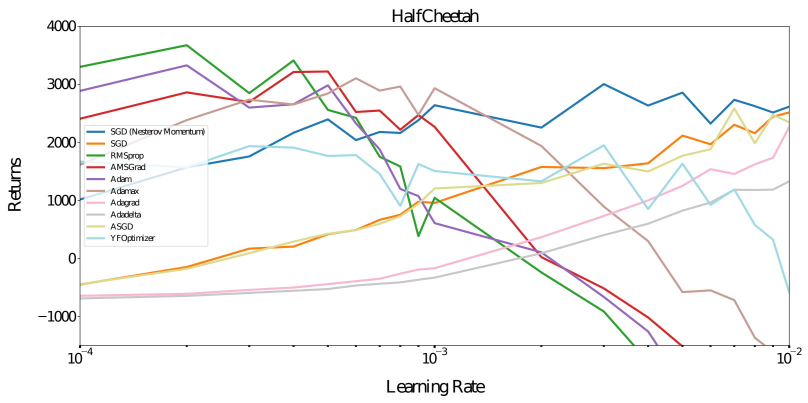
weight decay introduces an additional factor λ that with time exponentially decays the parameter values to regularize their growth. We follow the notation of Loshchilov and Hutter (2017) here:

$$\vec{\theta}_t = (1 - \lambda) \cdot \vec{\theta}_{t-1} - \alpha \cdot \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1}) \tag{4.2}$$

Combining this technique with the Adam algorithm (see algorithm Algorithm 1 (p. 14)), gives algorithm Algorithm 2 (p. 17), called *AdamW*



(a) A2C (actor-critic) agent in Ant environment.



(b) TRPO agent in Half-Cheetah environment.

Figure 4.1: Final performance of two different agents in different environments per setting of the learning rate for different optimizers (Henderson et al., 2018b, Figure 1). Note, how the performance for Adam (purple line) strongly peaks around an optimal setting, while falling apart for divergent settings.

by Loshchilov and Hutter (2017). The authors gave proof that while for stochastic gradient descent L_2-regularization and weight decay are equivalent - up to rescaling of the learning rate - this is not the case for Adam. This can easily be seen intuitively by changing line 3 in algorithm Algorithm 1 (p. 14) to

$$\vec{g}_t \leftarrow \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1}) + \boxed{\lambda \cdot \theta_{t-1}} \tag{4.3}$$

as would to the case for L_2-regularization. Since the gradient is used afterwards for the moment estimation, $\lambda \cdot \theta_{t-1}$ has a non-linear influence on the parameter update. For a formal proof that L_2-regularization and weight decay are not equivalent in the case of Adam see Loshchilov and Hutter (2017).

Properties

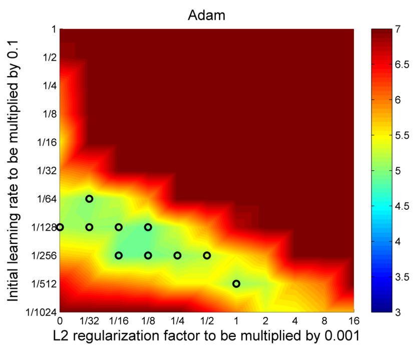
Generalization The authors conducted experiments on a number of tasks in a *supervised* machine learning setting and found that the usage of weight decay drastically improves Adams generalization performance,

```
Hyperparameters: step size $\alpha \in \mathbf{R}$, decay rates $\beta_1, \beta_2 \in \mathbf{R}$,
                 division stabilization constant $\epsilon > 0$,
                 weight decay factor $\lambda \in [0, 1)$
Given: initial parameter vector $\vec{\theta}_0 \in \mathbf{R}$,
       objective function series $\left(f_n\left(\cdot, \vec{\theta}\right)\right)_{n=0,1,\ldots}$
Optional: schedule multiplier series $(\eta_n)_{n=0,1,\ldots}$
Init: $t \leftarrow 0$; $\vec{m}_0, \vec{v}_0 \leftarrow \vec{0}$ ; (time step and moment vectors)
1  while stopping criterion not met do
2      $t \leftarrow t + 1$;
3      $\vec{g}_t \leftarrow \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1})$ ;        (computed on minibatch $\vec{m}$)
4      $\vec{m}_t \leftarrow \beta_1 \cdot \vec{m}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t$;        (moment estimation)
5      $\vec{v}_t \leftarrow \beta_2 \cdot \vec{v}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t^2$;
6      $\hat{\vec{m}}_t \leftarrow \vec{m}_t / (1 - \beta_1^t)$;             (zero-bias correction)
7      $\hat{\vec{v}}_t \leftarrow \vec{v}_t / (1 - \beta_2^t)$;
8      $\vec{\theta}_t \leftarrow \boxed{(1 - \lambda)} \cdot \vec{\theta}_{t-1} - \eta_t \cdot \alpha \cdot \hat{\vec{m}}_t / (\sqrt{\hat{\vec{v}}_t} + \epsilon)$;       (parameter
        update)
9  end
10 return $\theta_t$
```

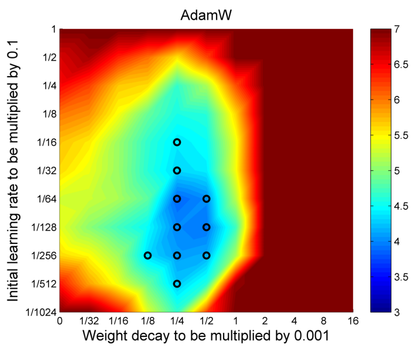
Algorithm 2: AdamW. Yielded by combining Adam (see algorithm Algorithm 1 (p. 14)) with weight decay (see Equation 4.2 (p. 15)).

while still allowing the further improvements via the use of *scheduled learning rate multipliers*. Zhang et al. (2018) confirmed these results and offered mechanisms by which these improvements are achieved.

Decoupling of Parameters Furthermore, Loshchilov and Hutter (2017) showed how the usage of weight decay over L_2-regularization decouples the learning rate and regularization hyperparameter. Figure 4.2 illustrates this fact, which teaches an incredibly valuable lesson. Recall that Henderson et al. (2018b) found a strong dependence of the performance of reinforcement learning algorithms on the hyperparameter settings of their optimizers. Henderson et al. (2018a) thus called for the development of what they called *hyperparameter agnostic* algorithms - algorithms that adjust their hyperparameters during training. Tuning of hyperparameters is a lot easier if they can be tuned mostly independently of each other without strong cross-interaction effects. Thus, AdamW seems like an important step towards achieving this goal.



(a) Adam: notice the interaction between both hyperparameters.



(b) AdamW: the two hyperparameters are mostly decoupled.

Figure 4.2: Heatmap, showing the test error of a deep neural network, trained with Adam or AdamW receptively, on an image classification task for different settings of the learning rate and regularization/weight decay factor (Loshchilov and Hutter, 2017, Figure 2).

4.2 AGENTS

4.2.1 *DQN: Deep Q-Networks*

DQN is a value-based deep reinforcement method for discrete and finite action spaces, first proposed by Mnih et al. (2013) for the use in the *Arcade Learning Environment* (Bellemare et al., 2013), an Atari game simulator.

Basic Idea

DQN works by approximating the optimal Q-function $Q^\star$ of a given state space and action space with a deep neural network with parameters $\vec{\theta}$, that is trained using a variation of Q-learning. It estimates the optimal policy $\pi^\star$ via greedily selecting the action a in state s that maximizes its current estimate of the optimal Q-function.

The agent observes preprocessed video inputs: the raw video frames σ_i are converted from RGB values to gray-scale, down sampled and cropped to reduce the input dimensionality and thus the computational burden. Finally, four consecutive of the so produced frames are stacked to produce one observation, as to approximate full observability [1] of the environment (for such features as the velocity of in-game objects). This observation ϕ_t is then treated as a full state description s_t.

1. Recall that full observability is a prerequisite for a model using a Markov decision process.

The reward given to the agent to maximize, is the raw game score. At each simulated time step t, the old and new observation are bundled together with the chosen action and received reward in a *transition* $(\phi_t, a_t, r_t, \phi_{t+1})$. Additionally, the agent receives a signal when the game terminates.

Instability and Convergence in Training

As the agent learns an approximation of the optimal policy $\pi^\star$ via approximating the corresponding optimal Q-function $Q^\star$, its current estimation of $\pi^\star$ directly depends on its current estimation of $Q^\star$. Therefor the estimation of $Q^\star$ is very unstable (meaning volatile), as a change in the estimation of $Q^\star$ can trigger a change in the estimation of $\pi^\star$ and vice versa, leading to chain reactions. This is especially grave, inasmuch as that both estimates are updated at the same time, whereas the classical policy iteration algorithm, that inspires value-based methods, only updates its policy once the value estimates have converged. DQN thus deploys two techniques to stabilize training (reduce volatility) and speed up the (practical) convergence of the estimates: *experience replay* and *off-policy learning*.

Experience Replay To tame rapid self-reinforcing changes in the estimation of $Q^\star$, DQN does not update $\vec{\theta}$ using the latest transition information, but maintains a *replay memory* in which all observed transitions are stored. For each update step a set of transitions is sampled at random, ensuring that they represent a diverse part of the past experience and smoothing out the updates of $\vec{\theta}$.

Off-Policy Learning As DQN does not solve $Q^\star$ exactly, like value iteration would, but merely estimates it, while at the same time already trying to maximize its rewards, there are two competing goals: the exploration of the action space in order to find better action sequences and the exploitation of what is currently considered the best action. This is known as the *exploration-exploitation trade-off*.

Furthermore, due to the use of experience replay, the network parameters that produced the targets of a minibatch are not identical with the current network parameters. DQN applies a technique called *off-policy learning* to solve this problem: with some small probability ϵ it will choose a random action, while else taking the action currently favored by the greedy selection.

Convergence Note, that while using these techniques hinders a theoretical convergence guarantee of DQN, it practically helps it to reach good estimates. Yang et al. (2019) showed statistical convergence of DQN under mild assumptions, justifying the use of the two techniques.

Behavior Distribution Recall from § 3.1.2 (p. 10) that the transition function of the underlying Markov process together with a policy implies a transition distribution over the state space. This is related to what Mnih et al. (2013) call the *behavior distribution* [2] $\rho(s, a)$ - a probability distribution over states s and actions a, given by the current greedy

2. Although Mnih et al. (2013) basically assume a deterministic transition function, further simplifying the problem.

policy and the off-policy probability, that is used to calculate the expected squared error between a target network and the current estimation of it.

