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, the practical application poses many complex problems beyond the pure 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 achieving - versus not achieving - 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. Whereas there exists some rough ideas, the exact effects of different optimizers and their hyperparameters on the returns of the agent, as well as the stability in training and rollout, are not yet well understood. Moreover, existing ideas and empirical results are sometimes contradictory, painting an unclear picture.

Using the well known DQN algorithm, we investigate the influence of the popular Adam optimizer and AdamW, a variation of Adam using weight decay, as well as the settings of their hyperparameters, on the agent's performance. Furthermore, we compare the optimizers' sensitivity to their hyperparameters and show how to judge their stability in training. Additionally, we investigate how to estimate the performance of a configuration on a given computational budget. Then, using BOHB, a hyperparameter optimization method, we strive to make a step towards hyperparameter agnostic deep reinforcement learning, trying to avoid the pitfalls of manual hyperparameter configuration.

ZUSAMMENFASSUNG

Im maschinellen Lenen stellen sich in der Praxis viele komplexe Probleme, die über das hinausgehen, was die bloße Theorie derzeit beantworten kann. Neben den verwendeten Algorithmen und Ausgangswerten der Zufallsgeneratoren, haben auch die verwendeten Optimierungsmethoden und deren Hyperparameter einen immensen Einfluss auf das Ergebnis - bis zur Entscheidung über Konvergenz oder Divergenz. Deep Reinforcement Learning ist auf Grund der unebenen und sich schnell ändernden Oberflächenstruktur der zu optimierenden Funktionen besonders schwierig. So bleiben die Agenten in lokalen Optima hängen anstatt das gewünschte Verhalten zu erlernen. Obwohl einige grobe Charakterisierungen formuliert wurden, sind die genauen Auswirkungen der verschiedenen Optimierungsmethoden und ihrer Hyperparameter auf das vom Agenten erzielte Ergebnis noch nicht hinreichend erforscht. Ebenso ihre Stabilität während des Trainings des Agenten, sowie die der gefundenen Optima. Ferner sind existierende Vorstellungen und empirische Ergebnisse manchmal widersprüchlich, was eine unklares Gesamtbild erzeugt.

Anhand des bekannten DQN Algorithmus untersuchen wir das Verhalten des populären Adam Optimierers nebst AdamW, einer Variante davon, welche Weight Decay einsetzt. Neben dem Einfluss auf die finale Leistung des Agenten betrachten wir dabei auch die Sensitivität der Optimierer gegenüber ihren Hyperparametern und zeigen, wie sich ihre Stabilität beurteilen lässt. Ferner untersuchen wir, wie sich die Leistung einer Konfiguration effizient schätzen lässt. Davon ausgehend versuchen wir mit der Hilfe von BOHB, einer Methode für Hyperparameter-Optimierung, einen Schritt in Richtung Hyperparameter-Agnostizismus zu machen, um die typischen Fallstricke der manuellen Abstimmung von Hyperparametern in Zukunft zu meiden.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

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

Moreover, I am grateful towards my supervisor Raghu Rajan for his help, advice and wit.

My further gratitudes goes to Devin and Nassim for being sources of confidence and inspiration. I thank Freddy and Philipp for getting me started on the cluster. My sincere appreciation to David, Lars and Panaiotis for their insightful feedback on my writings. A big thank you to everyone at the Lab for sharing bureaus, coffee, encouragement and thoughts.

And thank you to Ann-Catherine - for everything.

The author acknowledges 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).

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 for this kind of problem 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.

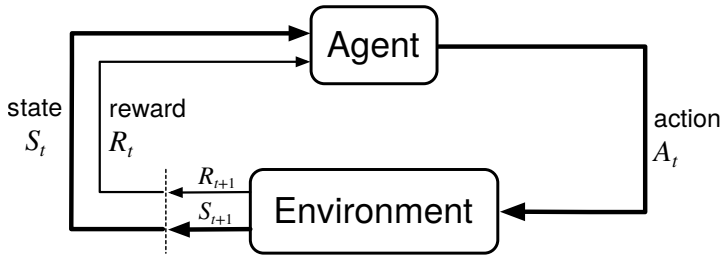


Figure 1.1: The reinforcement learning loop (Sutton and Barto, 2018, Figure 3.1). The agent observes the reward and current state of the environment and selects an action accordingly, that in turn alters the state of the environment.

1.1 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[1] 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 of 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.

1. Often on what is called a "super-human" level, generating a lot of media coverage for the field.

Arising Problems Although 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 function from a class of functions that is as well suited as possible to describe some relation in an unknown data distribution. This is usually achieved by minimizing some loss function, defined over the output of said function, for a sample of data points from the distribution. Hence, the training of a deep neural network is 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 as *gradient descent* and variations of it. Following are some of the problems that arise from this approach.

Local Optima A systematic problem of gradient-based methods is that they are generally only suited to find local, not global, optima.[2] In reinforcement learning the loss landscape is usually rough (volatile) 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 in their experiments.[3]

2. Global optima are only guaranteed to be found for convex functions.

3. See youtu.be/lKpUQYjgm8o

Trial-and-Error Reinforcement learning furthermore 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. 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. Because these estimations change drastically over time, the numerical optimization of these networks is unstable.

Methods and Hyperparameters Furthermore, it is not well understood, how different (adaptive) gradient-based methods, such as the popular Adam algorithm (Kingma and Ba, 2014) and AdamW (Loshchilov and Hutter, 2017), a variation of it, compare in reinforcement learning, especially because they are also somewhat sensitive to the settings of their hyperparameters. Thus, there is a call for the development of hyperparameter agnostic algorithms - algorithms that adapt their hyperparameters themselves during runtime (Henderson et al., 2018a).

Performance Estimation Automatic hyperparameter optimization - which is a step towards hyperparameter agnosticism - introduces further challenges, as 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 costly.[4] E.g., Faust and Francis (2019) reported training an automated reinforcement learning system on robotic navigation for the equivalent of 32 years of experience.

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

1.2 FOCUS

In this thesis, we focused on the analysis of the adaptive gradient-based optimization methods Adam and AdamW in deep reinforcement learning - reinforcement learning using deep neural networks. Furthermore, we used the well established algorithm DQN (Mnih et al., 2013) and trained the agents popular environments Cartpole (Barto et al., 1983), MountainCar (Moore, 1990) and parts of the Arcade Learning Environment (Bellemare et al., 2013). We deem it important to generalize our groundwork 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.[5]

5. However, we constructed our code base in such a way that it is easy to extend in the future.

1.3 OBJECTIVE

Performance Estimation We 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.

Towards Hyperparameter Agnosticism Then, using a hyperparameter optimization method we automatically find well performing hyperparameter settings for the optimizers to make a step towards hyperparameter agnosticism and to be able to compare both optimizers in a fair way. We furthermore take a look at the sensitivity of both optimizers towards their hyperparameters.

Comparison of Optimization Methods Finally, we compare Adam, a popular optimization method, with AdamW, an extension of it, to evaluate their respective performance and to gain an understanding of their different characteristics. These are 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. Remember the problem of **local optima** in reinforcement learning.

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

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 use their resources efficiently.

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 researchers 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 interests without the large burden that comes with the selection of optimization methods and their hyperparameters.

1.5 RELATED WORK

Heusel et al. (2017) empirically evaluated Adam for generative adversarial networks and formulated the idea of Adam as a "heavy ball with friction" to describe Adam's preference for flatter minima. Henderson et al. (2018b) compared different optimizers, including Adam, in a reinforcement learning setting and showed their sensitivity to the setting of the learning rate. They further showed the influence of the momentum factor for the example of stochastic gradient descent with Nesterov momentum (Nesterov, 1983).

In their paper proposing BOHB, Falkner et al. (2018) evaluated this hyperparameter optimization method by tuning the hyperparameters of the PPO algorithm (Schulman et al., 2017). Springenberg et al. (2016) tuned the hyperparameters of DDPG (Lillicrap et al., 2015) in Cartpole. Jaderberg et al. (2017) evaluated their *population based training* method on multiple challenging reinforcement learning problems.

1.6 OUTLINE

This thesis consists of two 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] The second part presents the three main problems that we tackled in our work. First, the problem of performance estimation in deep reinforcement learning and the experiments we conducted. Second, the automatic optimization of hyperparameters of the used optimizers. And third, a detailed comparison of the two optimizers.

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

8. This approach is motivated by Henderson et al. (2018a), who showed the huge influence of the code base on the final performance in deep reinforcement learning.

PART I

BACKGROUND

KEY CONCEPTS AND NOTATION

2

2.1 NOTATION

Whenever possible, we distinguish between different mathematical objects via the use of different symbols, see Table 2.1.

Object	Symbol	Modifiers	
scalar	x	-	
vector	$\vec{x}$	x_i	i-th element
matrix	X	$\vec{x}_i$	i-th column
		x_{ij}	j-th value of the i-th column
set	$\mathcal{X}$	x_i	i-th element
(time) series	$(\chi_n)_{n=0,1,\dots}$	χ_t	element at point t
function	f	$\hat{f}$	approximation of f

Table 2.1: Naming of common mathematical objects.

Table 2.2 gives a summary of commonly used symbols and variables.

Symbol	Meaning	Defined in
$a, \mathcal{A}$	action, action space	§ 3.1.1 (p. 11)
$s, \mathcal{S}$	state, state space	
$R(s,a)$	reward function	
$\mathcal{R}(s,a)$	expected immediate reward	
$\mathcal{P}$	state transition function	
γ	discount factor	
$p_i, \vec{\theta}$	parameter parameter vector	§ 2.2 (p. 8),
Θ_i, Θ	domain of p_i, joint domain of θ	§ 4.4 (p. 30)
$\hat{f}(\cdot;\vec{\theta})$	approximation of f with parameters $\vec{\theta}$	§ 2.2 (p. 8)
π	policy	§ 3.1.1 (p. 11)
Q	state-action value function	§ 3.1.2 (p. 12)

Table 2.2: Variables and symbols commonly used throughout the thesis.

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. Next, we will 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 $\mathcal{G} : \mathbb{R}^d \to \mathbb{R}^l$ and a matrix of training data $X \in \mathbb{R}^{d \times n}$ with targets $Y \in \mathbb{R}^{l \times n}$, 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** $L : \mathbb{R}^{l \times l} \to \mathbb{R}$ 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 domain Θ and numerically optimizing these parameters.

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 $f : \mathbb{R}^n \to \mathbb{R}$ 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.

Higher Order Optimization Methods and Momentum Numerical optimization can be further sped up by not only using the gradient to determine the step size but also using higher order moments. As these are usually computationally expensive to calculate, a lot of optimization

methods estimate them, e.g. via rolling averages of the gradient, called *momentum*.

(Mini-) Batch 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). On the other hand, the computation on a single data point leads to noisy estimates of the gradients in numerical optimization. Hence, 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.

Regularization A common approach to combat overfitting is *regularization*. L_2-regularization (Krogh and Hertz, 1992) is a popular example, adding a term $\lambda \cdot \sum_i \theta_i$ to the loss function, in effect shrinking the parameter vector θ during numerical optimization. The parameter $\lambda \in \mathbb{R}$ controls the regularization strength. Another technique is *weight decay*: directly shrinking $\vec{\theta}$ in each optimization step by multiplying it with λ.

(Deep) Neural Networks are non-linear function approximators commonly used in recent machine learning research. Their most common representatives - **multilayer perceptrons** - 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*.

Hyperparameter Optimization Machine learning methods are often sensitive towards the settings of their hyperparameters. For example a too large learning rate can make the numerical optimization of a function approximator overshoot minima and lead to divergence. Hence, it is important to properly tune these hyperparameters. The *automatic* hyperparameter optimization is thus an active research field.

REINFORCEMENT LEARNING

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 theory behind their practical application.

3

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 $\tau = 0, 1, \ldots$ 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 $\mathbf{P}[s_t \mid s_{t-1}, s_{t-2}, \ldots, s_0] = \mathbf{P}[s_t \mid s_{t-1}]$.* [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 $r \in \mathbb{R}$ and state $s \in \mathcal{S}_M$. 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 $h \in \mathbb{N} \cup \{\infty\}$ is called the *horizon* of the problem.[5]

5. The horizon can be infinite.

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

7. E.g. "game over" in video games, that 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* state-value function.

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

This famous property allows to iteratively compute the Q-function or V-function for a finite state space, a finite action space and a given policy, given the reward function R.

Remember, that the goal for the agent is to learn the optimal policy $\pi^\star$, which corresponds to an optimal *state-action-value function* $Q^\star$.