Learning of the Q-Function As the agents estimates $Q^\star(s,a)$ via a deep neural network $Q(s,a;\vec{\theta})$ with parameters $\vec{\theta}$, in each time step it needs to minimize a loss function (the expected squared error):

$$L(\vec{\theta}) = \mathsf{E}_{s,a\sim\rho}[(y_i - Q(s,a,\vec{\theta}))^2] \tag{4.4}$$

3. See how these targets echo the Bellman equation.

with targets [3] y_i (recall the transition distribution $\mathcal{P}$, reward function R and discount factor γ from the definition of Markov decision processes):

$$y_i = \mathsf{E}_{s'\sim\mathcal{P}_a(s)}[R(s,a) + \gamma \cdot \max_{a'} Q(s',a';\vec{\theta}) \mid s,a] \tag{4.5}$$

As introduced, in deep reinforcement learning optimization is solved numerically. Calculating the gradient of L with respect to $\vec{\theta}$ yields:

$$\nabla_{\vec{\theta}} L(\vec{\theta}) = \mathsf{E}_{s,a\sim\rho,s'\sim\mathcal{P}_a(s)}\left[\left(R(s,a) + \gamma\cdot\max_{a'} Q(s',a';\vec{\theta}) - Q(s,a;\vec{\theta})\right) \cdot \nabla_{\vec{\theta}} Q(s,a;\vec{\theta})\right] \tag{4.6}$$

Finally, as deep neural networks are trained using *stochastic* gradient descent, for some minibatch $\vec{m}$ with size k, consisting of transitions $(\phi_i, a_i, r_i, \phi_i')$, we get (assuming $\phi_i \equiv s_i$):

$$\nabla_{\vec{\theta}} L(\vec{\theta},\vec{m},\vec{y}) = \frac{1}{k}\sum_{i=0}^{k}\left(\underbrace{R(\phi_i,a_i) + \gamma\cdot\max_{a'} Q(\phi_i',a';\vec{\theta})}_{=y_i} - Q(\phi_i,a_i;\vec{\theta})\right) \cdot \nabla_{\vec{\theta}} Q(\phi_i,a_i;\vec{\theta}) \tag{4.7}$$

Algorithm Algorithm 3 (p. 26) shows how this estimation of the Q-function is combined with experience replay and the off-policy learning to form DQN. Figure 4.3 visualizes the process.

Extensions

6. Note further, that all these extensions add additional "tricks" to DQN, apparently helping performance in practice, with the theoretical side staying somewhat opaque. To our knowledge, Yang et al. (2019) were the first to deeply investigate DQN's convergence.

Researchers were quick to propose and evaluate a number of extensions to DQN, such as *prioritized experience replay* (Schaul et al., 2015), *Dueling DQN* (Wang et al., 2015), *Double Q-Learning* (Van Hasselt et al., 2016), *Distributional DQN* (Bellemare et al., 2017), *Noisy DQN* (Fortunato et al., 2017) and *Rainbow* Hessel et al. (2018), a combination of the above mentioned, yielding significantly improved performances. Whereas we are mainly interested in the effects of the optimizers and thus focus on the original formulation of DQN [6].

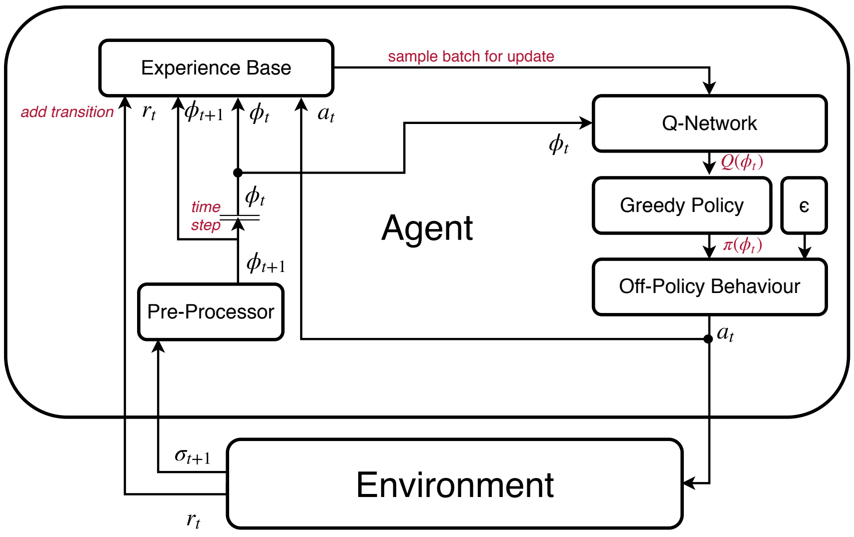


Figure 4.3: DQN. In each time step, the current policy is computed via greedy selection from the output vector of the Q-network and the off-policy probability ϵ. The new observations gets pre-processed and is combined with the old observation, the taken action and the received reward into a transition, which is added to the replay memory. Samples from the replay memory are used to update the Q-network, following Equation 4.7 (p. 20).

4.3 HYPERPARAMETER OPTIMIZATION

Hyperparameter Sensitivity As shown by Henderson et al. (2018b), reinforcement leaning algorithms are sensitive towards the setting of their hyperparameters, such as the learning rates. The authors found this to still hold true for adaptive learning rate algorithms like Adam, which are thought of being more robust due to them finding flatter minima (Heusel et al., 2017), which were traditionally considered to generalize better (Hochreiter and Schmidhuber, 1997; Keskar et al., 2016).

Opaque Tuning Unfortunately, it is common practice in reinforcement learning literature to tune hyperparameters without comprehensive reasoning, leading to hardly comparable results (Henderson et al., 2018a). It is not uncommon to find sentences like "Specifically, we evaluated two hyperparameters over our five training games and choose the values that performed best. The hyperparameter values we considered were [...] $\epsilon_{adam} \in \{1/L, 0.1/L, 0.01/L, 0.001/L, 0.0001/L\}$" (Bellemare et al., 2017, p. 16, Appendix C). Apart from the typical arbitrariness of the considered values, this is an interesting example, as it demonstrates the divergence of theory and practice. As the alert reader will remember, in Adam ϵ is simply a constant to stabilize the division by zero. From a theoretical standpoint, even tuning this constant at all without giving further motivation therefor seems absurd. Furthermore, to foster comparability between results, there should be a clear reasoning as to why and how a hyperparameter was tuned. Practically however, tuning ϵ leads to better results and is thus done, as in the case of [7].

7. The interested reader can find a résumé about the possible reasons for and a quick discussion of the tuning of this hyperparameter in Appendix A (p. 63).

As illustrated with DQN and its extensions, there is a lot of research focused on the development of new techniques and little on the theoretical and practical understandings of existing ones, resulting to an opaque situation in hyperparameter optimization, as bemoaned by Henderson et al. (2018a).

Towards Agnosticism In light of problems like these, Henderson et al. (2018a, p. 3) called for hyperparameter agnostic algorithms: "[...] such that fair comparisons can be made without concern about improper settings for the task at hand".

We regard the use of automatic hyperparameter optimization algorithms as a step towards this goal as it allows developers to draw upon a standardized process instead of manually tinkering with the hyperparameters themselves.

4.3.1 BOBH: Bayesian Optimization with Hyperband

In hyperparameter optimization there are two well known approaches: *Bayesian optimization* and *bandit-based methods*. Both have strengths and weaknesses that we will introduce next, as well as how *BOBH* combines the strengths of both. For we will only introduce all methods very briefly, interested readers are invited to deepen their understanding, e.g. in (Hutter et al., 2018), or for a quick brush-up in Falkner et al. (2018).

Bayesian Optimization

Bayesian optimization is an iterative approach to globally optimize black-box functions. It generates a probabilistic model of the function mapping hyperparameter settings to final performance by creating a *prior probability distribution* (expressing the belief in the behavior of the function without any evidence) and updating it by fitting it to data that is generate by evaluating hyperparameter settings that are generated by maximizing an *acquisition function*.[8] The process is summarized in Figure 4.4.

8. Here once again, the optimizer has to engage in a exploration-exploitation trade-off between generating configurations that probably perform well and reducing the uncertainty in the model of the target function.

Fortes and Flaws State-of-the-art Bayesian optimization methods for machine learning achieve good anytime and final performance, but either scale poorly for higher-dimensional problems and are inflexible with respect to different types of hyperparameters or have not yet been adopted for multi-fidelity optimization (Falkner et al., 2018). Falkner et al. (2018) also noted that *Hyperband* methods are usually quicker to find good settings.

Hyperband

Hyperband (Li et al., 2016) is a bandit-based optimization method resting on the premise that while the exact evaluation of the target function

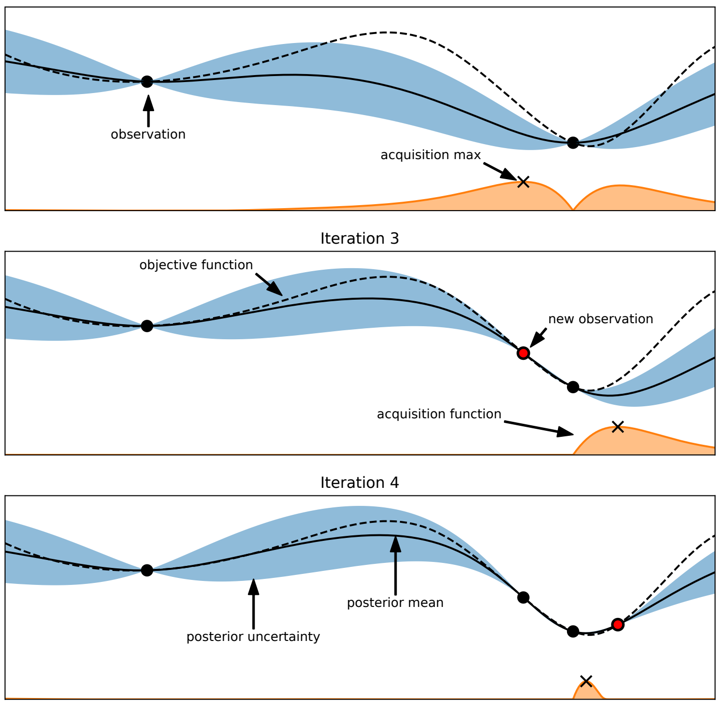


Figure 4.4: Illustration of Bayesian optimization for a one-dimensional function (Hutter et al., 2018, Figure 1.2). The dashed line is the objective function, the black line its current estimate and the blue area the confidence interval around it. The orange line and the area beneath it, depict the acquisition function: it is large in areas in which the estimate of the objective function is small and/or the confidence interval is large. New configurations (red dots) are chosen by maximizing the acquisition function.

is computationally expensive, it is possible to define a cheaper approximation for a given computational budget b that gets closer to the target function with increasing budget.[9]

9. In reinforcement learning, training multiple agents for a given setting or a single agent for a longer time will typically produce a more stable estimate of the expected performance of a hyperparameter setting.

It takes a total budget and divides it between a number of rounds that each have the same round budget but spend it differently on the number of sampled configurations and budget per configuration.

Then, in each round it samples a number n of random configurations and evaluates them on the lowest round budget $b = b_{min}$, iteratively only keeping the $\frac{n}{\eta}$ best performing configurations and evaluating them on the next higher budget $b \cdot \eta$ until the maximum round budget $b = b_{max}$ is reached. This process is called *successive halving* with hyperparameter η.

Fortes and Flaws According to Falkner et al. (2018) Hyperband works well in practice, typically outperforming Bayesian methods and random search on small to medium budgets, showing strong anytime performance while being flexible and scalable, but often having a worse final performance, especially for large budgets.

BOBH: Combining Hyperband and Bayesian Optimization

BOBH combines the strong final performance of Bayesian methods with the better any-time performance of Hyperband by building a model-based Hyperband variant, also fulfilling other virtues such as good parallelization, scalability to high-dimensional problems, robustness and flexibility, simplicity and computational efficiency [10] (Falkner et al., 2018).

10. Especially the parallelization and robustness are crucial in the noisy and computational expensive domain of reinforcement learning.

It essentially follows the same procedure as Hyperband, but instead of randomly sampling the configuration for each round, it builds a Bayesian model to select the configurations. In addition to the model-based picks, it still samples a fixed number ρ of random configurations to keep the convergence guarantees of Hyperband Falkner et al. (2018).

Bayesian Model Given a set D of observations (pairs of configurations $\vec{x}_i$ and observed values y_i), $D = \{(\vec{x}_o, y_o), \ldots (\vec{x}_n, y_n)\}$, BOBH selects a new configuration $\vec{x}$ by maximizing the *Tree Parzen Estimator* (Bergstra et al., 2011), which is equivalent to maximizing the often used fidelity of *expected improvement* over the currently best observed value $\alpha := \min\{y_i \mid i \in \{0, \ldots, n\}\}$:

$$a(\vec{x}) = \int \max(0, \alpha - f(\vec{x})) \cdot dp(f \mid D) \qquad (4.8)$$

where f is the objective function and $p(f \mid D)$ is the probabilistic model of the objective function given the data D. See Falkner et al. (2018) for how p is fitted to new data.

Parallelization BOBH is designed to be parallelized, meaning that it will usually sample multiple configurations at once and schedule their evaluation simultaneously. Thus, the update of the model will usually not happen after each sampling of a single configuration, but only once its evaluation is finished. Hence, the update of the model is subject to some delay, as other data points will be sampled in the meantime. How this trade-off between stronger parallelization and more frequent model-updates is handled, depends on the implementation.

Method Algorithm 4 (p. 27) summarizes a simplified version of BOHB. For each *bracket s*, a number $n_{configs}$ of configurations are sampled that then all go through the process of *successive halving* [11] - only keeping and evaluating the top $\frac{1}{\eta}$ share and evaluating those on the next higher budget. The brackets differ in how *aggressively* the evaluation is performed - starting with a large number of configurations and a low budget and moving into the opposite direction.

11. Despite the name suggesting a halving of the number of configurations, a common default for η is in fact 3.

For comprehensibility, we simplified the sampling process of BOBH in Algorithm 4 (p. 27). Depending on the implementation, not all configurations are sampled at once, but only as many as can be schedule at a given moment, such that the evaluations of these configurations can also be used to guide the sampling of other configurations for the same bracket and budget.[12]

12. Note, how this implies a trade-off between maximal computational parallelization and earliest usage of the model.

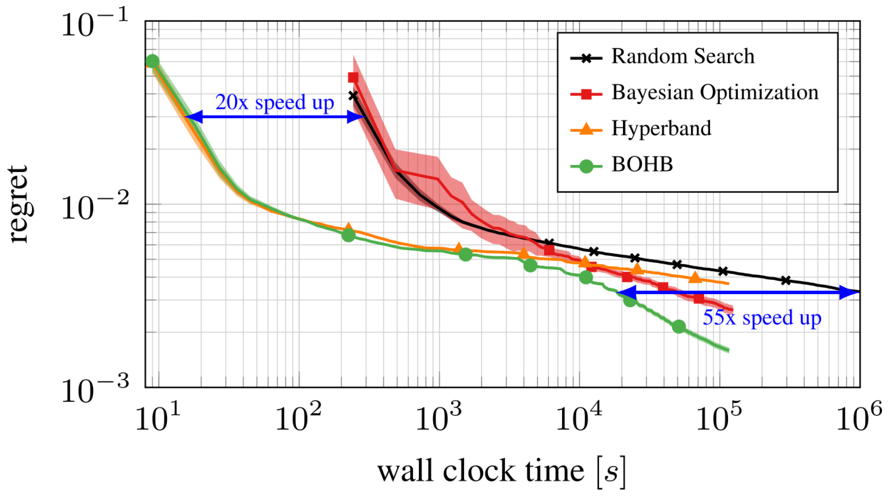


Figure 4.5: Characteristic performance of BOBH as compared with Hyperband and Bayesian optimization (Falkner et al., 2018, Figure 1). The plot shows the regret of using a certain hyperparameter optimization methods by elapsed time of the optimization run. The regret is the difference between the performance of an optimal configuration and the best configuration found by the algorithm and thus is a measure for the performance of the algorithm. Notice the characteristic improvement in early performance compared to Bayesian optimization and late performance compared to Hyperband.

Fortes and Flaws Falkner et al. (2018) found that BOHB combines the strong final performance of Bayesian methods with the better anytime performance of Hyperband. Figure 4.5 illustrates this characteristic speedup. Furthermore, it is designed to parallelize well. On the downside, it requires some settings of its own hyperparameters and a trade-off between maximal parallelization and best sampling of the configurations.

```
Hyperparameters: number of episodes $m$,
                 number of time steps per episode $l$,
                 size of replay memory $n$,
                 off-policy probability $\epsilon$,
                 minibatch size $b$
Given: starting observation $\sigma_0$, pre processing function $\phi$,
       Network $Q$ with parameters $\vec{\theta}$,
       a Markov decision process [4] $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$
Init: $\vec{\theta}$ randomly, replay memory $\mathcal{D}$ with size $n$
foreach episode $= 0 \ldots, m-1$ do
    $\kappa_0 \leftarrow \sigma_0$ ;                              (initialize history)
    $\phi_0 \leftarrow \phi(\kappa_o)$ ;                          (preprocessed history)
    foreach $t = 1, \ldots, l$ do
        sample $x \in X \sim B(\epsilon)$ ;                       (Bernoulli distributed)
        if $x = 1$ then
            select $a_t$ uniformly at random ;                    (off-policy)
        else
            $a_t \leftarrow \arg\max_a Q(\phi(s_t), a; \vec{\theta})$ ;    (on-policy)
        end
        $(r_t, \sigma_{t+1}) \leftarrow$ execute$(a_t)$;
        $\kappa_{t+1} = \kappa_t, a_t, \sigma_{t+1}$;
        $\phi_{t+1} = \phi(\kappa_{t+1})$;
        $\mathcal{D}$.add$((\phi_t, a_t, r_t, \phi_{t+1}))$;
        $\vec{m} \leftarrow \mathcal{D}$.sample_minibatch$(b)$;
        foreach $(\phi_i, a_i, r_i, \phi_{i+1}) \in \vec{m}$ do
            if terminal$(\phi_{i+1})$ then
                $y_i \leftarrow r_i$
            else
                $y_i \leftarrow r_i + \gamma \cdot \max_{a'} Q(\phi_{i+1}, a'; \vec{\theta})$
            end
        end
        $\vec{\theta} \leftarrow$ gradient_descent_step$(\vec{\theta}, \vec{m}, \vec{y})$ ;   (see eq. 4.7)
    end
end
```

Algorithm 3: Deep Q-Learning with Experience Replay. [5] For each training episode a number of training time steps will be executed. Each of these consists of: randomly determining whether to take an off-policy action or not, applying that action, storing the transition in the replay memory, sampling a minibatch from the latter and making one gradient step.

4. Or an approximation of one as, again, the Markov property is just assumed as an approximation.

5. Note, how the history of the agent is related to the notion of the trajectory.

```
Hyperparameters: minimum and maximum budget $b_{min}$, $b_{max}$,
                 fraction to evaluate on next higher budget $\eta$,
                 fraction of random configurations $\rho$,
                 min. num. of data points to build model $n_{min}$
Given: configuration space $\mathcal{C}$, objective function $f$
Init: $D \leftarrow \{\}$ ;        (empty data base for each budget)
$s_{max} \leftarrow \left\lfloor \log_\eta \frac{b_{max}}{b_{min}} \right\rfloor$;
foreach $s \in \{s_{max}, s_{max} - 1, \dots, 0\}$ do
    $n_{\text{configs}} = \left\lceil \frac{s_{max}+1}{s+1} \cdot \eta^s \right\rceil$;
    $b \leftarrow \eta^s \cdot b_{max}$;
    $C \leftarrow$sample $n_{config}$ configurations;
    foreach $i \in \{0, \dots, s\}$ do
        $D_i$.add($f(C)$);  (evaluate and add to data base)
        $C \leftarrow$ keep-the-$\frac{|C|}{\eta}$-best($C$);
        $b \leftarrow b \cdot \eta$;
    end
end
Function sample($D, \rho, n_{min}, n$):
    if num of random configs so far $< \rho$ then
        return random-configuration($\mathcal{C}$)
    else if $\{D_b : |D_b| \geq n_{min} + 2\} = \varnothing$;   (not enough data to
     build model for any budget)
     then
        return random-configuration($\mathcal{C}$)
    else
        build model for $b_m = \arg\max_b \{D_b : |D_b| \geq n_{min} + 2\}$;
        sample from $C$ according to model;      (see Eq. 4.8)
    end
```

Algorithm 4: A simplified depiction of BOBH. For each bracket s, a number $n_{configs}$ of configurations are sampled that then are evaluated using successive halving. Note, how the data points are stored and the models are build per budget b.

5 TOOLS AND LIBRARIES

Reproducibility Henderson et al. (2018a) found a staggering difference in performance for reinforcement learning algorithms with the same hyperparameter settings for different code bases. We therefore carefully present all used libraries and tools to achieve maximal reproducibility and transparency.