Theorem 2 (Bellmann Optimality Equation) *For the* optimal policy $\pi^\star$ *the* state-action-value function $Q^\star$ *is given by:* [9]

$$Q^\star(s,a) = \sum_{s'} P_a(s,s') \cdot \left(R(s,a) + \gamma \cdot \max_{a'} Q^\star(s',a') \right)$$

Value Iteration Due to this property, one can compute the optimal Q-function $Q^\star$ and extract the optimal policy $\pi^\star$ from it. As the Bellman equation illustrates, maximizing Q under policy π is achieved by greedily selecting the action a in state s that maximizes

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

Hence, to calculate $Q^\star$ one solely has to initialize all values (e.g. as zero) and then repeatedly maximize the Bellman equation over all actions for each state until the values converge (Bellman, 1957). This is called *value iteration*. The optimal policy $\pi^\star$ is then given by:

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

Policy Iteration Just as $Q^\star$ can iteratively be computed, so can $\pi^\star$ be directly as well, e.g. via *dynamic programming* (Howard, 1960). As the agent only needs to know $\pi^\star$ and not necessarily $Q^\star$, whereas $\pi^\star$ often converges sooner than $Q^\star$, *policy iteration* is computationally more efficient than value iteration.

For this purpose, one can initialize a random policy π. Alternating between computing Q^π, like in value iteration, and then updating π to greedily maximize Q^π, improves the policy π. Repeating this process until π converges, directly gives $\pi^\star$. Q^π does not have to be converged.

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: *value based methods* and *actor-critic methods*.

3.2.1 Value Based and Actor-Critic Methods

Valued Based Methods From value iteration follows how an agent A can learn an approximation to the optimal policy $\pi^\star$ in an environment M by estimating $Q^\star$. It can 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 $Q^\star$. This technique is at

9. Again, a similar property holds for the state-value function.

the heart of so called *value based methods*, such as *Q-learning* (Watkins and Dayan, 1992).

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. Methods that works this way are called *actor-critic methods*. As in each time step they estimate the optimal Q-function and the optimal policy separately, they are more akin to policy iteration.

10. Greedy selection connotes selecting the option that at the current point in time has the best value by some metric. Here; the best Q-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. 12)) . 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.

Exploration-Exploitation Trade-Off The primary target of an agent is to maximizes its reward. However, to do so it needs a decent estimate of Q^*, which requires exploring the action space. Hence, the agent has to engage in a trade-off between taking the greedily selected action and exploring new actions to improve its estimate of Q^*. This is called the *exploration-exploitation trade-off*.

3.2.2 Q-Learning and Deep Reinforcement Learning

Q-learning works on finite action and state spaces and estimates the Q-function via a simple look-up table. It then selects an action using an **ϵ-greedy strategy**. This means that it picks a random action with some small probability ϵ and the action with the currently highest Q-value otherwise. Employing such a strategy is a common approach to solve the exploration-exploitation trade-off.

Deep Reinforcement Learning As look-up tables are not memory-efficient for large action or state spaces, more recent algorithms use function approximators instead. A popular class of non-linear function approximators are *deep neural networks*. Standing to reason is replacing the look-up table in Q-learning with a neural network, giving us DQN (Mnih et al., 2013), which we will introduce in the next chapter.

Actor-critic methods for example might also approximate the policy via a deep neural network.

3.2.3 *Practicality of the Markov Property*

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 *partially observable*.

Reinforcement Learning Praxis As noted by Arulkumaran et al. (2017), the underlying assumption of the *Markov Property* does often not hold in practical applications, because it requires full observability of the states. Although 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 emulator, 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.

3.2.4 *Scope of this Thesis*

In this thesis we will examine DQN, a *value based deep reinforcement learning* algorithm. However, our approach and code base can easily be extended to also investigate *actor-critic methods* and others as well.

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

ALGORITHMS AND METHODS

4

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*, as gradients can be easily computed for deep neural networks, whereas analytical solutions are infeasible and higher order moments 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. 18). 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. It then updates its moment estimates and uses these 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 $\alpha \in \mathbb{R}$, decay rates $\beta_1, \beta_2 \in \mathbb{R}$, division stabilization constant $\epsilon > 0$

Given: initial parameter vector $\vec{\theta}_0 \in \mathbb{R}$, 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: $t \leftarrow 0$; $\vec{m}_0, \vec{v}_o \leftarrow \vec{0}$; `(time step and moment vectors)`

1 **while** *stopping criterion not met* **do**

2 $\quad t \leftarrow t + 1$;

3 $\quad \vec{g}_t \leftarrow \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1})$; `(computed on minibatch` $\vec{m}$`)`

4 $\quad \vec{m}_t \leftarrow \beta_1 \cdot \vec{m}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t$; `(moment estimation)`

5 $\quad \vec{v}_t \leftarrow \beta_2 \cdot \vec{v}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t^2$;

6 $\quad \hat{\vec{m}}_t \leftarrow \vec{m}_t / (1 - \beta_1^t)$; `(zero-bias correction)`

7 $\quad \hat{\vec{v}}_t \leftarrow \vec{v}_t / (1 - \beta_2^t)$;

8 $\quad \vec{\theta}_t \leftarrow \vec{\theta}_{t-1} - \eta_t \cdot \alpha \cdot \hat{\vec{m}}_t / (\sqrt{\hat{\vec{v}}_t} + \epsilon)$; `(parameter update)`

9 **end**

10 **return** θ_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. Another extension *AdamW* was put forward by Loshchilov and Hutter (2017), adding the popular *weight decay* regularization method to Adam.

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 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 § 1.1 (p. 1) 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. Although both are equivalent for stochastic gradient descent, they are not for Adam. And whereas naive implementations often just apply L_2-regularization, Adam benefits far more from weight decay.

4.1.2 AdamW: Adam with Weight Decay

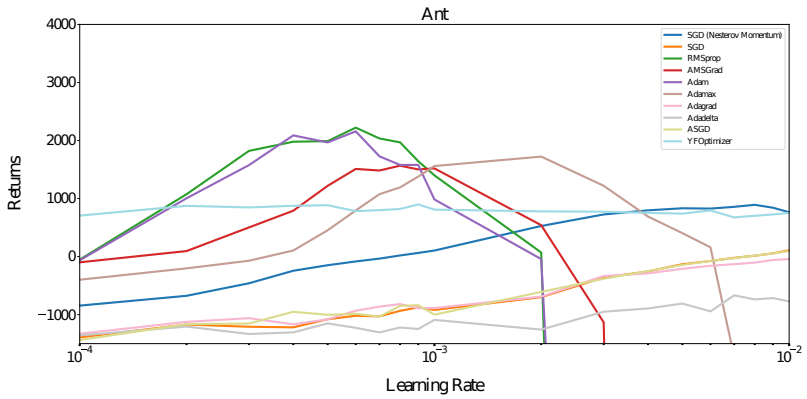
Method

Whereas 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

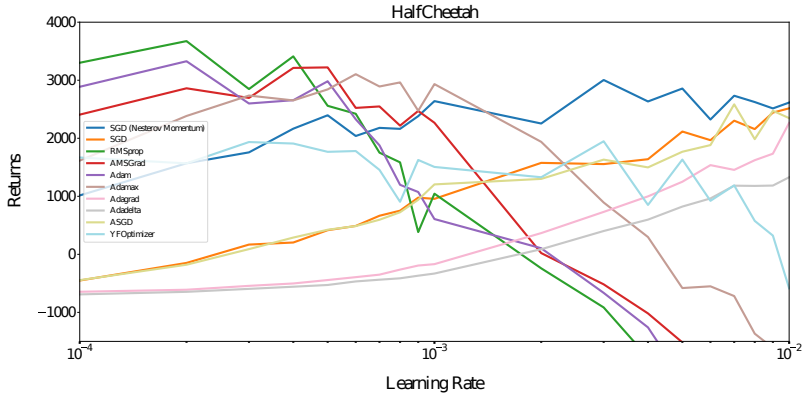
$$\vec{\theta}_t = \vec{\theta}_{t-1} - \alpha \cdot \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1}) \tag{4.1}$$

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:

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



(a) A2C (Mnih et al., 2016) agent in Ant (Schulman et al., 2015b) environment.



(b) TRPO (Schulman et al., 2015a) agent in Half-Cheetah (Tassa et al., 2018) 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, whereas falling apart for divergent settings.

Combining this technique with the Adam algorithm (see algorithm Algorithm 1 (p. 18)), gives algorithm Algorithm 2 (p. 21), called *AdamW* by Loshchilov and Hutter (2017). The authors gave proof that although 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. 18) 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. Because 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).

Hyperparameters: step size $\alpha \in \mathbb{R}$, decay rates $\beta_1, \beta_2 \in \mathbb{R}$,
division stabilization constant $\epsilon > 0$,
weight decay factor $\lambda \in [0, 1)$
Given: initial parameter vector $\vec{\theta}_0 \in \mathbb{R}$,
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: $t \leftarrow 0$; $\vec{m}_0, \vec{v}_o \leftarrow \vec{0}$; `(time step and moment vectors)`

while *stopping criterion not met* **do**
- $t \leftarrow t + 1$;
- $\vec{g}_t \leftarrow \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1})$; `(computed on minibatch` $\vec{m}$`)`
- $\vec{m}_t \leftarrow \beta_1 \cdot \vec{m}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t$; `(moment estimation)`
- $\vec{v}_t \leftarrow \beta_2 \cdot \vec{v}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t^2$;
- $\hat{\vec{m}}_t \leftarrow \vec{m}_t / (1 - \beta_1^t)$; `(zero-bias correction)`
- $\hat{\vec{v}}_t \leftarrow \vec{v}_t / (1 - \beta_2^t)$;
- $\vec{\theta}_t \leftarrow \boxed{(1 - \lambda)} \cdot \vec{\theta}_{t-1} - \eta_t \cdot \alpha \cdot \hat{\vec{m}}_t / (\sqrt{\hat{\vec{v}}_t} + \epsilon)$; `(parameter update)`

end

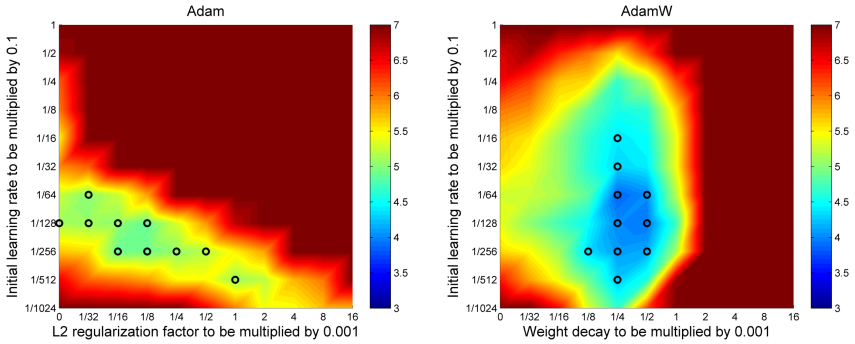
return θ_t

Algorithm 2: AdamW. Yielded by combining Adam (see algorithm Algorithm 1 (p. 18)) with weight decay (see Equation 4.2 (p. 19)).

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, however still allowing for 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 ω_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.

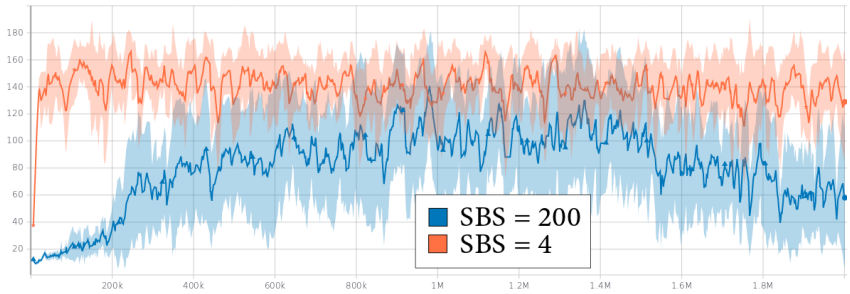


Figure 4.3: The mean reward and standard deviation of two groups of 10 trials each with weakly correlated and strongly correlated samples. The DQN agents were trained in Cartpole for 2 million time steps. Correlating the samples was achieved by increasing the *sample batch size* - the number of samples with which the replay buffer is updated. This led to it sampling less diverse, hence correlated, transitions for the gradient updates. Note, that the more diverse samples do not only help the agent to learn in the beginning, but lead to an improved overall performance and less variance.

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 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 based on sampled transitions. On the other hand, the classical value iteration algorithm, that inspires value-based methods, updates its estimate $Q^\star$ using all possible transitions from each state. 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 $\vec{\theta}$.

The importance of this technique is illustrated in Figure 4.3. The figure illustrates the effects of a large *sample batch size* on the learning curves. When the replay buffer is updated with more samples at once, these samples depict a less diverse set of experiences. As a consequence, the randomly drawn samples have a higher correlation, what it turn worsens the agent's learning progress.