For the same reason, all code for our experiments can be found in a accompanying Git repository.[1] Our custom code is packaged as a Pip-installable Python package containing all requirements, such that it can be installed and executed without any additional setup, we solely recommend the usage of a virtual Python environment, e.g. via Conda.[2]

1. See github.com/vonHartz/bachelorthesis-public

2. For information, see conda.io.

Library Versions All computations were done in Python3. For tested versions of Python and the used libraries see the `requirements.txt` of the Python package in the repository. Newer versions may still work if no breaking API-changes were introduced. We recommend to setup a new virtual Python environment using Pip and the accompanying `requirements.txt` or the provided `Makefile`.[3]

3. For information on Pip visit pip.pypa.io, for GNU Make gnu.org/software/make. A quick start guide can be found in appendix Appendix B (p. 65).

5.1 NUMERICAL OPTIMIZATION

We used the implementations of Adam and AdamW provided by Tensorflow (Abadi et al., 2015).[4] Both are direct implementations of the algorithms proposed by Kingma and Ba (2014) and Loshchilov and Hutter (2017) and did not need any customization for our needs, but worked natively with the agents' implementation.

Since we were interested in how volatile the optimization proceeds, we logged the optimizers' internal momentum variables $\vec{m}$ and $\vec{v}$, but did not have to customize the optimizers themselves for this.

4. The source code is publicly available at github.com/tensorflow/tensorflow.

5.2 AGENTS

The agents are taken from Ray (Moritz et al., 2018), more specifically its Rllib (Liang et al., 2017).[5] We used the standard DQN implementation (not the APE-X variant) and disabled advanced techniques, such as Dueling- and Double-DQN, to get a basic DQN, as we want to focus on the influence of the optimizer and its hyperparameters.

5. The source code can be found at github.com/ray-project/ray.

We then created custom agents for both Adam and AdamW, inheriting from the vanilla DQN agent, as to 1. use a custom optimizer (in the case of AdamW) and 2. implement custom features, such as logging of the optimizers internal values, for our experiments.

Ray furthermore provides training and rollout routines for the agents, that we used to build our evaluation workflow, that we will describe in detail in the experiments section.

The main rationale for the choice of Rllib is that Ray is build as a framework for distributed execution, making effective use of parallelization, while introducing relatively little overhead, thus speeding up the expensive training of reinforcement learning algorithms despite the little cost of individual operations that usually slow down distributed systems (Nishihara et al., 2017).

Ray moreover works natively with the Tensorflow implementations of the used optimizers, as well as the used environments.

Crucially, it also provides a good groundwork of logging the agents' training statistics to Tensorflow's Tensorboard , upon which our custom loggers were build.

Training In Rllib agents are trained in individual *training iterations* that in our case each last for a fixed number of time steps, reporting statistics, such as the mean episode reward, after each iteration. A single training iteration can thereby contain an arbitrary number of episodes, as the episodes are terminated by the environments when a terminal state is reached. By controlling the number of training iterations and the number of time steps, we are able to train the agents for a predetermined number of time steps, which allows to fairly evaluate the agents learning capabilities, and balance the resolution of the training statistics.

5.3 ENVIRONMENTS

Ray natively works with environments implemented in OpenAI Gym (Brockman et al., 2016).

5.3.1 *Cartpole*

As it is computationally relatively cheap to evaluate and often used in reinforcement learning research, we focused on the classical control tasks Cartpole[6] as a baseline problem. In it "[a] pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart's velocity" (CartPole v0, 2019).

6. More precisely, we used *CartPole V0*, see github.com/openai/gym/wiki/CartPole-v0.

The training episode is automatically terminated, when one of the following conditions is met:

- the magnitude of the angle between the cart and the pole is greater than 12°,

- the cart leaves the predefined display area, or
- the length of the episode exceeds 200 time steps.

The reward given to the agent is 1 per time step, hence the episode reward equates the episode length.

The agents observes the following quantities, which fully describe the environment state: the 1. cart position, 2. cart velocity, 3. pole angle and 4. pole velocity.

Its sole available actions are pushing the cart to the left or right.

Episodes are initialized with random values of these four variables.

Fortes and Flaws As it is computationally inexpensive and the task does not require a lot of abstraction and generalization, it is easy and fast to learn, which makes it convenient for a cheap performance estimation, as we are interested in. On the other hand it does not challenge the agent's generalization, thus not uncovering potential biases towards local minima that fail to generalize well.

5.4 HYPERPARAMETER OPTIMIZATION

We used the BOBH implementation provided by HPBanster.[7] It is build with distributed computation in mind, with one central server building the model and scheduling evaluations on a number of workers.

7. See github.com/automl/HpBandSter

Evaluations are scheduled greedily: the server will sample and dispatch as many configurations as workers are currently available. Remember that this implies a trade-off between speeding up the computation by using a large number of workers and using the model as early as possible to sample new configurations, as the model used by BOBH to sample configurations is build using past evaluations on the same budget.

Fortes and Flaws Note that while HPBanster is build for parallel computation, it installs its own abstraction layer for it by introducing workers that execute computations for a central server. As we already use Ray to parallelize computations, this introduces some complications in the design of the software. Ray introduced its own hyperparameter optimization framework Tune (Liaw et al., 2018). For future work, it might be worthwhile to implement BOBH in Tune, as a remove a further layer of abstraction and the computational overhead that comes with it.

5.5 VISUALIZATION AND EVALUATION

5.5.1 *Logging*

As noted, the used agents create Tensorboard logs of the training process, plotting quantities such as the episode reward per training iteration.

Tracking Momentum Tensors To gain further inside into how volatile the training process proceeds, we built a custom Tensorflow logger that also tracks the optimizers internal momentum variables $\vec{m}$ and $\vec{v}$. As these are large tensors, entirely tracking them over the whole trajectory would introduce large memory requirements. Instead we tracked them in three more efficient ways. Keep in mind, that we were primarily interested in how strongly these values change.

1. We calculated the means of each tensor and tracked those instead. Note though, that this underestimates the actual volatility. [8]
2. To get an unbiased impression of the volatility of the tensors across time steps, we buffered them for one time step respectively and in each time step reported the Euclidean distance of each tensor to the last time step's tensor.
3. To get an impression of the spread of the momentum variables, we aggregated the scalar values of each scalar in each time step into a histogram. This gives a fair overview over the tensor's scalar values, while accumulating reasonably less data when the bucket count is not chosen to highly. From the histogram, distribution plots are also automatically generated.

8. As the change of one scalar value in the tensor in one direction will in calculation of the mean effectively be subtracted from the change of another scalar in the opposite direction, the change in the mean underestimates the change of the individual scalars.

Aggregation As in our experiments we wanted to compare different optimizers and hyperparameter settings, we needed to get an impression of 1. the average performance of an optimizer/setting and 2. the variance of the performance.

For this implies that we had to make multiple runs of any trial, we also created a script to automatically aggregate the individual log files of repeated trials into a single Tensorflow log, displaying the mean and variance of scalar values across all trials. [9]

9. Our script is based on the script released by Sebastian Penhouet under the MIT License, see github.com/Spenhouet/tensorboard-aggregator.

Performing the aggregation directly on the Tensorflow logs has the advantages of 1. still being able to compare the aggregated log files in Tensorboard and 2. automatability and flexibility, as all computations are performed on a common format. The resulting graphs with confidence intervals allow for a nice side-to-side comparison of different configurations and algorithms.

Figure 5.1 shows examples for the four plot types and illustrates their use in comparing different optimizers and configurations.

Note, that as we also wanted to aggregate the histograms of runs with the same configuration to their respective mean, we had to set a fixed resolution and domain of the histograms in advance, which comes at the cost of potentially displaying areas of no interest while having a lower resolution in areas of interest.

5.6 HYPERPARAMETER ANALYSIS

CAVE In order to evaluate the results from HpBandster/BOBH, we relied on CAVE (Biedenkapp et al., 2018), a tool to automatically analyze the data produced by the run of a hyperparameter optimization method, such as BOBH, reporting among other things, a performance analysis, hyperparameter importance and feature analysis.

CAVE works natively with the artifacts created by running an instance of the BOBH implementation in HPBanster.

We were particularly interested in the hyperparameter importance, which CAVE evaluates using *fANOVA*, *ablation analysis* and *local parameter importance*.

fANOVA (Hutter et al., 2014) is a method to evaluate both the global importance of a single hyperparameter for the performance of a machine learning model, as well as interactions of such.

It is based on fitting a random forest[10] on existing data gathered by a Bayesian optimization method (such as BOBH) to approximate a function, mapping the hyperparameter space to the performance value, and then applying a *functional ANOVA* (analysis of variance) to asses the hyperparameters' individual importance and interaction effects (Hutter et al., 2014).

10. Random forests are a popular and computationally efficient class of machine learning models. A random fore st is an ensemble of regression trees, which each sequentially partition the input space into areas that are each assigned a constant prediction based on the training data. The prediction of the forest then is the average of the predictions of the individual trees.

Functional ANOVA (Hooker, 2007; Huang et al., 1998) decomposes the variance $\mathrm{Var}[f]$ of some black-box function[11] $f : \Theta_0 \times \cdots \times \Theta_k \to \mathbf{R}$ with parameter space $\Theta = \Pi_{i=0}^{k} \Theta_i$ into a set of disjunct components $\mathrm{Var}[f] = \sum_{s \in \mathcal{P}\{\Theta_i | 0 \leq i \leq k\}} v_s$ with one component per set s of parameters.

11. In our case, a function mapping the hyperparameter space of some algorithm to the expected performance value of the algorithm.

Hutter et al. (2014) again use random forests to produce their functional ANOVA, using the fact that random forests are an ensemble method, in which each single tree t defines a partition P_t of the parameter space Θ.

Analyzing these different partitions, allows to efficiently estimate the marginal prediction of each parameter set $s \subseteq \{\Theta_i \mid 0 \leq i \leq k\}$, which is equivalent to the respective fractional variance v_s.

The importance of a single hyperparameter or the interaction of multiple hyperparameters is then given by the respective fractional variance v_s.

Ablation Analysis (Fawcett and Hoos, 2016; Biedenkapp et al., 2017) determines the parameter importance locally by comparing two parameter configurations θ_{source} and θ_{target}. It ranks the parameters by starting with θ_{source} and successively, greedily selecting the parameter $p \in P$, from the set P of parameters that span Θ, that if it is changed from its value in θ_{source} to its value in θ_{target}, while keeping the other parameters constant, improves the performance the most. To judge the performance of these hypothetical parameter configurations, ablation analysis resorts to estimating it via an *empirical performance model*, fitted on the available data - similar to fANOVA.