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, two competing goals emerge: the explo-

ration 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, whereas else taking the action currently favored by the greedy selection.

Convergence Note, that although 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. 12) 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 policy and the off-policy probability. As we demonstrate next, this can be used to calculate the expected squared error between a function (here a neural network) and the current estimation of it.

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

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):

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

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):

3. See how these targets echo the Bellman equation.

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

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

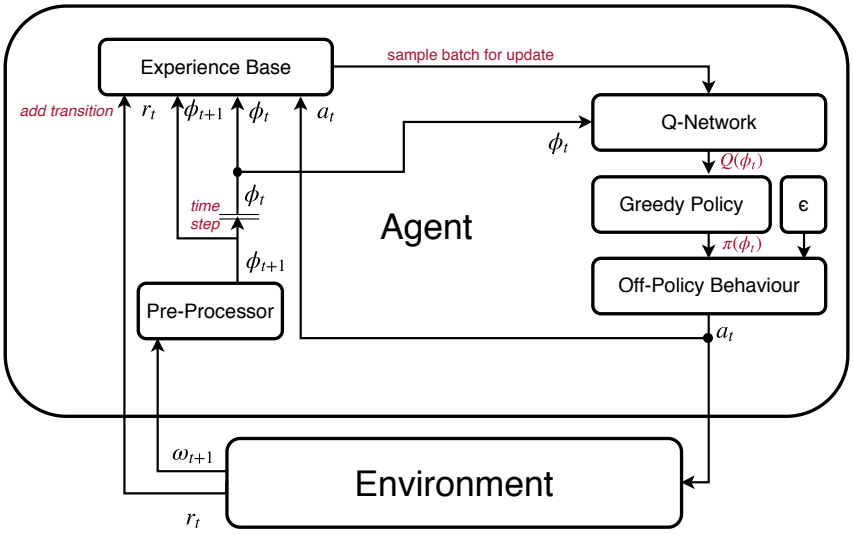


Figure 4.4: DQN. In each time step, the current policy is computed via greedy selection from the output vector of the Q-network and the off-policy probability ϵ. The new observations gets pre-processed and is combined with the old observation, the taken action and the received reward into a transition, which is added to the replay memory. Samples from the replay memory are used to update the Q-network, following Equation 4.7 (p. 25).

$(\phi_i, a_i, r_i, \phi'_i)$, we get (assuming $\phi_i \equiv s_i$):

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

Algorithm 3 (p. 32) shows how this estimation of the Q-function is combined with experience replay and the off-policy learning to form DQN. Figure 4.4 visualizes the process.

Extensions

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].

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.

4.3 HYPERPARAMETER OPTIMIZATION

4.3.1 Motivation

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, especially if only the best runs are reported (Henderson et al., 2018a).

Even more fundamental, to us there seems to be a lack of understanding about the effects of hyperparameters. Sentences like the following are not uncommon and illustrate the problem. "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 theoretical understanding and practical application. As the alert reader will remember, in Adam ϵ is intended as 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 this case.[7]

7. The interested reader can find a résumé about the influence of ϵ and a quick discussion of the tuning of this hyperparameter in Appendix A (p. 103).

Yet, our argument should not be mistaken for a direct critique of the authors. If tuning this hyperparameter has an effect on the performance, it makes perfect sense to do it. The lack of an explanation for an effect does not mitigate the effect's existence and it is science's manifest fate to chase and model phenomena only after they have been observed. We merely call for a deeper understanding of these effects - that are practically exploited - in order to improve the comparability of results.

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. For our experiments we used the hyperparameter optimization algorithm BOHB, which we will introduce next.

4.3.2 BOHB: Bayesian Optimization with Hyperband

In hyperparameter optimization there are two well known approaches: *Bayesian optimization* and *bandit-based methods*, e.g. *Hyperband*. Both have strengths and weaknesses that we will introduce next, as well as how *BOHB* combines the strengths of both. For we will only introduce all methods 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 the final performance by first creating a *prior probability distribution* - expressing the belief in the behavior of the function without any evidence. This is then updated by fitting it to data that is produced by evaluating hyperparameter settings that are generated by maximizing an *acquisition function*.[8] The process is summarized in Figure 4.5.

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 bandit-based methods, such as *Hyperband*, are usually quicker to find good settings.

Hyperband

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

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

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.

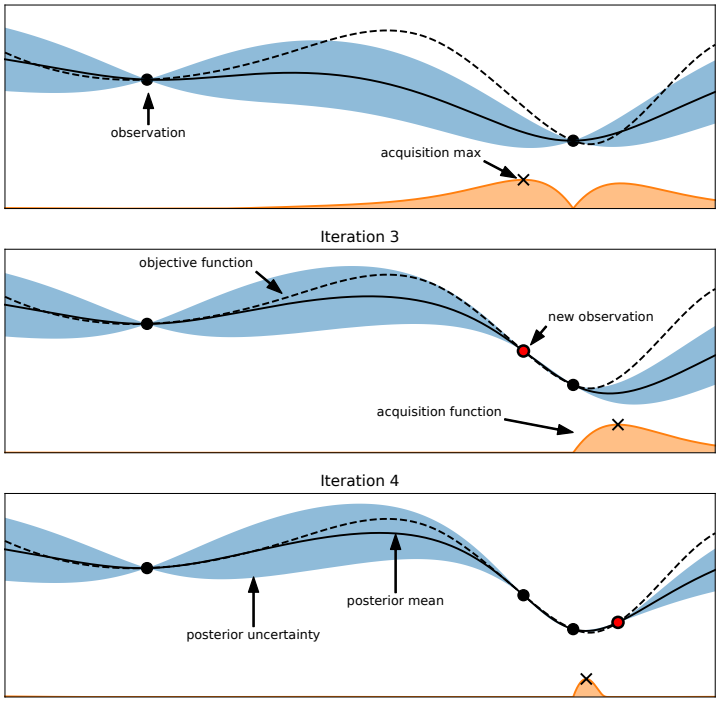


Figure 4.5: Illustration of Bayesian optimization for a one-dimensional function (Hutter et al., 2018, Figure 1.2). The dashed line is the objective function, the black line its current estimate and the blue area the confidence interval around it. The orange line and the area beneath it, depict the acquisition function: its values are 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.

Then, in each round it starts by sampling a number n of random configurations and evaluating them on the lowest round budget $b = b_{min}$. It proceeds iteratively by selecting 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 in addition to being flexible and scalable, but often having a worse final performance, especially for large budgets.

BOHB: Combining Hyperband and Bayesian Optimization

BOHB combines the strong final performance of Bayesian methods with the better anytime 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), $D = \{(\vec{x}_o, y_o), \dots (\vec{x}_n, y_n)\}$, BOHB 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 $\alpha := \min\{y_i \mid i \in \{0, \dots, n\}\}$:

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

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 BOHB 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. 33) summarizes a simplified version of BOHB. For each *bracket* [11] s, a number n_{configs} of configurations are sampled that then all go through the process of *successive halving* [12] - 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.

For comprehensibility, we simplified the sampling process of BOHB in Algorithm 4 (p. 33). 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.[13]

Fortes and Flaws Falkner et al. (2018) found that BOHB combines the strong final performance of Bayesian methods with the better any-time performance of Hyperband. Figure 4.6 illustrates this characteristic

11. *Brackets* are equivalent to the *rounds* in Hyperband.

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

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

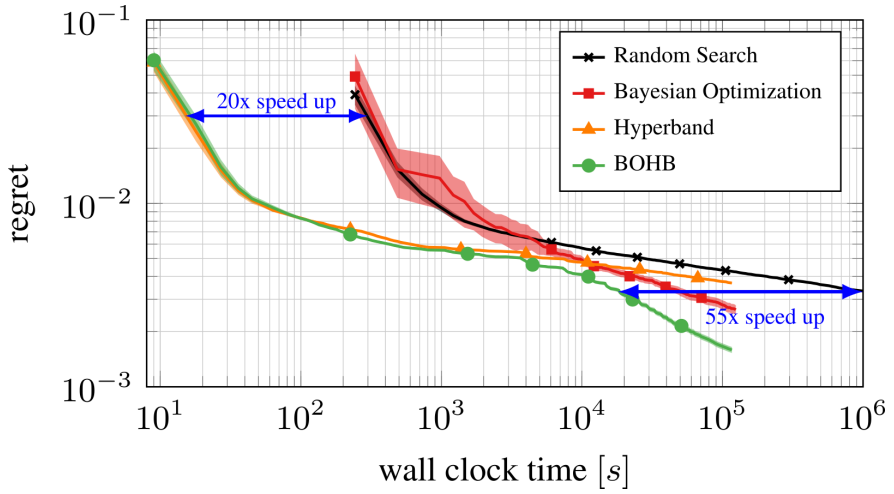


Figure 4.6: Characteristic performance of BOHB as compared with Hyperband and Bayesian optimization (Falkner et al., 2018, Figure 1). The plot shows the regret of using a certain hyperparameter optimization methods by elapsed time of the optimization run. The regret is the difference between the performance 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.

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.

4.4 HYPERPARAMETER ANALYSIS

Because one of the raised concerns about Adam is its sensitivity towards its hyperparameters, we analyzed the influence of the optimized hyperparameters on the agent's performance. For this we used - besides manual analyses - two methods: *fANOVA* and *local parameter importance.* Both build a model mapping the joint domain of the hyperparameters to a performance value using available evaluations - here given by first running BOHB.

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[14] on existing data gathered by a Bayesian optimization method (such as BOHB) 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).

Functional ANOVA (Hooker, 2007; Huang et al., 1998) decomposes

14. Random forests are a popular and computationally efficient class of machine learning models. A random forest 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.

the variance $\mathrm{Var}[f]$ of some black-box function[15] $f : \Theta_0 \times \cdots \times \Theta_k \to \mathbb{R}$ with parameter space $\Theta = \Pi_{i=0}^{k} \Theta_i$ into a set of separate components $\mathrm{Var}[f] = \sum_{s \in \mathcal{P}\{\Theta_i | 0 \leq i \leq k\}} v_s$ with one component v_s per set s of parameters.

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.

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

Local Parameter Importance (Biedenkapp et al., 2018), on the other hand, 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, in which the parameter p takes the value x and all other parameters take the same values as in θ. Then:

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

Hyperparameters: number of episodes m,
number of time steps per episode l,
size of replay memory n,
off-policy probability ϵ,
minibatch size b

Given: starting observation ω_0, pre processing function ϕ,
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**
  $\kappa_0 \leftarrow \omega_0$; (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
    $(r_t, \omega_{t+1}) \leftarrow \text{execute}(a_t)$;
    $\kappa_{t+1} = \kappa_t, a_t, \omega_{t+1}$;
    $\phi_{t+1} = \phi(\kappa_{t+1})$;
    $\mathcal{D}$.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*(ϕ_{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
    $\vec{\theta} \leftarrow \text{gradient\_descent\_step}(\vec{\theta}, \vec{m}, \vec{y})$; (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 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 η,
fraction of random configurations ρ,
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 $s \in \{s_{max}, s_{max} - 1, \ldots, 0\}$ **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 $i \in \{0, \ldots, s\}$ **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, ρ, n_{min}, n)**:**

  if *num of random configs so far* $< \rho$ **then**

    return random-configuration($\mathcal{C}$)

  else if $\{D_b : |D_b| \geq n_{min} + 2\} = \emptyset$; `(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 BOHB. 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 built 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. 105).

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 agent's implementation.

Because we were interested in how volatile the optimization proceeds, we logged the optimizers' internal moment 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 use different optimizers (in the case of AdamW) and 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 built 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 built.

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. This allows us to evaluate the agents learning capabilities in a fair way and to balance the resolution of the training statistics.

5.3 ENVIRONMENTS

Ray natively works with environments implemented in OpenAI Gym (Brockman et al., 2016). For the most part, we used the popular and simple Cartpole environment. For a guide on how to repeat our experiments on different environments, see Appendix B (p. 105).

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.

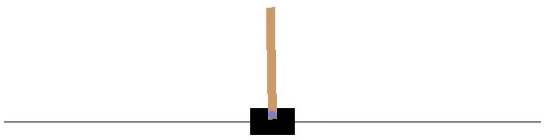


Figure 5.1: A rendering of the Cartpole environment from OpenAI Gym. The horizontal line is the track on which the cart can move. Its ends mark the positions at which the episodes are automatically terminated. The brown line is the pole which the agent must balance, the blue dot is its pivot point.

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

- the magnitude of the angle between the pole and a the line perpendicular to the cart 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 one per time step, hence the episode reward equates the episode length.[7]

7. Note how this induces a performance ceiling. No agent can receive an episodic reward greater than 200.

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. Figure 5.1 illustrates the environment.

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 optima that fail to generalize well.