Local Parameter Importance (Biedenkapp et al., 2018), similarly, is also a local method, investigating the neighborhood of a single parameter configuration. It also builds an empirical performance model and then judges the importance of a single parameter p_i by first estimating the variance generated by changing each parameter around the considered configuration and then dividing the so estimated variance for p_i by the sum of the so estimated variances for all parameters in P. Let $\hat{f}$ be the empirical performance model for an algorithm with a hyperparameter space Θ, spanned by a set of parameters P, and let $\theta_t \in \Theta$ be a configuration and $p_i \in P$ be a parameter with domain Θ_i. Let $\theta[p = x]$ denote the configuration $\hat{\theta}$, in which the parameter p takes the value x and all other parameters take the same values as in θ. Then:

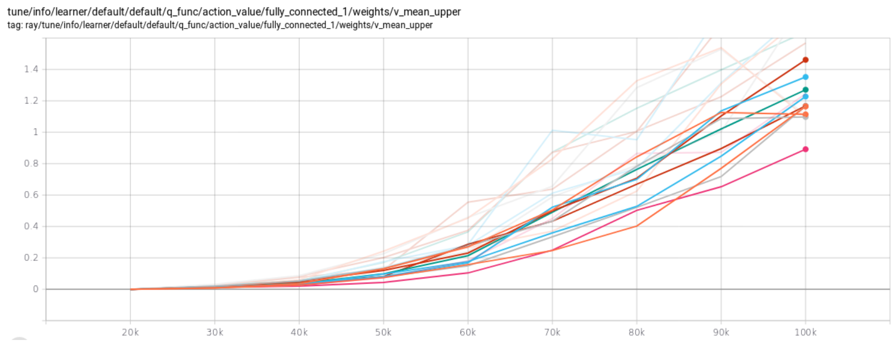
$$LPI(p_i \mid \theta_t) := \frac{\mathrm{Var}_{v \in \Theta_i}[\hat{f}(\theta_t[p_i = v])]}{\sum_{p_j \in P} \mathrm{Var}_{w \in \Theta_j}[\hat{f}(\theta_t[p_j = w])]}$$

5.7 COMPUTATIONAL RESOURCES

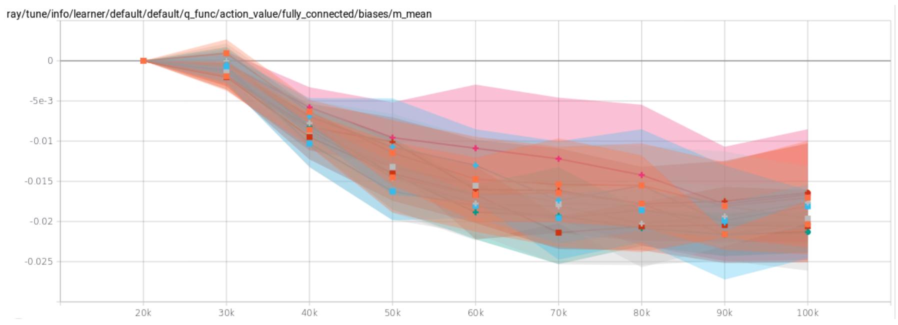
As our experiments required large computational capacities, they were carried out on two computation clusters: META and BwForCluster NEMO. META runs the SLURM workload manager, NEMO uses MOAB.[12] Job scripts to execute and reproduce our experiments, along with instructions and hints, can be found in the accompanying repository.

12. See slurm.schedmd.com and adaptivecomputing.com/moab-hpc-basic-edition

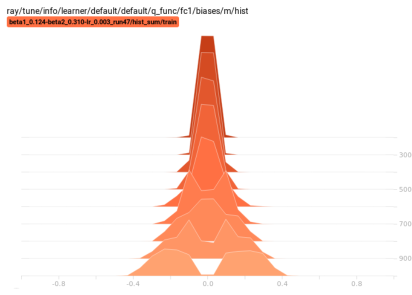
For flexibility and portability, all experiments were run on CPUs, not using any GPUs.



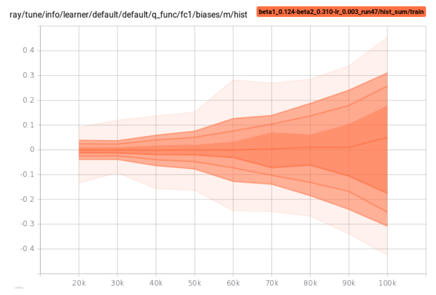
(a) An example of a vanilla Tensorboard graph, showing the mean of the moment-tensor $\vec{v}$ of the weights of one layer of the Q-network over the training time steps for one training run each for eight configurations.



(b) An example for a custom Tensorboard graph with confidence intervals, showing the mean of the moment-tensor $\vec{m}$ of the biases of one layer of the Q-network over the training time steps, for eight configurations, each aggregated over multiple training runs.



(c) An example for a custom Tensorboard histogram, showing the distribution of scalars for one moment-tensor $\vec{m}$ over the training time steps, aggregated over multiple training runs for the same configuration.



(d) An example for a Tensorboard distribution plot, automatically generated for the distribution in subfigure c. The middle line represents the median of the distribution, the colored areas the 100%, 93%, 84% and 69% percentiles.

Figure 5.1: Dummy examples for the four generated plot types. While the vanilla graph (subfigure a) allows to nicely compare a limited number of runs over some time interval, it does not allow to neatly compare a number of runs for multiple agents or configurations at a time. The confidence interval plot (subfigure b) fills this gap, as the individual runs are aggregated to their mean and variance. The histogram (subfigure c) and distribution plots (subfigure d) furthermore allow to visualize larger tensors across time steps.

PART II

EXPERIMENTS

6 PERFORMANCE ESTIMATION

"It's Difficult to Make Predictions, Especially About the Future"

— Karl Kristian Steincke, *(probably)*

6.1 PROBLEM

Estimating Performance As introduced in § 4.3.1 (p. 22), Hyperband, on which BOBH is based, builds on the premise that it is possible to define a cheaper to evaluate approximation $\hat{f}_b$ of an expensive to evaluate target function f for a given computational budget b, that becomes a better approximation for f with increasing b.

For we are interested in optimizing the hyperparameter settings of a reinforcement learning agent, we require a way to, for a given computational budget, estimate the final performance of the agent for a certain hyperparameter setting.

Volatility Henderson et al. (2018a) showed, that besides fixable factors, such as the used code base and hyperparameter settings, random seeds and running different trials of the same experiment play a big role for the achieved performance in reinforcement learning, due to stochasticity in the environment and the learning process. Figure 6.1 illustrates this fact; two groups of five runs each with identical hyperparameter settings produced statistically significant results.

In the experiment, agents were trained for two million time steps - shorter training times might even increase the uncovered volatility.

6.1.1 *Success Metrics*

Another crucial point to consider was the metric by which the agents' performance is judged. Naive implementations might just consider the mean reward during training. However, this might not be optimal, as the goal is an as good as possible *final* performance of the agent. For example, an agent that explores a broader action space during training might yield lower rewards during training but a better performing final policy. We thus collected *training* and *rollout metrics*.

Training Metric To estimate the final performance instead of the anytime performance, we used a *last n mean reward*, i.e. the mean reward over some last n number of training iterations. As the agents were

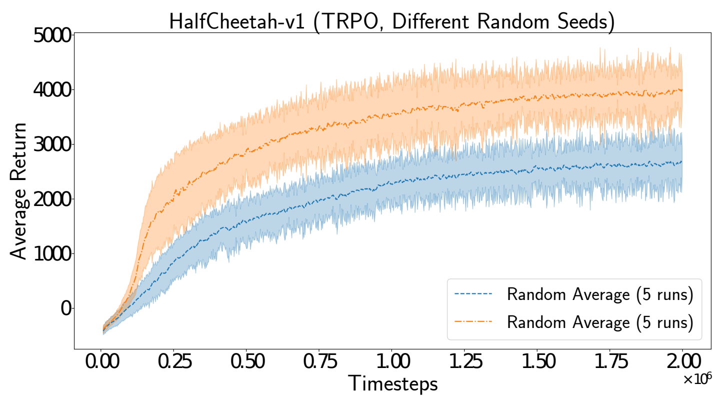


Figure 6.1: Difference between multiple aggregated trials (Henderson et al., 2018a, figure 5). The authors trained a TPRO agent in the half cheetah (OpenAI Gym) environment 10 times using the same hyperparameter settings and then splitted the runs in two groups with five runs each. The plot shows the aggregation for both groups. Note, "[...] that the variance between runs is enough to create statistically different distributions just from varying random seeds" (Henderson et al., 2018a, p. 5)..

trained for 200 iterations with 1000 time steps each, we chose $n = 20$, capturing the last 10% of training iterations.

Rollout Metric Because the episode length in our experiments was not fixed, due to environments automatically terminating the episode when certain criteria were met (see § 5.3 (p. 30)), simply calculating the mean reward over all episodes, might introduce some bias. Short, fast converging episodes would not get awarded enough, as due to their early termination they do not generate a much higher reward, while still increasing the number of episodes, and hence do not play a significant role in the mean.

The alternative was to calculate the total sum of episode rewards, divided by the total number of time steps. The advantage of this is that by dividing the sum of rewards by the number of time steps instead of the number of episodes, short, converging episodes are more awarded.

Both metrics however, do not punish agents a lot for short, diverging episodes.

Due to the reward structure in Cartpole though, we had to resort to the naive mean. Recall, that the reward in Cartpole is equivalent to the episode length. Using the alternative metric would thus return a constant result of 1 for each rollout.

6.2 EXPERIMENTAL SETUP

For we had to predict the performance of an agent for a given hyperparameter setting, while the former is subject to a large variance, it seemed obvious to not just train a single agent, but a number of agents.

	Parameter	Value
Ray	Dueling DQN	False
	Double DQN	False
	Prioritized replay	False
	Noisy DQN	False
	Time steps per training iteration	1000
	$n_{training\ iterations}$ for $n_{agents} \propto$ budget	200
	Sample batch size	200
	Training batch size	32
Adam	Θ_α	$[10^{-6}, 10^{-1}]$
	Θ_{β_1}	$[10^{-1}, 1]$
	Θ_{β_2}	$[10^{-1}, 1]$
	ϵ	10^{-8}
Runner	Budget steps	$9, 27, 81$
	Rollout time steps	10000
	Last n for training metric	20
	n_{agents} for $n_{training\ iterations} \propto$ budget	15
	Runs per sampled configuration	5
	Sampled configurations	5

Table 6.1: Parameters in the estimation experiment. Other parameters of DQN were left at the default. The number of training iterations for the number-of-agents interpretation of the budget and the number of agents for the training-time interpretation of the budget were chosen such that the training cost for both interpretations was roughly equivalent. The runner parameters were chosen by computational feasibility.

Because we had to do so on a given computational budget, there were two immediate factors to consider: the number of trained agents and the training duration, as both heavily influence the computational cost.