5.3.2 *MountainCar*

MountainCar (Moore, 1990) is similar to Cartpole in that it is an environment that is both computationally inexpensive and does not require the agent to do a lot of abstraction. In it, the agent controls a cart that is placed in between two hills and has to drive up the cart on the hill to the right. However, the cart's engine is not strong enough, such that it needs to swing up in between both hills to build up enough momentum to reach the top. The reward the agent receives is minus one per time step that it takes to reach the goal. The episode is automatically terminated

after 200 time steps. We used it as a second inexpensive problem on which to perform hyperparameter optimization.

5.3.3 *Arcade Learning Environment*

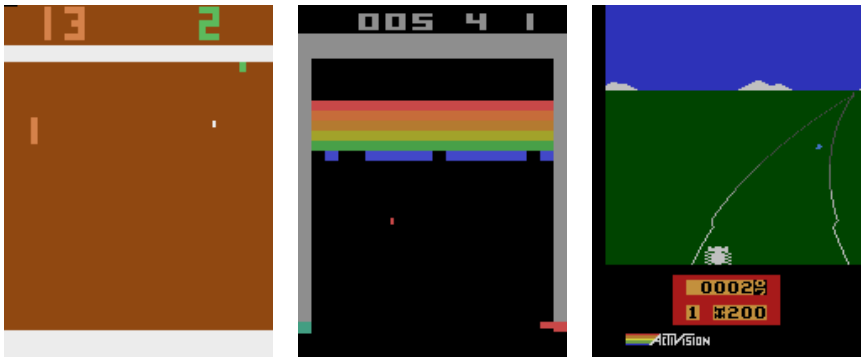
The *Arcade Learning Environment* ALE (Bellemare et al., 2013) is a set of environment that model classic Atari 2600 arcade games. It is a popular testing ground for reinforcement learning algorithms ever since Mnih et al. (2013) demonstrated their DQN algorithm on a number of Atari games. Popular examples are the games *Pong*, *Enduro* and *Breakout*. We selected this subset of games as these are games that Mnih et al. (2013) found DQN to perform well in. Hence, it is straightforward for us to evaluate the different optimizers in. In environments in which DQN systematically fails, on the other hand, it would be hard to observe the influence of different optimizers.

Stochasticity While in Cartpole stochasticity is introduced by randomly initializing the environment, this is not the case in ALE. Instead *sticky actions* Machado et al. (2018) are employed. This means that the action selected by the agent will be repeated for a random number of time steps.

Fortes Other than in Cartpole the relevant features of the environment are not directly fed to the agent, but instead images showing the current game screen (see § 4.2.1 (p. 22) for common preprocessing steps). From these, the agent has to extract the necessary features itself, posing an additional challenge through a larger state space and the need for feature extraction. Further difficulties involve the larger set of possible actions and the scarcity of the reward. Hence, the Atari Learning Environment poses greater challenges to the agent as it requires abstraction from the observation and generalization of of behavior through the larger state space.

Flaws These qualities however come at a cost. On the one hand, simulating the environment itself is computationally more expensive. On the other hand, the observations vectors are larger themselves and due to their greater complexity they require the DQN agent to posses a larger neural network to achieve a decent a performance. Hence, whereas a DQN agent can be trained in Cartpole on a single CPU for 2 million time steps in about three and a half hours, in ALE it can only be trained for about 20,000 time steps in the same time.

Additionally to every training iteration taking 100 times longer, these more challenging environments also take a longer training in order to be mastered. Machado et al. (2018, p. 9) reported that "it is fairly standard to train agents for 200 million frames", which corresponds to 50 million time steps with preprocessing. These factor rendered it infeasible for



(a) Pong (b) Breakout (c) Enduro

Figure 5.2: Screen renders of three environments from ALE. These video frames are often preprocessed and then given to the agent as its observations, as discussed in § 4.2.1 (p. 22).

us to perform hyperparameter optimization, which requires repeated evaluation of agents, in ALE.

Pong

In Pong the agent has to control a cart, moving it right and left on a track. The aim is to repeatedly deflect an incoming ball, which is bouncing off the cart upon hitting it, and not to let it pass. The agent is playing against a game-controlled opponent who is placed on the opposite side of the playing field, can execute the same actions and follows the same aim. The agent and the opponent score by playing the ball past each other. The aim for the agent is to maximize its game score over that of the opponent.

Renders off all three environments are depicted in Figure 5.2.

Breakout

Breakout works similar to Pong. However, here the agent does not play against an opponent but tries to deflect the ball in such a way as to hit blocks that are placed on the other side of the playing field and that are destroyed upon collision and deflect the ball back. The agent scores by destroying a block and loses a live by letting the ball pass. The agent has a limited number of lives. When all of them are used up, the game is over. The agent's objective is to maximize the game score.

Enduro

In Enduro the agent has to control a racing car, accelerating it and moving left and right on the track to avoid other cars. The agent scores by overtaking other cars and is bumped back by crashing into another car. What sets it apart from Breakout and Pong is that it possesses changing background textures and a day-night cycle. Hence, it demand higher standards of abstraction due to the changing visuals.

5.4 HYPERPARAMETER OPTIMIZATION

We used the BOHB implementation provided by HpBandSter.[8] It is built with distributed computation in mind, with one central server building the model and scheduling evaluations on a number of workers.

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. This is due to the fact that the model used by BOHB to sample configurations is built using past evaluations on the same budget. Usually an instance of the optimizer will run multiple iterations of BOHB that each use the models from the previous iterations.

8. See github.com/automl/HpBandSter

Fortes and Flaws Although HpBandSter is built for parallel computation, it installs its own abstraction layer for it with a central server and an number of workers. As we already use Ray to parallelize computations, this introduces some complications in the design of the code base. Ray introduced its own hyperparameter optimization framework Tune (Liaw et al., 2018). For future work, it might be worthwhile to implement BOHB in Tune, as to remove a further layer of abstraction and the computational overhead that comes with it.

5.5 LOGGING AND VISUALIZATION

Tracking Moment Tensors To gain further inside into how volatile the training process proceeds, we propose to track the optimizers internal moment variables $\vec{m}$ and $\vec{v}$. As these are large tensors, entirely tracking them over the whole trajectory would introduce large memory requirements. Instead there are three more efficient ways. Keep in mind, that we were primarily interested in how strongly these values change.

1. Calculating the mean of each tensor and tracking those instead. Note though, that this underestimates the actual volatility. [9]
2. Buffering the tensors for one time step respectively and in each time step calculating the Euclidean distance of each tensor to the last time step's tensor gives an unbiased impression of the volatility of the tensors across time steps.
3. To get an impression of the spread of the moment variables, we propose to aggregate 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 be to high. From the histogram, distribution plots can also be automatically generated.

9. 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.

As noted, the used agents create Tensorboard logs of the training process, plotting quantities such as the episode reward per training iteration. We built a custom Tensorflow logger to track all these values in addition to the values that are tracked in RIlib by default.[10]

10. Note, that logging these values adds a large computational cost, such that we could not do so during all experiments.

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. Because 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. [11]

11. 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 5.3 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. Hence, 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 and as a consequence having a lower resolution in areas of interest.

5.6 HYPERPARAMETER ANALYSIS

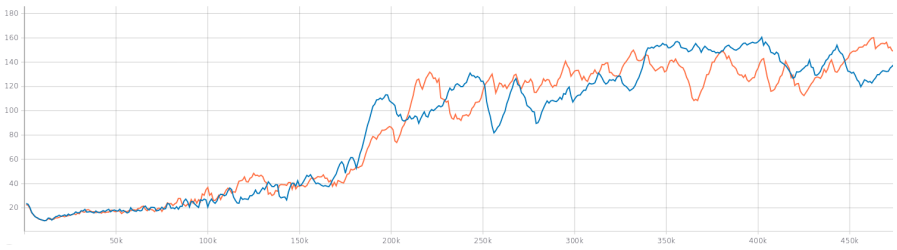
CAVE In order to evaluate the results from HpBandster/BOHB, we relied on CAVE (Biedenkapp et al., 2018), a tool to automatically analyze the data produced by the run of a hyperparameter optimization method, such as BOHB, reporting among other things, a performance analysis, hyperparameter importance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

CAVE works natively with the artifacts created by running an instance of the BOHB implementation in HpBandSter.

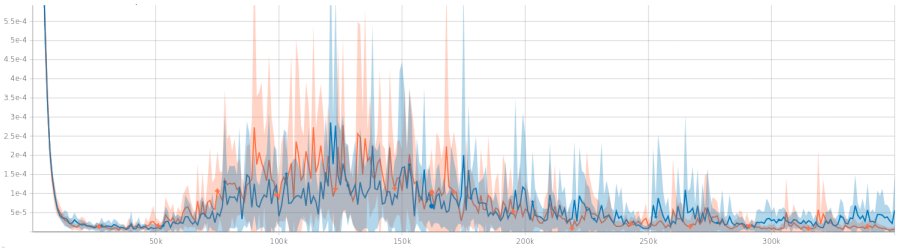
We were particularly interested in the hyperparameter importance, which CAVE evaluates using *fANOVA* and *local parameter importance*. Furthermore it reports the correlation between the performance of configurations at different budgets. This allows to judge how sensible an optimization run actually was.

5.7 COMPUTATIONAL RESOURCES

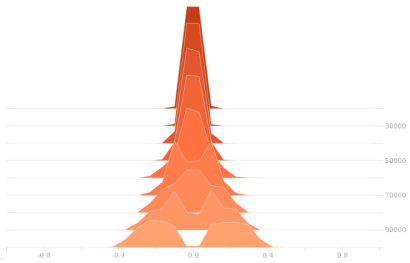
As our experiments required large computational capacities, they were carried out on two computation clusters: META and BwForCluster



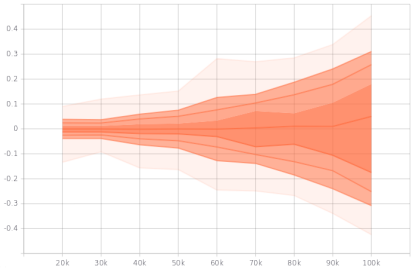
(a) A vanilla Tensorboard graph, showing the mean reward of two DQN agents over the training time steps - allowing to get a first impression of the training progress.



(b) A custom Tensorboard graph with confidence intervals, showing the mean Euclidean distance of the moment-tensor $\vec{m}$ of the biases of one layer of the Q-network to the last time step, over the training time steps for two groups of agents. This plot illustrates in which part of the training the momentum for the Q-network is changing the most, giving a deeper understanding of the progress.



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



(d) The distribution plot for c. The lines represent (from top to bottom) the 100%, 93%, 84%, 69%, 50%31%, 16%, 7% and 0% percentiles - i.e. the extreme values, median, $1.5\sigma, \sigma$ and 0.5σ percentiles, assuming Gaussianity.

Figure 5.3: Examples for the four generated plot types. Whereas 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, giving more insight in how their components develop.

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.

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

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

PART II

EXPERIMENTS

6 PERFORMANCE ESTIMATION

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

— Karl Kristian Steincke, *(probably)*

6.1 PROBLEM

Efficiently Estimating Performance As introduced in § 4.3.2 (p. 27), Hyperband, on which BOHB 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. $\hat{f}_b$ then becomes a better approximation for f with increasing budget 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 performance of the agent for a certain hyperparameter setting. This further involves finding a suited metric for BOHB to optimize.

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 of an reinforcement learning agent with identical hyperparameter settings produced statistically significant results across the whole training interval.

Noise-Cost-Trade Off The results of Henderson et al. (2018a) illustrate that to get a precise estimate of the performance of a given algorithm and configuration, it is necessary to not only train the agent for a long time, but also to train a large number of agents. This however implies a large computational cost. As the automatic hyperparameter optimization of an agent requires to evaluate a number of configurations, such costly evaluation for a single configuration is not feasible. Hence, we have to engage in a trade-off between having a cheap estimation and having an estimation with a good signal-noise ratio.

Duration-Repetition Trade-Off Additionally, the computational budget needs to be effectively split between training the agent for a longer time and training multiple agents for the same configuration. As training multiple agents can be effectively parallelized, training multiples agents

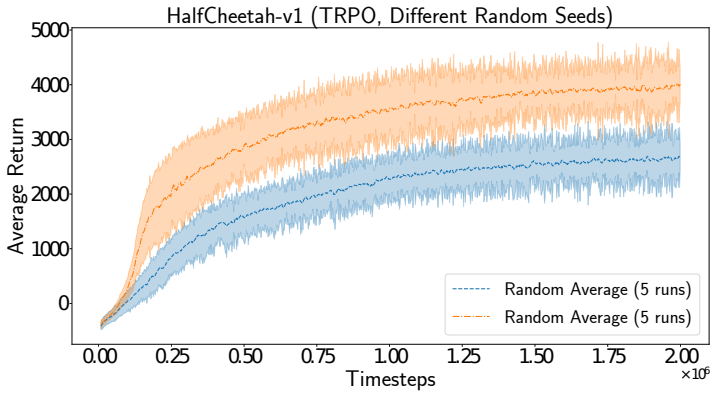


Figure 6.1: Performance difference between two groups of aggregated trials (Henderson et al., 2018a, figure 5). The authors trained a TRPO agent in the Half Cheetah (OpenAI Gym) environment 10 times using the same hyperparameter settings and then splitted the runs in two groups with five runs each. The 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). The authors determined significance via the average of the 2-sample t-test ($t = -9.0916$, $p = 0.0016$) across all the training iterations.

scales almost linearly with the number of available processors. Longer training times, on the other hand, can not be sped up in such a manner. Hence, there is an incentive to have stable estimates for shorter training times at the cost of more concurrent evaluations. Especially, as Bayesian hyperparameter optimization methods work by sequentially evaluating configurations to build a model over time.

Budget Interpretation and Evaluation Metric In a Hyperband-like setting the budget could be interpreted as both the number of trained agents and the training duration. However, it is unclear which of the two provides the more stable estimates. This - and which metric is most suited to rate an agent's performance - are the key questions that we try to answer with this experiment.

6.2 DESIGN CHOICES

6.2.1 *Evaluation Metrics*

A 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, we expected that this might not be optimal, as one of the goals is to find 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. Estimating the agent's final performance can be achieved by rolling out the agent for a fixed number of time steps and reporting the average reward it achieved. Another possibility is to

report the *last n reward*, i.e. the average reward over some last number n training iterations.

Falkner et al. (2018) reported the *number of epochs until convergence* of the performance of the agent, effectively capturing the learning speed. This however poses a number of problems. First of all, it only works for environments with a performance ceiling, like Cartpole. Though we also train in this environment, we only use it as a cheap testing ground. Overall we are interested in more general environments. Second, judging convergence is somewhat arbitrary as it usually means considering a run to have converged when the return was constant for an arbitrarily chosen number of episodes. Just slightly missing the chosen number of stable episodes can make a drastic difference in this metric. Hence, a metric capturing the *anytime* performance might be more appropriate.

As can be seen, there are a lot of possible metrics, the suitability of which depends on the setting at hand. To gain an insight into which is most suited, we collected both *training* and *rollout metrics* in our experiments.

Training Metrics A common metric to judge the training progress is the *mean episodic reward*, which is the mean reward over one training iteration.[1] Other training metrics that we tracked per training iteration are the maximum and minimum episodic reward and the *last n reward*, i.e. the mean reward over some last n number of episodes. This metric estimates the current final performance of the agent and good correlation of this metric with actual final performance would allow us to omit rolling out the agents to judge their final performance, which would save computational cost.

1. In all our experiments, one training episodes lasted for 1000 time steps.

To summarize the whole training interval, all these metrics can be be trivially generalized over the set of all training iterations. For a better distinction we will refer to the mean reward over the whole training duration as the *anytime performance*.

Rollout Metrics When rolling out the agent, it is also possible to track different metrics. Due to the reward structure in Cartpole, we had to resort to the naive mean. However, in other cases, different metrics are possible and might be more adequate.

If the episode length is not fixed, due to environments automatically terminating the episode when certain criteria are met (see § 5.3 (p. 36)), 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. An alternative is 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 and

converging episodes are awarded more . Recall however, 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. For other environments, though, it might be worth to experiment with this alternative.

6.2.2 *Cost-Critical Factors*

Parallelization Running a single Rllib DQN agent in Cartpole only makes effective use of a single processor core. Giving the agent additional cores does not speed up the computation. This in part due to the chosen environment. In environments where the sampling of observations is more costly, the agent can benefit from using additional Ray workers. This means that the total time it takes to train a single agent to a point where its performance is a decent estimator for the usual performance of this hyperparameter setting is the crucial variable in our experiments.

Training Duration and Metric Selection The number of necessary training samples is highly dependent on the environment at hand. Furthermore, we wanted to get a first impression on which of the metrics listed above give a decent performance estimate. We performed a few simple experiments in Cartpole, training agents with default parameters to get sense of the number of necessary training time steps.

The usually used default settings for Adam are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$. For the regularization strength of AdamW on the other hand, we know of no commonly agreed upon default, yet. We settled on $\lambda = 0.00025$ as this value performed well across the benchmarks run by Loshchilov and Hutter (2017). Figure 6.4 (p. 54) shows the development of training metrics of agents using Adam and AdamW over the course of 2 million time steps. We collected mean, maximum, minimum and last n reward. As the agents were trained for 2000 iterations with 1000 time steps each, we chose $n = 20$, never capturing more then the last four training iterations.

The training of a single agent took about three and a half hours total on a single CPU. However, we found the agents' performance to be stable after around 200,000 time steps. The result further suggests, that both the mean episode reward and the last n reward are good metrics for the performance, not so the minimum and maximum reward. The maximum reward simply achieves the peak performance too fast to be expressive about the training progress. The minimum reward did not even improve with training but in fact got worse for Adam. On the other hand, the mean reward summarizes the performance over the whole training iteration and the last n reward gives an estimate of the current final performance.

Note, that these findings are highly dependent on the environment at hand. We trained in Cartpole here as this is the environment in that we will also perform the hyperparameter optimization. The motivation

to chose this environment is that its computational efficiency that makes the hyperparameter optimization feasible in the first place.[2]

Note further, that at this point we were not interested in the differences between Adam and AdamW. We solely deployed both to make sure that our set up works with both of them.

2. Remember, that a single agent can be trained for two million time steps in about three and a half hours on a single processor core, while the same takes about 100 times longer for the games of the Atari Learning Environment.

Number of Trials Reminiscent to the results of Henderson et al. (2018b) we performed a few basic experiments to determine how many repeated evaluations of one configuration were necessary to reliably estimate its performance. Remember, that a single evaluation underlies severe volatility. We found that evaluating 10 agents gave remarkably stable results in Cartpole. In more complex and dynamic environments however, this might differ. Figure 6.2 illustrates the robustness of the average. Although the confidence intervals for both groups are quite large, the development of the means of both groups is surprisingly congruent and the results of a t-test are not statistically significant ($p = 0.470$). We repeated the experiment twice more and found these results to be stable. We opted to plot only two repetitions for illustrative purposes.

This illustrates that the variance in performance is highly dependent on both the agent and the environment. To see whether we could reproduce the results of Henderson et al. (2018a) with DQN and Cartpole, we performed another ten trials of a randomly sample configuration of Adam and then cherry-picked the best and worst runs into two groups. We evaluated the performance over the first 200,000 time steps, the interval in which the agent made the biggest performance improvements. As Figure 6.3b illustrates, the neighborhood of the area of largest performance gain usually is a reliable area of high variance as well. However, as Figure 6.3a illustrates, even with the cherry-picking, we did not get significantly different results between the two groups here. The variance level also depends on the sampled configuration - for the configuration depicted in Figure 6.3 the standard deviation never surpasses 17, while reaching values twice as high for the configuration shown in Figure 6.2, despite the larger group sizes. However, there is qualitative difference between our results and those of Henderson et al. (2018b). We repeated the experiment with various configurations, getting different levels of variance. The means of both groups would vary and move around each other over the training duration. But for no configuration did we find a systematic difference in the performance in the way found by Henderson et al. (2018a), where the performance curves for two groups did diverge from each other. We think that this might occur in richer environments like Half Cheetah because of path dependencies in the learning process. An agent may make a valuable experience by chance that drastically impacts the further learning process. In Cartpole however, the experience space is so simple that this simply does not occur.

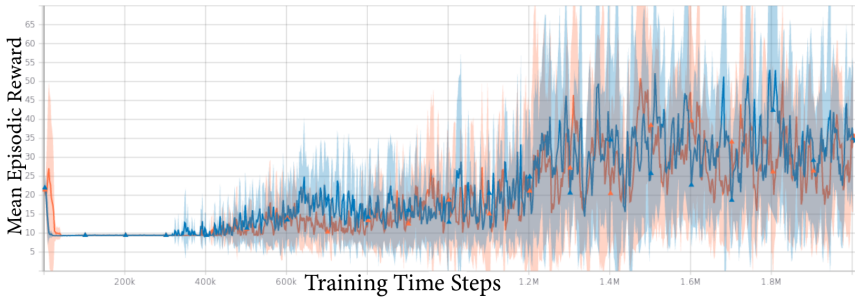
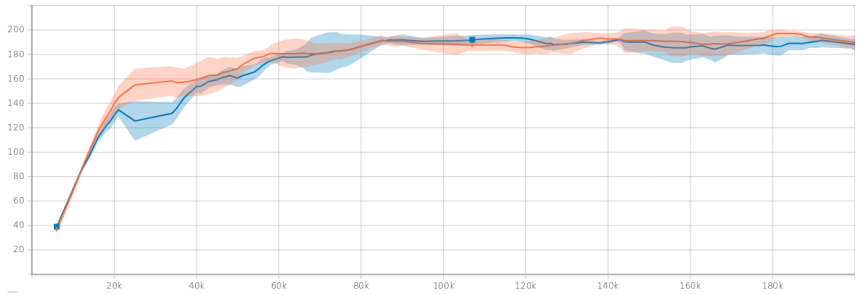
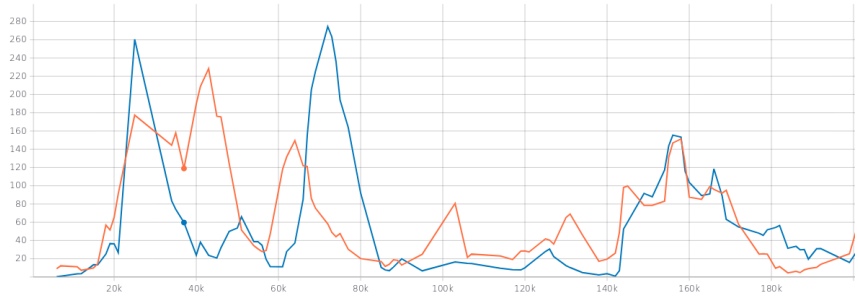


Figure 6.2: The mean reward and standard deviation of two groups of 10 trials each with the identical random configuration. The DQN agents were trained in Cartpole for 2 million time steps. Although the standard deviation is quite high, the developments of the group means are surprisingly congruent. Repeating the experiment confirmed this effect - we just plotted two groups for lucidity. Following Henderson et al. (2018a) we calculated the average 2-sample t-test across the training time steps and found no significant deviation between the two groups ($p = 0.470$, $t = -0.322$).



(a) The average mean episodic reward of both groups with confidence intervals.



(b) The variance of the mean episodic reward of both groups.

Figure 6.3: The average episodic reward for two cherry-picked groups of five runs, each of the same randomly sample configuration of Adam in Cartpole. Other than in the experiments of Henderson et al. (2018a) the performance values for both groups did not diverge from each other.

6.3 EXPERIMENTAL SETUP

Interpretation of the budget As we have seen, there are two factors that both lead to a more stable estimate of the performance but also an increasing cost: the number of trained agents and the training duration. The BOHB 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.

	Parameter	Value
Ray	Dueling DQN	False
	Double DQN	False
	Prioritized replay	False
	Noisy DQN	False
	Time steps per training iteration	1000
	$n_{\textit{training iterations}}$ for $n_{\textit{agents}} \propto$ budget	2000
	Training batch size	32
	Sample batch size	4
Adam	Θ_α	$[10^{-6}, 10^{-1}]$
	Θ_{β_1}	$[10^{-1}, 1]$
	Θ_{β_2}	$[10^{-1}, 1]$
	ϵ	10^{-8}
AdamW	Θ_λ	$[10^{-5}, 10^{-1}]$
Runner	Budget steps	1, 3, 9, 18
	Rollout time steps	10,000
	$n_{\textit{agents}}$ for $n_{\textit{training iterations}} \propto$ budget	10
	Sampled configurations	10
	Runs per sampled configuration	2

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. The values from each Θ_i were sampled using a logarithmic scale.

We thus designed an experiment to investigate which of the two interpretations gives a more stable estimate, i.e. for which of the two 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. A full list of hyperparameters can be found in the accompanying repository.

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

We then randomly sampled ten different configurations from the configuration space spanned by $\Theta_\alpha \times \Theta_{\beta_1} \times \Theta_{\beta_2}$, respectively $\Theta_\alpha \times \Theta_{\beta_1} \times \Theta_{\beta_2} \times \Theta_\lambda$, with the domains listed in Table 6.1. 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 constructed a set of budget steps, analogously to following successive halving with $\eta = 3$, a minimum budget of 1 and a maximum budget of 9, plus an additional budget step of 18: $\{1, 3, 9, 18\}$. η was chosen to be 3 as this is a commonly used value. The minimum and maximum budget were motivated by the presented experiences; training agents for 200,000 time steps seemed to already give a decent estimate, while 2 million time steps seemed definitely enough to judge the final performance even for worse performing configurations. Similarly we found that averaging about 10 agents gives very stable estimates. More precisely, for a budget of b we trained 10 agents for $b \cdot 200{,}000$ time steps each and b agents for 2 million time steps each. Hence, both interpretations of the budget work well with the proposed budget steps. The extra step of 18 was added to see whether further increasing the budget does still result in meaningful improvements.