Interpretation of the budget The budget could be interpreted as either of the two: training a fixed number of agents for a number of time steps that corresponds (or is proportional) to the budget, or training a number of agents that correspond to the budget for a fixed number of time steps.

We thus designed an experiment to investigate which of the two interpretations gives a more stable estimate - for which the correlation of the performance on different budgets was higher.

Preliminaries We used DQN agents with vanilla Adam and the Cartpole environment. We resorted to mostly using default values for the hyperparameters of the agent. A list of parameter settings for the experiment can be found in table Table 6.1. The only parameters that were not fixed but sampled were the α, β_1 and β_2 parameters of Adam.

Setup For both interpretations of the budget we created routines, carrying out 1. the training of a number of agents and 2. the rollout of each of the agents for a fixed number of time steps.

We then randomly sampled five different configurations from the configuration space spanned by $\Theta_\alpha \times \Theta_{beta_1} \times \Theta_{beta_2}$, with $\Theta_\alpha = [10^{-6}, 10^{-1}]$, $\Theta_{\beta_1} = \Theta_{\beta_2} = [10^{-1}, 1]$. The domains were chosen as widely as possible, not to perform any manual pre-tuning. The values from each Θ_i were sampled using a logarithmic scale.

Furthermore we set a set of budget steps, analog to following successive halving with $\eta = 3$ with an arbitrary starting point, $\{9, 27, 81\}$.

For each sampled configuration, five trials were executed for each of the interpretation of the budget and on each budget step. We collected both training and rollout metrics, to investigate which of them give the more useful estimates.

6.3 RESULTS

6.4 DISCUSSION

HYPERPARAMETER OPTIMIZATION

7

7.1 PROBLEM

The problems of numerical optimization in deep reinforcement learning are manifold. Recall the following issues and open questions:

1. **local optima** in which agents could get stuck
2. **sensitivity towards hyperparameters** with small differences deciding between convergence and divergence
3. **volatility in performance** making it difficult to compare results from different publications

In this chapter, we investigate whether hyperparameter optimization, equipped with the insight of the last chapter, can help with these issues.

7.2 EXPERIMENTAL SETUP

Using the results from Chapter 6 (p. 39) , we use BOBH to automatically optimize the hyperparameters of Adam and AdamW.

What are they?

7.2.1 *Architecture*

The used implementation of BOBH starts a central server, which samples configurations and dispatches their evaluation on a number of workers. This allows for further parallelization beyond what is provide by Rllib.

We make use of a number of computation nodes by starting a BOBH worker on each and a BOBH server on one of them. On each node an instance of Ray is running, all of which are connected to a head, which is started on the first node. As each BOBH worker sequentially trains a number of agents,[1] these will be executed by Ray not only making use of all cores of the node that it is run on, but also *spilling over* to other nodes, if necessary and possible. This architecture is illustrated in Figure 7.1.

1. Ray is not designed to run multiple - what is called - *trials* at once.

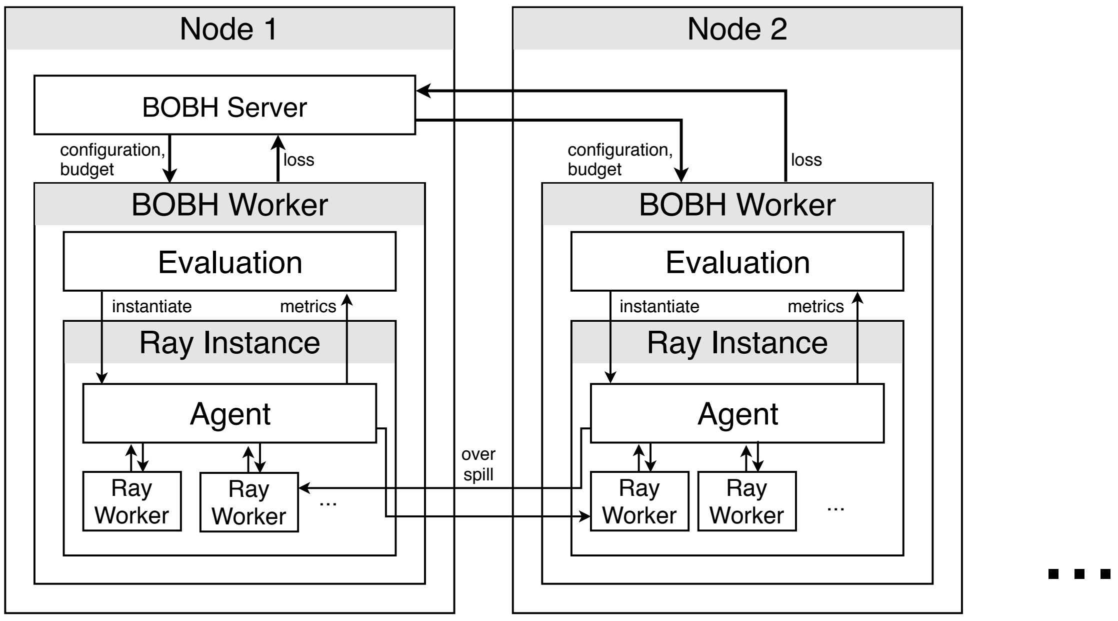


Figure 7.1: Sketch of the architecture of hyperparameter optimization experiment. Note, that the BOBH worker reports *loss*, which simply is negative performance. Upon receiving a configuration and a budget, a BOBH worker will start an evaluation by sequentially instantiating agents with the given configuration and training them on the given budget. Using Ray's parallelization, each agent might use multiple Ray workers and even *spill over* computations to other connected Ray instances.

7.2.2 Parameters

Just as in the last chapter, we used a basic DQN and mostly identical settings. Table 7.1 lists all parameters specific to this experiment, i.e. the parameters for BOBH. Other parameters are identical to the the experiment in the last chapter.

n_iterations von BOBH eingef"uhrt?

	Parameter	Value
BOBH	b_{max}	243
	b_{min}	9
	η	3
	ρ	$\frac{1}{3}$
	$n_{workers}$	10
	n_{min}	dim +1

Table 7.1: Parameters specific to the hyperparameter optimization experiment. Other parameters were chosen identical to the performance estimation experiment, see Table 6.1. *dim* denotes the dimensionality of the configuration space. Other parameters of BOBH that were not introduced were left at their default. The number of workers was chosen according to the number of used computation nodes. η and ρ are at their default, the budgets were chosen based on the results from the last chapter.

7.3 RESULTS

7.4 DISCUSSION

OPTIMIZER ANALYSIS

8

8.1 PROBLEM

In this chapter we will make a closer analysis of Adam with and without weight decay, building on the results of the last two chapters. We also shed some light on the not well understood effects of weight decay on Adam in reinforcement learning

8.2 EXPERIMENTAL SETUP

8.3 RESULTS

8.4 DISCUSSION

PART III

SUMMARY

CONCLUSION

9

None

BIBLIOGRAPHY

Abadi, Martín; Agarwal, Ashish; Barham, Paul; Brevdo, Eugene; Chen, Zhifeng; Citro, Craig; Corrado, Greg S.; Davis, Andy; Dean, Jeffrey; Devin, Matthieu; Ghemawat, Sanjay; Goodfellow, Ian; Harp, Andrew; Irving, Geoffrey; Isard, Michael; Jia, Yangqing; Jozefowicz, Rafal; Kaiser, Lukasz; Kudlur, Manjunath; Levenberg, Josh; Mané, Dandelion; Monga, Rajat; Moore, Sherry; Murray, Derek; Olah, Chris; Schuster, Mike; Shlens, Jonathon; Steiner, Benoit; Sutskever, Ilya; Talwar, Kunal; Tucker, Paul; Vanhoucke, Vincent; Vasudevan, Vijay; Viégas, Fernanda; Vinyals, Oriol; Warden, Pete; Wattenberg, Martin; Wicke, Martin; Yu, Yuan; and Zheng, Xiaoqiang. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. URL https://www.tensorflow.org/. Software available from tensorflow.org. Cited on p. 29.

Arulkumaran, Kai; Deisenroth, Marc Peter; Brundage, Miles; and Bharath, Anil Anthony. 2017. *A brief survey of deep reinforcement learning*. In arXiv preprint arXiv:1708.05866. Cited on p. 12.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. jun 2013. *The Arcade Learning Environment: An Evaluation Platform for General Agents*. In Journal of Artificial Intelligence Research, vol. 47, pp. 253–279. Cited on p. 18.

Bellemare, Marc G; Dabney, Will; and Munos, Rémi. 2017. *A distributional perspective on reinforcement learning*. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 449–458. JMLR. org. Cited on pp. 20, 21, and 64.

Bellman, Richard. 1957. *A Markovian decision process*. In Journal of Mathematics and Mechanics, vol. 6, no. 5, pp. 679–684. Cited on p. 11.

Bergstra, James S; Bardenet, Rémi; Bengio, Yoshua; and Kégl, Balázs. 2011. *Algorithms for hyper-parameter optimization*. In Advances in neural information processing systems, pp. 2546–2554. Cited on p. 24.

Biedenkapp, A.; Marben, J.; Lindauer, M.; and Hutter, F. June 2018. *CAVE: Configuration Assessment, Visualization and Evaluation*. In Proceedings of the International Conference on Learning and Intelligent Optimization (LION'18). Cited on pp. 32 and 33.

Biedenkapp, André; Lindauer, Marius; Eggensperger, Katharina; Hutter, Frank; Fawcett, Chris; and Hoos, Holger. 2017. *Efficient parameter importance analysis via ablation with surrogates*. In Thirty-First AAAI Conference on Artificial Intelligence. Cited on p. 33.

Bringhurst, Robert. October 2004. *The elements of typographic style*. Hartley & Marks Publishers, Point Roberts, WA, USA, 3rd edn. ISBN 0-881-79205-5. Cited on p. d.

Brockman, Greg; Cheung, Vicki; Pettersson, Ludwig; Schneider, Jonas; Schulman, John; Tang, Jie; and Zaremba, Wojciech. 2016. *OpenAI Gym*. Cited on p. 30.

CartPole v0. 2019. *CartPole v0 – OpenAI Github Wiki*. URL https://github.com/openai/gym/wiki/CartPole-v0. Cited on p. 30.

Dinh, Laurent; Pascanu, Razvan; Bengio, Samy; and Bengio, Yoshua. 2017. *Sharp minima can generalize for deep nets*. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 1019–1028. JMLR. org. Cited on p. 15.

Falkner, Stefan; Klein, Aaron; and Hutter, Frank. 2018. *Bohb: Robust and efficient hyperparameter optimization at scale*. In arXiv preprint arXiv:1807.01774. Cited on pp. vi, 22, 23, 24, and 25.