For each sampled configuration, for each budget value, for each interpretation of the budget, we executed two trials for some extra redundancy and dispersion measurement. We also rolled out each agent for 10,000 time steps to judge its final performance. We then collected the anytime performance, maximal episodic reward, last n reward and mean rollout reward.

Repetitions We repeated the experiment three times for Adam to get a sense of how volatile the results were and did another repetition with AdamW to see whether the other optimizer causes a qualitative change of the results. Although the different sampled configurations between the repetitions affected the correlation values (±20%), the observed trends were the same across all repetitions. Only for one repetition of Adam, all sampled configurations reached the performance ceiling, causing a zero variance for some of the reported metrics. In this case, the reported values are the average of the other repetitions. Furthermore, we did not find a qualitative differences between Adam and AdamW and report the average of all repetitions of the experiment.

6.4 RESULTS

Table 6.2a summarizes the Pearson correlation between the performance metrics on different budgets for both interpretations. The considered metrics are the maximum, mean (anytime) and last n training reward and the mean rollout reward. As the Pearson correlation only measures linear relations, we also calculated the respective Spearman rank correlation coefficient, which can be found in Table 6.2b. The Spearman correlation coefficient should also be more robust against outliers and is used by hyperparameter optimization methods, such as BOHB. Both correlations, however, show the same patterns.

interpretation	reward	budgets					
		1-3	1-9	1-18	3-9	3-18	9-18
agents ∝ budget	mean	0.802	0.745	0.778	0.835	0.921	0.886
	max	0.748	0.708	0.602	0.958	0.968	0.962
	last n	0.878	0.674	0.716	0.660	0.736	0.695
	rollout	0.852	0.847	0.844	0.426	0.802	0.367
time steps ∝ budget	mean	0.657	0.325	-0.081	0.776	0.241	0.672
	max	0.673	0.144	0.091	0.370	0.278	0.962
	last n	0.651	0.263	-0.107	0.710	0.181	0.619
	rollout	0.676	0.267	-0.139	0.612	0.093	0.531

(a) Pearson correlation coefficient.

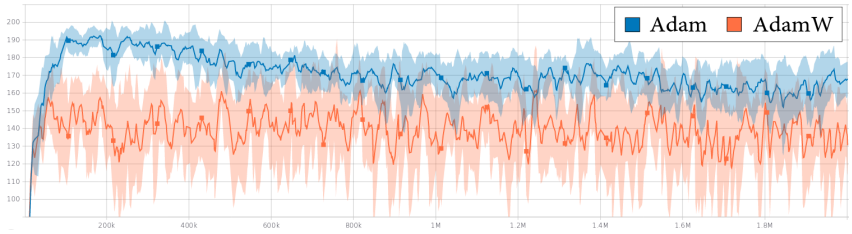
interpretation	reward	budgets					
		1-3	1-9	1-18	3-9	3-18	9-18
agents ∝ budget	mean	0.682	0.696	0.684	0.818	0.903	0.801
	max	0.879	0.759	0.687	0.864	0.851	0.988
	last n	0.717	0.603	0.721	0.695	0.749	0.699
	rollout	0.786	0.784	0.760	0.489	0.704	0.510
time steps ∝ budget	mean	0.555	0.263	-0.043	0.732	0.358	0.720
	max	0.714	0.414	0.310	0.613	0.495	0.932
	last n	0.499	0.237	-0.116	0.580	0.277	0.560
	rollout	0.676	0.283	-0.163	0.505	0.132	0.544

(b) Spearman rank correlation coefficient.

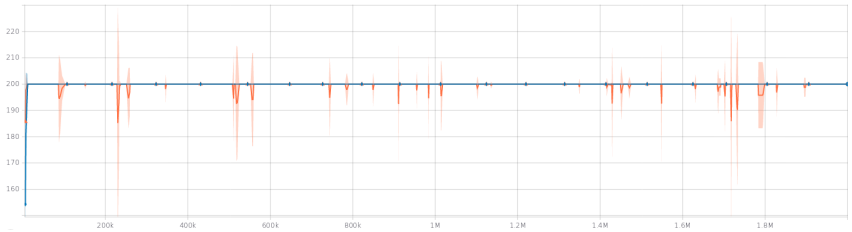
Table 6.2: Correlation coefficient between the performance of the configurations on the different budgets. The correlations were computed over the vectors of the performances of the 20 trials (two trials per 10 configurations) for each repetition of the experiment and then averaged over the repetitions. All repetitions showed the same basic patterns. However, note that for one repetition all configurations have reached a maximum reward of 200 after two million time steps. Hence, for the agents-interpretation of the budget the maximum, last n and rollout performance had zero variance and thus no correlation could be computed. The reported values were then only averaged over the remaining repetitions of the experiment.

For the interpretation of the budget as a number of agents, there is a strong positive correlation (≥ 0.682) for the mean training reward across all budgets. Similarly for the maximum, last n and mean rollout reward.

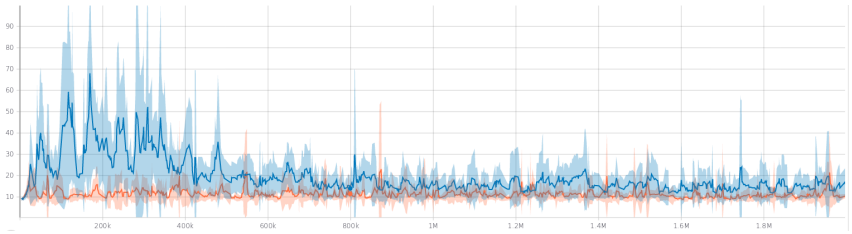
On the contrary, the interpretation of the budget as a number of time steps does not result in a strong positive correlation of the performance across budgets steps for the mean training and rollout reward. The mean reward shows a strong positive correlation (≥ 0.555) for budgets 1-3 and 9-18 and only a weak one (≥ 0.263) for budgets 1-9 and 3-18. Particularly, there even is a near zero negative correlation between the performance at budgets 1 and 18.



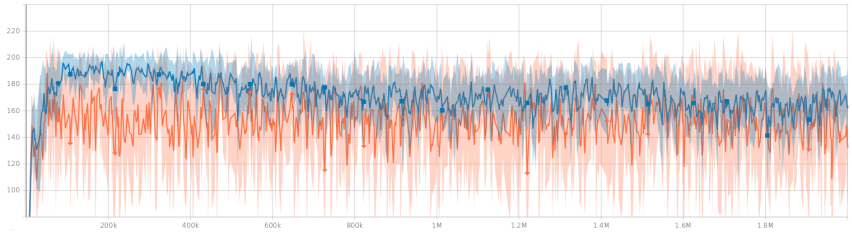
(a) The mean episode reward.



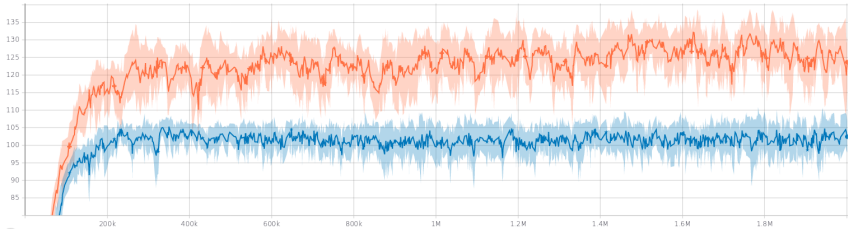
(b) The maximum episode reward.



(c) The minimum episode reward.



(d) The *last n* reward with $n = 20$.



(e) The mean predicted Q-value.

Figure 6.4: Training metrics for a DQN agent in Cartpole for Adam and AdamW per training time steps. Default values were used for the hyperparameters - for both: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, for AdamW: $\lambda = 0.00025$. Remember that in Cartpole episodes are automatically terminated after 200 time steps and that thus 200 is the maximum reward possible. As Subfigure e illustrates, the agents prediction have reached their final state around 200,000 time steps. However, as Subfigures a and d illustrate, the agents have already achieved their final performance around 100,000 time steps. Subfigures c and b suggest that both the min and max reward are poor performance measure in this case, as the max reward does not change significantly over the course of the training and the min reward in fact gets worse. Subfigure e may hint at why Adam outperforms AdamW here: its predicted reward is way lower, which usually leads to a better exploration of the state space.

6.5 DISCUSSION

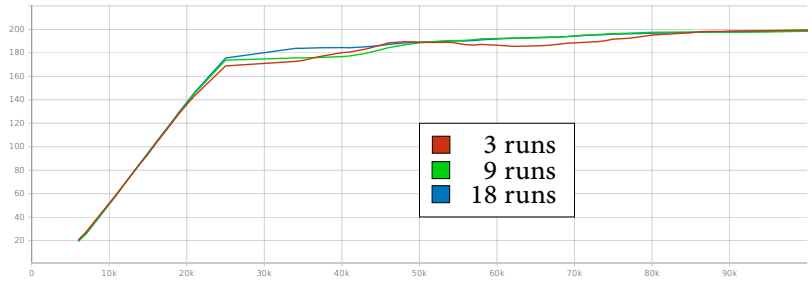
The results suggest that interpreting the budget as a number of agents gives a stable and strong positive correlation for the mean (anytime) training reward across budgets. Hence, this interpretation provides a stable estimate of the performance of a configuration and the mean reward is a well suited training metric for this interpretation.

Additionally, the mean rollout reward and maximum training reward are not informative for well performing configurations as they show almost no variance for this interpretation of the budget. This is due to the fact, that all well performing configurations reach the performance ceiling of Cartpole. Remember, that for a whole repetition of the experiment these values showed zero variance between different configurations. As hyperparameter optimization methods are expected to find well performing configurations, this problem would only become more grave.

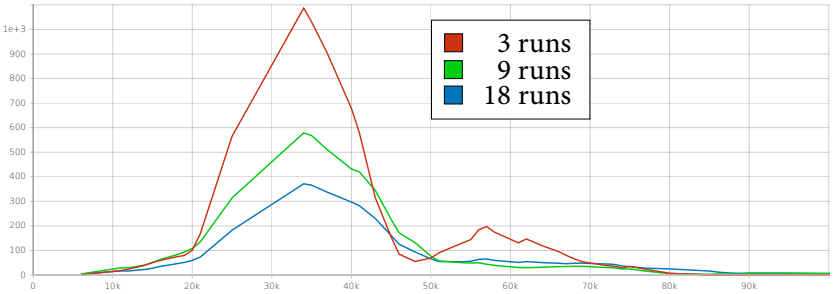
Interpreting the budget as a number of time steps on the other hand does only give a poor correlation of the mean training reward, hence rendering it unsuited for a stable performance estimation. Particularly interesting is the near zero negative correlation between budgets 1 and 18.

More repetitions reduce variance Interpreting the budget as the number of agents leads to good correlations because the performance of a single run is an unbiased estimator for the average performance of multiple runs of the same configuration. Increasing the budget, hence the number of agents, only reduces the variance of the performance. Figure 6.5 illustrates this fact. Variance in performance peaks shortly after the largest average performance gains, as some members of the group are a bit slower to learn. However, as expected there is no systematic difference in the mean performance of all groups. Instead, the growing number of repetitions reduces the variance of the group. This behavior is again illustrated in Figure 6.7. The performance values for all five configurations are stable across all budgets for the agents-interpretation of the budget.

In Cartpole many configurations succeed after some time As we will discuss in the following paragraphs, Figure 6.6 illustrates the cause for the bad correlations of the configurations' performances for the time-interpretation of the budget and the unsuitability of the maximum episodic reward and mean rollout reward. In Cartpole most of the sampled configurations reached the best possible performance after some time. What varies is the number of training time steps it takes them to get there. Hence, the quality of a configuration cannot be judged by the the final performance but only by the performance across the whole training interval. Hence, the maximum episode reward in training



(a) The mean episodic reward, averaged for different groups of different size.



(b) The variance of the mean episodic reward for groups of different sizes.

Figure 6.5: Mean episodic reward (averaged over different group sizes) and its variance for a randomly sampled configuration of Adam in Cartpole.

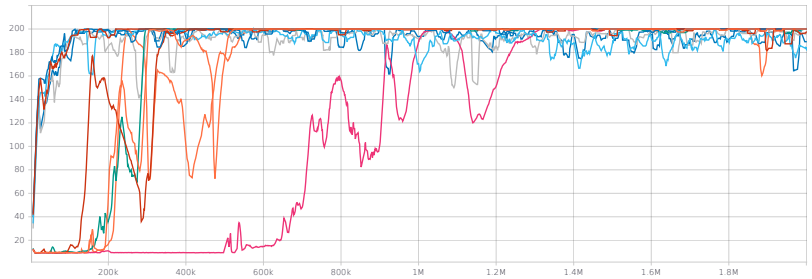


Figure 6.6: Mean episode reward over the training time steps for ten randomly sample configurations of Adam in Cartpole. Note that after a long enough time, all configurations reach the optimal performance. Only the number of training iterations it takes to reach that point vary.

and the rollout reward are uninformative for judging the quality of a configuration.

Different training lengths change the composition of variance Besides random noise, there are two sources of variance for the anytime performance between configurations. The first one is the initial learning speed and the second one is the stability once the performance ceiling is reached.

All of the sampled configuration reach the optimal mean reward after between 100,000 and one million time steps. For the interpretation of the budget as a number of agents, the time it takes the agent to reach this performance is reflected in the anytime reward. As the training duration is fixed, so is the composition of variance out of learning speed and stability. The only variable between the budgets is the number of

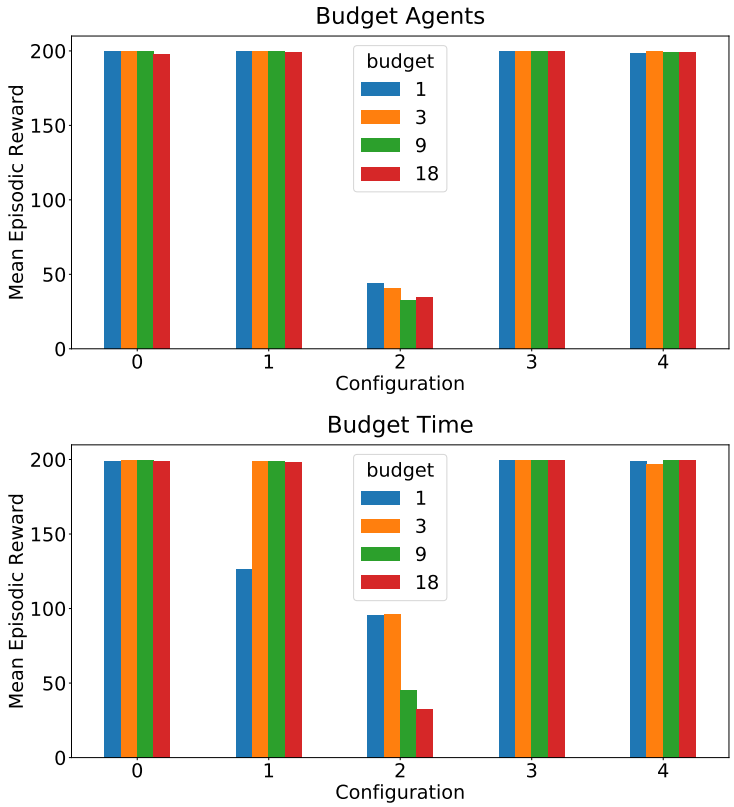


Figure 6.7: Performance development of five random configurations of Adam in Cartpole for both interpretations of the budget. Notice how the performance is fairly constant across all budgets for the agents-interpretation. For the time-interpretation however, this is not the case. The values are also stable for configurations 0, 3 and 4 who are fast learning and stable configurations. For configuration 1 however, that is slower to learn, and configuration 2, that is very unstable, the performance develops in opposite directions across budgets.

agents being trained, leading to a decent correlation across the budgets.

On the other hand, training an agent for longer than one million time steps only shrinks the variance in average performance, as no configuration can improve its performance any more. Even more, some of the configurations that initially learn faster and thus quickly reach a good performance, tend to repeated, small deviations from the optimal policy when being trained further. This effect can be seen in the blue graphs in Figure 6.6: they reach the peak performance fast but in the long run are less stable than the slower to improve red and orange graphs. At the same time though, there are also configurations that only learn slowly but are therefore very stable.

These observations can explain the bad correlation of the performance values for the time-interpretation of the budget. Increasing the budget shifts the composition of the variance as well: more variance is contributed by instabilities and less from learning speed. Some fast learning configurations tend to later instabilities. Hence, a part of these configurations that have an above-average performance initially perform

below-average for longer training durations (hence budgets).

Figure 6.7 once again illustrates this point. Configuration 1 is slow to learn and improves over time/budgets. Configuration 2 however is unstable and gets worse over time/budgets.

Leaning Speed-Stability Trade-Off Hence, there is a trade-off for an optimizer between picking configurations that learn fast and ones that learn slower but are more stable over longer training periods. The total length of the training period then determines in favor of which option the trade-off is decided. If the agents are trained for a longer time, a larger part of the variance between different configurations will be contributed by these small deviations from the optimum. For a shorter training period, on the other hand, a larger part of the variance will stem from the initial learning speed.

Variance Composition This even explains the specific correlation values for the time-interpretation of the budget found in Table 6.2. For the time-interpretation of the budget, there is a strong positive rank correlation (0.732) between the performance for budgets 3 and 9, as well as 9 and 18 (0.720), but only a weak correlation between 3 and 18 (0.358). This is due to the fact that after 600,000 time steps (corresponding to a budget of three) most configurations have already reached their best performance. Hence, there is almost no more variance coming from learning speed, but a slowly growing part from instability. Hence, while for neighboring budgets the composition of the variance is relatively similar, it diverges for budgets that are further apart. The rank correlation between budgets 1 and 3 is also noticeable (0.555), as in this period most configurations make their biggest performance improvements and there is not much instability yet. However, the fact that almost all configurations jump up to the performance ceiling in this interval, leads to a worse signal to noise ratio as there is little differentiation between the configurations. For more fine grained steps (e.g. 100,000 – 200,000 – 300,000) we would expect an even strong correlation here. The correlation between budgets 1 and 18 however is near zero (−0.043) because some of the configurations that learn the fastest were the same that were the most unstable and tended to deviate again from the optimum when being trained for a longer time, while even very slow learning configurations reached the performance ceiling within that time.

Roundup For different training durations, the composition of the variance of performance between different budgets is different as well. For short training durations, most variance stems from different learning speeds of the configurations. For longer training durations, a larger share is contributed by configurations that learn fast but are unstable. Hence, the time-interpretation of the budget leads to a different composition of

the variance between the budgets, whereas this is not the case for the other interpretation. This causes the great discrepancies in correlation.

Suitability of Metrics This moreover implies that for a long enough training duration the average rollout reward is biased towards more stable configurations. This is due to the variance in performance for more unstable configurations and the fact that it does not factor in learning speed.

Just as the last n reward, it further is not suited to adequately rank configurations as a wide range of configuration can reach the performance ceiling in Cartpole (remember Figure 6.6).

We conclude further that the anytime reward across the whole training period thus is the most suited performance metric. It respects both sources of variance and the trade-off can be directly adjusted via the training duration. The last n reward on the other hand would only give a more stable estimation of the average final performance, than actually rolling out would give. This is due to the fact that for the more unstable configurations, the exact point at which it is rolled out makes an impact on the reward, whereas this is smoothed out when averaging the reward over multiple training episodes. However, it would not incorporate the learning speed, which we found to be more crucial here.

Suitability of Budget Interpretations Interpreting the budget as a number of agents gives a good correlation between budgets because it is independent of the different compositions in variance in performance between budgets that can be found for the interpretation of budget as training time. Hence, it is generally well suited for hyperparameter optimization.

The practicality of interpreting the budget as a number of time steps, on the other hand, is strongly depending on the exact training duration. In keeping the budget steps from this experiment, reducing the number of time steps per budget from 200,000 to 100,000 should make a big improvement. Both the default configuration and the best randomly sampled ones reach the best possible performance around 100,000 time steps, the worst sampled configuration at around one million. This window of training durations thus allows for enough variance to adequately rank different configurations while at the same time increasing the resolution and improving the correlation between budgets by bringing the composition of their variance closer together.

6.6 CONCLUSION

Other than in more complex environments there is no strong path-dependency in the learning process in Cartpole. Thus, the performance of a single configuration does vary for different random seeds, but does not show systematic differences between (cherry-picked) groups of trials.

Still, it is required to train multiple agents for the same configuration to get a stable estimation due to stochasticity in the environment and the learning process. Whether interpreting the budget for the evaluation of a single configuration as the training duration does yield a good correlation of the performance between budgets depends on the exact training duration. Therefore, interpreting the budget as the number of agents to train and average the performance over is generally the better choice. The anytime performance is a suitable metric to judge an agents performance as it captures both the learning speed and the stability of the performance across the training duration.

HYPERPARAMETER OPTIMIZATION

7

7.1 PROBLEM

The problems of numerical optimization in deep reinforcement learning are manifold. Recall the problems of the optimizers' **sensitivity towards hyperparameters** with small differences deciding between convergence and divergence, as well as the **opaque effects** of hyperparameters on the behavior of the optimizer, making it difficult to compare results.

In this chapter, we investigated whether hyperparameter optimization, equipped with the insight of the last chapter, can help with these issues. Specifically, we applied BOHB to automatically optimize Adam and AdamW. We then analyzed the results from the optimization runs to see how reliably well performing configurations were found and evaluate the estimated parameter importance values. Furthermore, we manually evaluated the learning curves of the sampled configurations to judge the optimization process itself.

7.2 EXPERIMENTAL SETUP

Using our results from the previous experiments on estimating an agent's performance , we use BOHB to automatically optimize the hyperparameters of DQN using Adam and AdamW in Cartpole and MountainCar. The following are the core lessons we have learned.

Environments We chose these environment due to their low computational requirements. As discussed, one configurations can be evaluated on a single processor core for 2 million time steps in about three and a half hour. This means that a BOHB run is feasible in a matter of hours or days. More complex environments like games from the Arcade Learning Environment would take around three orders of magnitude longer.

Budget Interpretation We used both interpretations of the budget again. Recall that we have found the interpretation of the budget as a number of agents to train to give a better correlation of the budgets. However, the interpretation of the budget as training duration is more straight forward to parallelize and towards the end of our discussion

we speculated that fewer training iterations might in fact improve the correlation.

Optimization Target We selected the anytime performance, i.e. the average reward over the whole training duration as the metric for BOHB to optimize, as we found it most suited for Cartpole. This is due to the environment's performance ceiling that can be reached by wide array of configurations. Hence, what matters are the learning speed and the stability which are both factored in the anytime performance.

7.2.1 Architecture

The used implementation of BOHB 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 BOHB worker on each and a BOHB server on one of them. On each node an instance of Ray is running, all of which are connected to a head instance. Due to this, each instance of Ray may not only make use of all cores of the node that it is run on, but also *spill over* to other nodes, if necessary and possible.[1] Each node might posses multiple cores, in our case ten.

Each BOHB worker will be be told how many agents to train and for how long, depending on the budget and its interpretation. It then train up to as many agents as its node posses cores in parallel. As we train up to ten agents per configuration, we used nodes with ten cores each, such that each worker can train all its workers locally in parallel. We decided to handle it that way because we found it to be the most robust. The worker then reports back the loss (which is the negative anytime performance) and additional information to the BOHB server. The server uses the reported loss to build a Bayesian model to sample further configurations, as described in § 4.3.2 (p. 28). This architecture is illustrated in Figure 7.1.

1. This could happen in two ways: an agent might use multiple workers, some of which run on another node, or a whole agent could be trained on an other node. In case of our experiments, spilling over was not necessary as each agent only made use of one worker and no BOHB worker trained more agents in parallel than it e cores in its node.

7.2.2 Parameters

Just as in the last chapter, we used a vanilla DQN and mostly identical settings. Table 7.1 lists all parameters specific to this experiment, i.e. the parameters for BOHB. The selection of running 16 iterations of BOHB was motivated by Falkner et al. (2018) who did the same in Cartpole. Other parameters, that were not introduced were left at their default. The parameters of DQN and the configurations spaces were chosen identically to the experiment in the last chapter. Again, the full configurations can be found in the accompanying repository.