Fawcett, Chris and Hoos, Holger H. 2016. *Analysing differences between algorithm configurations through ablation*. In Journal of Heuristics, vol. 22, no. 4, pp. 431–458. Cited on p. 33.

Fortunato, Meire; Azar, Mohammad Gheshlaghi; Piot, Bilal; Menick, Jacob; Osband, Ian; Graves, Alex; Mnih, Vlad; Munos, Remi; Hassabis, Demis; Pietquin, Olivier; et al. 2017. *Noisy networks for exploration*. In arXiv preprint arXiv:1706.10295. Cited on p. 20.

Goodfellow, Ian; Bengio, Yoshua; and Courville, Aaron. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org. Cited on p. 7.

Heess, Nicolas; Sriram, Srinivasan; Lemmon, Jay; Merel, Josh; Wayne, Greg; Tassa, Yuval; Erez, Tom; Wang, Ziyu; Eslami, SM; Riedmiller, Martin; et al. 2017. *Emergence of locomotion behaviours in rich environments*. In arXiv preprint arXiv:1707.02286. Cited on p. 2.

Henderson, Peter; Islam, Riashat; Bachman, Philip; Pineau, Joelle; Precup, Doina; and Meger, David. 2018a. *Deep reinforcement learning that matters*. In Thirty-Second AAAI Conference on Artificial Intelligence. Cited on pp. vi, 2, 3, 4, 17, 21, 22, 29, 39, and 40.

Henderson, Peter; Romoff, Joshua; and Pineau, Joelle. 2018b. *Where did my optimum go?: An empirical analysis of gradient descent optimization* *in policy gradient methods*. In arXiv preprint arXiv:1810.02525. Cited on pp. v, 15, 16, 17, and 21.

Hessel, Matteo; Modayil, Joseph; Van Hasselt, Hado; Schaul, Tom; Ostrovski, Georg; Dabney, Will; Horgan, Dan; Piot, Bilal; Azar, Mohammad; and Silver, David. 2018. *Rainbow: Combining improvements in deep reinforcement learning*. In Thirty-Second AAAI Conference on Artificial Intelligence. Cited on p. 20.

Heusel, Martin; Ramsauer, Hubert; Unterthiner, Thomas; Nessler, Bernhard; and Hochreiter, Sepp. 2017. *Gans trained by a two time-scale update rule converge to a local nash equilibrium*. In Advances in Neural Information Processing Systems, pp. 6626–6637. Cited on pp. 14 and 21.

Hochreiter, Sepp and Schmidhuber, Jürgen. 1997. *Flat minima*. In Neural Computation, vol. 9, no. 1, pp. 1–42. Cited on pp. 15 and 21.

Hooker, Giles. 2007. *Generalized functional anova diagnostics for high-dimensional functions of dependent variables*. In Journal of Computational and Graphical Statistics, vol. 16, no. 3, pp. 709–732. Cited on p. 33.

Howard, Ronald A. 1960. *Dynamic Programming and Markov Processes*. In . Cited on pp. 9 and 11.

Huang, Jianhua Z et al. 1998. *Projection estimation in multiple regression with application to functional ANOVA models*. In The annals of statistics, vol. 26, no. 1, pp. 242–272. Cited on p. 33.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. June 2014. *An Efficient Approach for Assessing Hyperparameter Importance*. In Proceedings of International Conference on Machine Learning 2014 (ICML 2014), p. 754–762. Cited on p. 33.

Hutter, Frank; Kotthoff, Lars; and Vanschoren, Joaquin, eds. 2018. *Automated Machine Learning: Methods, Systems, Challenges*. Springer. In press, available at http://automl.org/book. Cited on pp. v, 22, and 23.

Karpathy, Andrej. 2017. *Medium*. URL https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106. Cited on p. 14.

Keskar, Nitish Shirish; Mudigere, Dheevatsa; Nocedal, Jorge; Smelyanskiy, Mikhail; and Tang, Ping Tak Peter. 2016. *On large-batch training for deep learning: Generalization gap and sharp minima*. In arXiv preprint arXiv:1609.04836. Cited on pp. 8, 15, and 21.

Kingma, Diederik P and Ba, Jimmy. 2014. *Adam: A method for stochastic optimization*. In arXiv preprint arXiv:1412.6980. Cited on pp. 13, 14, 15, 29, and 63.

Krogh, Anders and Hertz, John A. 1992. *A simple weight decay can improve generalization.* In Advances in neural information processing systems, pp. 950–957. Cited on p. 15.

Li, Lisha; Jamieson, Kevin; DeSalvo, Giulia; Rostamizadeh, Afshin; and Talwalkar, Ameet. 2016. *Hyperband: A novel bandit-based approach to hyperparameter optimization.* In arXiv preprint arXiv:1603.06560. Cited on p. 22.

Liang, Eric; Liaw, Richard; Moritz, Philipp; Nishihara, Robert; Fox, Roy; Goldberg, Ken; Gonzalez, Joseph E; Jordan, Michael I; and Stoica, Ion. 2017. *RLlib: Abstractions for distributed reinforcement learning.* In arXiv preprint arXiv:1712.09381. Cited on p. 29.

Liaw, Richard; Liang, Eric; Nishihara, Robert; Moritz, Philipp; Gonzalez, Joseph E; and Stoica, Ion. 2018. *Tune: A Research Platform for Distributed Model Selection and Training.* In arXiv preprint arXiv:1807.05118. Cited on p. 31.

Loshchilov, Ilya and Hutter, Frank. 2017. *Fixing weight decay regularization in adam.* In arXiv preprint arXiv:1711.05101. Cited on pp. v, 14, 15, 16, 17, 18, and 29.

Mnih, Volodymyr; Kavukcuoglu, Koray; Silver, David; Graves, Alex; Antonoglou, Ioannis; Wierstra, Daan; and Riedmiller, Martin. 2013. *Playing atari with deep reinforcement learning.* In arXiv preprint arXiv:1312.5602. Cited on pp. 2, 12, 18, and 19.

Moritz, Philipp; Nishihara, Robert; Wang, Stephanie; Tumanov, Alexey; Liaw, Richard; Liang, Eric; Elibol, Melih; Yang, Zongheng; Paul, William; Jordan, Michael I; et al. 2018. *Ray: A distributed framework for emerging {AI} applications.* In 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), pp. 561–577. Cited on p. 29.

Ng, Andrew Y; Coates, Adam; Diel, Mark; Ganapathi, Varun; Schulte, Jamie; Tse, Ben; Berger, Eric; and Liang, Eric. 2006. *Autonomous inverted helicopter flight via reinforcement learning.* In Experimental Robotics IX, pp. 363–372. Springer. Cited on p. 2.

Nishihara, Robert; Moritz, Philipp; Wang, Stephanie; Tumanov, Alexey; Paul, William; Schleier-Smith, Johann; Liaw, Richard; Niknami, Mehrdad; Jordan, Michael I; and Stoica, Ion. 2017. *Real-time machine learning: The missing pieces.* In Proceedings of the 16th Workshop on Hot Topics in Operating Systems, pp. 106–110. ACM. Cited on p. 30.

OpenAI; Andrychowicz, Marcin; Baker, Bowen; Chociej, Maciek; Józefowicz, Rafal; McGrew, Bob; Pachocki, Jakub W.; Pachocki, Jakub; Petron, Arthur; Plappert, Matthias; Powell, Glenn; Ray, Alex; Schneider, Jonas; Sidor, Szymon; Tobin, Josh; Welinder, Peter; Weng, Lilian;

and Zaremba, Wojciech. 2018. *Learning Dexterous In-Hand Manipulation*. In CoRR, vol. abs/1808.00177. URL http://arxiv.org/abs/1808.00177. Cited on p. 2.

Reddi, Sashank J.; Kale, Satyen; and Kumar, Sanjiv. 2018. *On the Convergence of Adam and Beyond*. In International Conference on Learning Representations. URL https://openreview.net/forum?id=ryQu7f-RZ. Cited on p. 14.

Schaul, Tom; Quan, John; Antonoglou, Ioannis; and Silver, David. 2015. *Prioritized experience replay*. In arXiv preprint arXiv:1511.05952. Cited on p. 20.

Silver, David; Huang, Aja; Maddison, Chris J; Guez, Arthur; Sifre, Laurent; Van Den Driessche, George; Schrittwieser, Julian; Antonoglou, Ioannis; Panneershelvam, Veda; Lanctot, Marc; et al. 2016. *Mastering the game of Go with deep neural networks and tree search*. In nature, vol. 529, no. 7587, p. 484. Cited on p. 2.

Silver, David; Schrittwieser, Julian; Simonyan, Karen; Antonoglou, Ioannis; Huang, Aja; Guez, Arthur; Hubert, Thomas; Baker, Lucas; Lai, Matthew; Bolton, Adrian; et al. 2017. *Mastering the game of go without human knowledge*. In Nature, vol. 550, no. 7676, p. 354. Cited on p. 2.

Statt, Nick. 2019. *The Verge*. URL https://www.theverge.com/2019/4/13/18309459/openai-five-dota-2-finals-ai-bot-competition-og-e-sports-the-international-champion. Cited on p. 2.

Sutton, Richard S and Barto, Andrew G. 2018. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, USA, 2nd edn. ISBN 978-0-26203-924-6. Cited on pp. v, 1, and 7.

Tesauro, Gerald. 1995. *Temporal difference learning and TD-Gammon*. In Communications of the ACM, vol. 38, no. 3, pp. 58–69. Cited on p. 2.

Tieleman, Tijmen and Hinton, Geoffrey. 2012. *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*. In COURSERA: Neural networks for machine learning, vol. 4, no. 2, pp. 26–31. Cited on p. 13.

Tufte, Edward R. may 2001. *The Visual Display of Quantitative Information*. Graphics Press LLC, Cheshire, CT, USA, 2nd edn. ISBN 0-961-39214-2. Cited on p. d.

———. jul 2006. *Beautiful Evidence*. Graphics Press LLC, Cheshire, CT, USA. ISBN 0-961-39217-7. Cited on p. d.

Van Hasselt, Hado; Guez, Arthur; and Silver, David. 2016. *Deep reinforcement learning with double q-learning*. In Thirtieth AAAI Conference on Artificial Intelligence. Cited on p. 20.

Wang, Ziyu; Schaul, Tom; Hessel, Matteo; Van Hasselt, Hado; Lanctot, Marc; and De Freitas, Nando. 2015. *Dueling network architectures for deep reinforcement learning.* In arXiv preprint arXiv:1511.06581. Cited on p. 20.

Watkins, Christopher JCH and Dayan, Peter. 1992. *Q-learning.* In Machine learning, vol. 8, no. 3-4, pp. 279–292. Cited on p. 11.

Wilson, Ashia C; Roelofs, Rebecca; Stern, Mitchell; Srebro, Nati; and Recht, Benjamin. 2017. *The marginal value of adaptive gradient methods in machine learning.* In Advances in Neural Information Processing Systems, pp. 4148–4158. Cited on p. 14.

Yang, Zhuoran; Xie, Yuchen; and Wang, Zhaoran. 2019. *A Theoretical Analysis of Deep Q-Learning.* In CoRR, vol. abs/1901.00137. URL http://arxiv.org/abs/1901.00137. Cited on pp. 19 and 20.

Zhang, Guodong; Wang, Chaoqi; Xu, Bowen; and Grosse, Roger B. 2018. *Three Mechanisms of Weight Decay Regularization.* In CoRR, vol. abs/1810.12281. URL http://arxiv.org/abs/1810.12281. Cited on p. 17.

INDEX

APPENDICES

THE ROLE OF EPSILON IN ADAM

A

"In theory there is no difference between theory and practice; in practice there is."

— Anonymous

As discussed in section 4.3, we deemed it absurd from a theoretical standpoint to even tune the ϵ hyperparameter of the Adam optimizer at all. Recall from line 8 of algorithm 1 that ϵ is specified as a mere small constant > 0 to numerically stabilize the division by $\hat{v}$ in the parameter update. We argued that this hyperparameter should not be optimized without clear reasoning to foster the comparability across publications. We will now present what we think are the practical reasons for the tuning and give a quick discussion of the issue.

Note, that in the following we will no longer explicitly notate vectors as such.

Recall the parameter update rule for Adam [1]:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) \tag{A.1}$$

1. We dropped the optional schedule multiplier η_t for simplicity.

Upper Bounding the Maximum Step-Size In the original formulation of Adam, Kingma and Ba (2014) stated, that assuming $\epsilon = 0$ the effective step size $\Delta_{\theta_t} = \alpha \cdot \hat{m}_t / \sqrt{\hat{m}}$ of the parameter update is subject to an upper bound of

$$|\Delta_{\theta_t}| \le \alpha \cdot (1 - \beta_1)/\sqrt{1 - \beta_2} \quad , \text{if } (1 - \beta_1) > \sqrt{1 - \beta_2} \tag{A.2}$$

$$|\Delta_{\theta_t}| \le \alpha, \quad \text{otherwise} \tag{A.3}$$

where the first case only arises if a gradient has been zero at all prior time steps but not the current one.

Hence, usually $|\Delta_{\theta_t}| \le \alpha$. Thus α can be used to control the maximum step size.

The Influence of Epsilon Now, not fixing $\epsilon = 0$ means that this upper bound is also affected by the selection of ϵ and not just α. Both ϵ and α rescale all steps and not only control the upper bound and due to the nonlinear influence, there is no general rescaling of either to compensate for the other. Two update steps for different α and ϵ are only equal if

$$\alpha_1 / (\sqrt{\hat{v}_t} + \epsilon_1) = \alpha_2 / (\sqrt{\hat{v}_t} + \epsilon_2) \Leftrightarrow \frac{\alpha_1}{\alpha_2} = \frac{\sqrt{\hat{v}_m} + \epsilon_2}{\sqrt{\hat{v}_m} + \epsilon_1} \tag{A.4}$$

which is dependent on $\hat{v}_m$. Hence, jointly optimizing both parameters can practically yield better results, for more variable step sizes are possible. This seems to practically improve the performance, as also found by Bellemare et al. (2017).

Specifically, if the moment estimates are small, a larger ϵ will shrink the effective steps size more strongly, while having a much smaller impact for larger moments. Thus, a larger epsilon will slow down the optimizer in flatter terrain, while having a smaller influence in rougher terrain. Depending on the magnitude of the variation of ϵ, this could have an influence on the smoothness of the optima that are preferred by Adam.

Update to new specifics

The Counterargument Against these empirical results stand our apprehension that the tuning of this additional hyperparameter beyond its theoretical intention, increases the risk of overfitting, especially in the notoriously noisy field of reinforcement learning. It thus seems necessary to rigorously evaluate 1. the influence of ϵ on performance and overfitting, 2. the parameter importance and cross-correlation with the α-parameter.

if time: investigate parameter importance and correlation of epsilon and alpha

QUICK START GUIDE

B

In this chapter, we give a short introduction to how to create our results and use our code base for further experiments.

The guide is written for Linux users, but as all used tools are cross-platform, it can also be applied to Windows and MacOS. Some command line commands might slightly differ though. Standard command line tools, such as *git* are required.

B.1 ENVIRONMENT SETUP

Clone the accompanying repository and enter the folder.

```
$ git clone git@github.com:vonHartz/bachelorthesis-public.git
$ cd bachelorthesis-public
```

We recommend to use *Conda* or a similar software for managing Python environment. To install it, visit conda.io/en/latest/miniconda.html and download the latest Python 3 installer for your operating system.

Having installed Conda, create a new environment - we name it *rl_env* - and automatically set it up via Make.

```
$ conda create -n rl_env python=3.6.8
$ conda activate rl_env
$ make inst_dev
```

That will automatically install all packages listed in `requirements.txt` and install our code base as a development package. If you cannot use GNU Make, just manually execute the steps listed in the `Makefile` for the target *inst_env* and *inst_dev*.

You can verify that everything went well, e.g. by running the following command. If you get an output indicating that an agent was successfully trained, all went well.

```
$ run-dqn-agents-main
```

B.2 REPRODUCTION OF EXPERIMENTS

First, make sure to activate the environment, just created.

```
$ conda activate rl_env
```

Within our code base, the individual experiments are implemented using *runners* with a number of arguments, which allow to customize the parameters of the experiment. Each runner can be directly accessed from the console via an *entry point*, which is automatically created upon setting up the environments. All entry points can be found in the `setup.py`. Calling a runner with the -h option will display all its parameters. E.g.

```
$ robustness-experiment-runner -h
```

The used parameter settings can be found in the job scripts in the `META` and `NEMO` folder. These furthermore contain additional steps necessary to run the experiments on a cluster using Slurm, respectively Moab.

To reproduce our results simply call the respective runner with the given arguments.

B.3 CUSTOM EXPERIMENTS

B.3.1 *Environments*

To run one of our experiments with another OpenAI Gym environment, simply pass its name to the runner. As long as it is a registered environment known to Ray, this will work.

B.3.2 *Agents*

As the workers for our experiments essentially wrap Ray agents, a custom worker can be easily be realized via inheriting from the *RllibWorker* found in `src/core/bohb_workers.py`:

```
import CUSTOM_AGENT, DEFAULT_CONFIG

class CustomWorker(RllibWorker):
  def __init__(self, **kwargs):
    super().__init__(**kwargs)
    self.agent = CUSTOM_AGENT
    self.agent_name = CUSTOM_AGENT_AGENT
    self.default_agent_config = DEFAULT_CONFIG.copy()
```

This worker can then be used in the `bohb_runner`. As for simplicity, the other experiments also use workers implementing the same interface but without the additional setup needed for BOBH. Hence, for efficiency, a worker for these experiments should inherit from the `FakeWorker` found in `src/robustness_experiment/workers.py`.

Unknown Configuration Due to the design of the experiments, in might be necessary to set *_allow_unknown_configs = True* in the agent class, as the configuration dictionary of the agent is for example also used by the BOBH worker to pass the configuration to the optimizer.

Configuration Space To actually work with BOBH thought, the worker has to also implement the following method, generating a configuration space for BOBH to sample from.[1]

1. For information on the ConfigSpace package, visit automl.github.io/ConfigSpace/master/.

```
@staticmethod
def get_configspace():
  config_space = CS.ConfigurationSpace()

  learning_rate = CSH.UniformFloatHyperparameter('lr',
      ↪ lower=1e-6, upper=1e-1, default_value='1e-3',
      ↪ log=True)
  config_space.add_hyperparameters([learning_rate])

return config_space
```

Registration of Custom Agents Furthermore, for rolling out the agent, it is necessary to add it to Rllib's registry. This can be achieved by adding the following code if the agent is not already part of Ray. The chosen name can then be passed to the worker upon construction.

```
def register_custom_agents():
  def _import_my_agent():
    return DqnAgentAdamW

  CONTRIBUTED_ALGORITHMS = {"MyAgent": _import_my_agent
      ↪ }

  ray.rllib.contrib.registry.CONTRIBUTED_ALGORITHMS =
      ↪ CONTRIBUTED_ALGORITHMS
  ray.rllib.agents.registry.CONTRIBUTED_ALGORITHMS =
      ↪ CONTRIBUTED_ALGORITHMS
```

The registration process needs to be run in each Ray instance. The BOBH worker however, will automatically do that.

B.3.3 Optimizers

To use a custom optimizer, it is easiest to define a new agent that inherits from the desired agent. The interesting part happens in what in Ray is called the *policy graph*.

Policy Graph Define a new policy graph that inherits from the one that the agent usually uses and override the function creating the optimizer, as shown in the example below. Tensorflow optimizers should work out of the box.

```
from ray.rllib.agents.dqn.dqn_policy_graph import
    ↪ DQNPolicyGraph
import tensorflow as tf

class DQNPolicyGraphAdamW(DQNPolicyGraph):
  @override(DQNPolicyGraphCustom)
  def optimizer(self):
    return tf.contrib.opt.AdamWOptimizer(
      learning_rate=self.config["lr"],
      weight_decay=self.config["weight_decay"],
      beta1=self.config["beta1"],
      beta2=self.config["beta1"],
      )
```

As you can see, the configuration of the optimizer is passed via the agents configuration.[2]

2. Hence *_allow_unknown_configs* needs to be set to true.

Logging of Additional Values For an example on how internal values of the optimizer can be tracked, see `src/core/dqn_policy_graphs.py`.

Agent Now, a custom agent can be easily be defined. Remember to define your configuration space, as shown above.

```
import CUSTOM_AGENT, DEFAULT_CONFIG,
    ↪ OPTIMIZER_SHARED_CONFIGS
import CUSTOM_POLICY_GRAPH

class CUSTOM_AGENT(DqnAgentCustom):
  _agent_name = "MzAgent"
  _default_config = DEFAULT_CONFIG
  _policy_graph = CUSTOM_POLICY_GRAPH
  _optimizer_shared_configs = OPTIMIZER_SHARED_CONFIGS

  _allow_unknown_configs = True
```

Additional Details Most other details should be lucid from the code in the repository. For further questions or to report bugs, feel free to open an issue on GitHub or to write us an email.