We solely altered the training duration in the following ways. For the interpretation of the budget as the number of agents, we fixed the training duration to one million time steps. This was done as this duration captures the variance in learning speed that we found. Choosing a

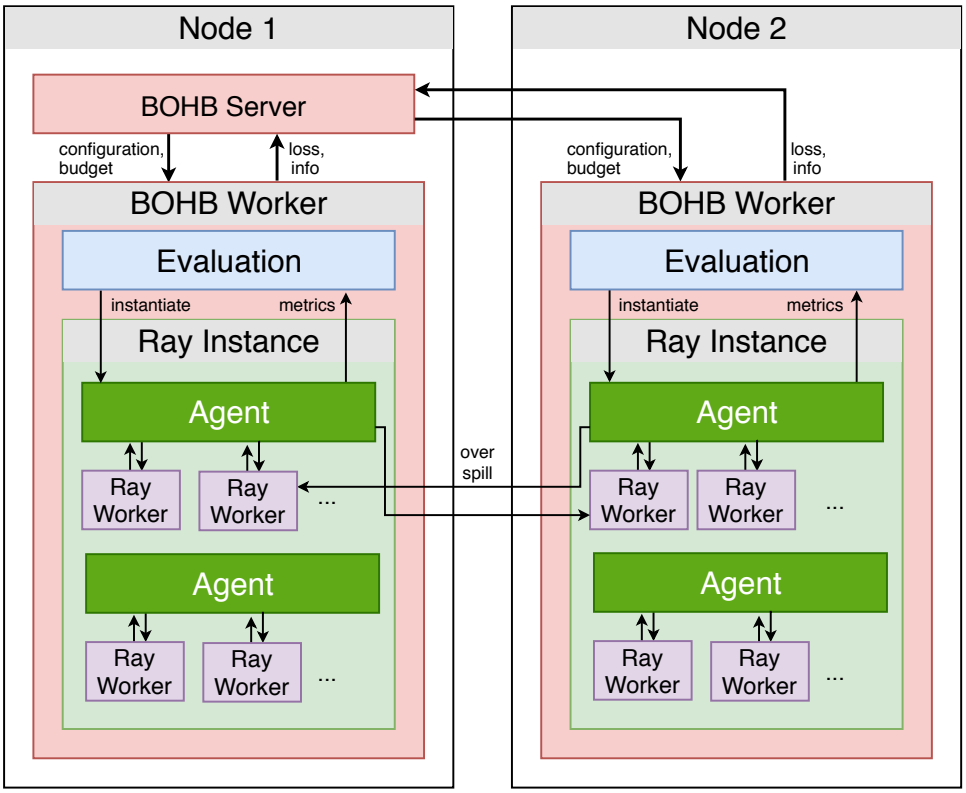


Figure 7.1: Sketch of the architecture of hyperparameter optimization experiment. Note, that the BOHB worker reports *loss*, which simply is negative performance. Upon receiving a configuration and a budget, a BOHB worker will start an evaluation by (sequentially or in parallel) instantiating agents with the given configuration and training them on the given budget. Using Ray's parallelization, each agent might use multiple Ray workers and each Ray instance may even *spill over* computations to other connected Ray instances. Note that all Ray instances share a common server for object storage which we omitted here for lucidity.

longer training duration would not only increase the computational cost, but also shift the found trade-off between learning speed and stability further towards stability. Then again, a shorter training duration would neither capture the whole variance in training speed, nor factor in the stability of the performance.

For the interpretation of the budget as the number of training iterations, we fixed one training iteration to last for 100,000 time steps, compared to 200,000 in the previous experiment. This was done because we speculated that it would improve the correlation between the performances between the budgets, as discussed.

We then used the same parameters for successive halving as before.

7.2.3 *Repetitions*

We repeated each BOHB run in Cartpole twice, giving us a total of eight BOHB runs. In MountainCar we only did one repetition each.

7.3 RESULTS

Meta Data For both environments, interpretations, optimizers and each repetition of the experiment, BOHB each evaluated around 54 configurations on budget 1, 33 on budget 3 and 26 on budget 9.

Best Configurations Table 7.2 depicts the best configurations that BOHB found for both interpretations and both optimizers in Cartpole. Note the big volatility for β_1 and β_2 between budgets and interpretations. Considering the far larger domain, the values for α are a lot more stable across budgets, interpretations and optimizers. For λ the values seem even more stable for a single run, but vary by three orders of magnitude between runs.

These results were consistent across environments, optimizers and repetitions. We also manually inspected the fastest learning configurations and found the values for both β_1 and β_2 to vary between as much as 0.1 and 0.9. Additionally both did not necessarily change in parallel - well performing configurations could for example have $\beta_1 \approx 0.1, \beta_2 \approx 0.9$ and vice versa.

Well performing learning rates however, seemed to mostly lie in $[1e{-}5, 7e{-}5]$ independently of environments, optimizers and repetition.

For AdamW, well performing values for the weight decay factor seemed to similarly lie within an interval, which however we found to be dependent on the environment at hand. In Cartpole it was around $[1e{-}5, 8e{-}3]$ and in MountainCar around $[1e{-}3, 3e{-}2]$.

Speedup As also illustrated in Figure 7.2, BOHB did find configurations for Adam that reached the performance ceiling in Cartpole within less than 30,000 time steps. Recall from Figure 6.4 that our default configuration failed to reach the performance ceiling, even with two million time steps of training. Recall further from Figure 6.5 that the best of

	Parameter	Value
BOHB	maximal budget b_{max}	9
	minimal budget b_{min}	1
	successive halving factor η	3
	fraction of random configurations ρ	$\frac{1}{3}$
	optimization iterations	16
	$n_{workers}$	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 BOHB 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.

	budgets			budgets		
	1	3	9	1	3	9
α	1.33e-4	9.20e-05	4.83e-05	3.87e-4	1.88e-05	1.88e-05
	2.89e-05	2.89e-05	3.69-05	2.10e-4	1.73e-4	1.18e-4
β_1	0.167	0.112	0.164	0.117	0.542	0.542
	0.297	0.297	0.840	0.179	0.132	0.121
β_2	0.460	0.500	0.313	0.758	0.655	0.655
	0.198	0.198	0.228	0.134	0.214	0.182
λ				2.01e-2	2.39e-2	2.39e-2
				1.16e-05	4.14e-05	2.15e-4
	Adam			AdamW		

Table 7.2: Rounded best configurations found by BOHB for Adam and AdamW. The upper value is the one for time steps $\propto$ budget, the one below for agents $\propto$ budget.

	budgets		
	1-3	1-9	3-9
Adam	0.03	0.77	0.49
AdamW	0.26	0.68	0.47

(a) time steps $\propto$ budget

	budgets		
	1-3	1-9	3-9
Adam	0.45	NaN	0.29
AdamW	-0.04	-0.65	0.58

(b) agents $\propto$ budget

Table 7.3: Budget correlations (Spearman). NaN signifies that a value could not be computed due to numerical reasons. For the cell in question, BOHB only reported five common configurations, apparently leading to a zero variance at one of the budgets.

our randomly sampled configurations reached the performance ceiling within not less than 150,000 time steps. Hence, while the default configuration failed to *solve* Cartpole within training on a single processor core for three and a half hours and the best randomly sampled configuration solved it within 16 minutes, the best configurations found by BOHB only needed about three minutes. This is a considerable speedup.

Budget Correlations The budget correlations, averaged over all BOHB runs in Cartpole are depicted in Table 7.3. As we expected, the finer resolution increased the correlation for the time-interpretation of the budget compared to our experiment in the previous chapter; especially for higher budgets, reaching values up to 0.77. Surprisingly the correlation for agent-interpretation of the budget got significantly worse, even reaching high negative values (-0.65) for AdamW.

Parameter Importance Table 7.4 lists the parameter importance averaged over the budgets and repeated BOHB runs in Cartpole. As can be seen, the local parameter importance reflects the strong influence of the learning rate that we also found. The fANOVA however estimates the

	Adam		AdamW	
	fANOVA	LPI	fANOVA	LPI
α	5,11	59.9	4.60	70.1
β_1	23.7	13.7	28.2	26.1
β_2	47.4	22.5	43.45	5.21
λ	-	-	18.6	11.8

Table 7.4: Parameter importance in Cartpole averaged over all BOHB runs as estimated via fANOVA and local parameter importance (LPI).

influence of β_1 and especially β_2 to be much higher. Both reckon λ to have only a minor importance.

7.4 DISCUSSION

Best Configurations Both the concrete values of Table 7.2 and their variance can be understood by considering Figure 7.2. It shows zoomed in learning curves for some of the best configurations that were sampled during one BOHB run for Adam. Associating the configurations with the respective course of the graph paints a clear picture. Both β_1 and β_2 vary greatly between well performing configurations with the changes in performance being below the noise level of the performance of a configuration. Consider the four curves in Figure 7.2 that are grouped together on the left. For example for the pink curve we have a value for β_1 of 0.12, while for the blue one it is at 0.88. Yet, at all times the performance values of all four configurations are apart less than 20 points. However, the standard deviation of the performance of a single configuration is as high as 20 in areas of large performance growth.

On the other hand the learning rate has a profound impact. The four quickly growing graphs that are grouped closely together each have a learning rate within $[3.69e{-}5, 4.55e{-}5]$. The configuration corresponding to the green graph that needs about twice as long to reach the top performance, however, has a learning rate α of 2.40e−5.

In Cartpole the best performing configurations for AdamW had a regularization strength λ in between 1e−5 and 8e−3. In MountainCar however, it was in between 1e−3 and 3e−2. This shows an interesting fact that we will take closer look at in the following chapter; it seems that the interval of well-performing regularization strengths and the general effectiveness of weight decay has to do with how quickly the agent has to react to an event in the environment. In Cartpole episodes are already terminated if the angle of the pole exceeds 12°. Hence, the agent needs to react quickly and vigorously to stabilize the pole or will be severely punished as the whole episode terminates. In MountainCar however, a slower reaction will simply worsen its reward but not be punished as severely. As we will discuss in more detail in the next chapter, we

found environments that require quick, vigorous action to favor lower regularization strengths.

Budget Correlation Furthermore, all configurations learn quick and reach a stable performance. This also means that their mean reward over the whole training duration will be very closely together. Consider again the example of the green and red curve from Figure 7.2. For simplicity we assume a total training duration of one million time steps and identical performance upon first reaching the top performance level. The initially faster rise of the red curves makes it reach the top performance level about 20,000 time steps earlier, which in the average over the whole training duration makes a difference of not more than 2%. The standard deviation of the performance of a group of 10 runs of the same configuration on the other hand easily surpasses 10%.

Additionally, the configurations in our robustness experiment were sampled completely at random, leading to a large variance in performance. BOHB on the other hand makes model-based picks, what further shrinks the variance between the performance of the configurations.

These factors make it hard for the optimizer to differentiate between the configurations and possibly explain the mediocre budget correlations.

The observation that the correlations are even worse for AdamW than for Adam (see Table 7.3) might be due to the fact that the weight decay introduces new dynamics that we did not explore deeply enough in our experiment in the previous chapter.

Parameter Sensitivity Looking at a single run gives the impression of a strong sensitivity of AdamW towards λ, as the optimal values are very stable across budgets (see Table 7.2). Comparing the values between both interpretations of the budget however, makes the opposite impression, as the values differ for as much as three orders of magnitude. Additionally, the configurations found for time-interpretation of the budget seem to favor both a larger learning rate and a weaker regularization. Due to the discussed high noise level in the performance values we are unsure as to whether this effect is due to different budget interpretations, shows a general interrelation of the learning rate and regularization strength, or is simply coincidental. However, the sensitivity of AdamW towards λ will be examined in the next chapter in some more depth.

As discussed above, we found the learning rate to play a big role, while β_1 and β_2 were virtually indifferent. This most likely has to do with the environments at hand. These parameters control the strength of the influence of the momentum on the parameter update. Hence, they are expected to have an influence if the terrain of the loss function is more complex - changing between flatter and rougher areas. The dynamics of both Cartpole and MountainCar however are seemingly so simple that this does not play a significant role.

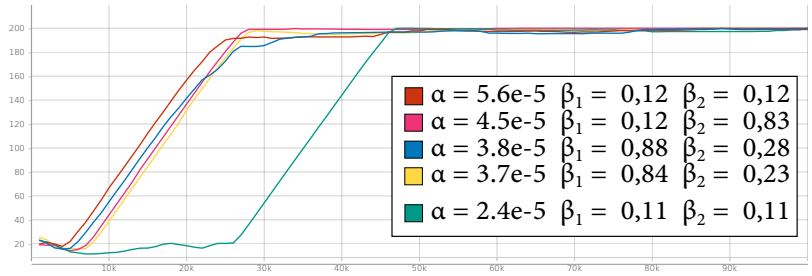


Figure 7.2: Magnified learning curves for five of the best configurations for Adam over the first 100,000 time steps. Notice, how close four of them lie together despite big differences in their respective β parameters. A medium strength change of the learning rate, however, has a noticeable effect, see the green curve.

Parameter Importance Qualitatively, the local parameter importance concurs with our manual analyses. The effects are not depicted as drastic as we see them, but the general trend seems to match. The fANOVA on the other portrays a situation that is contrary. We assume that this is due to the underlying random forest overfitting the performance data, that has a very low signal-noise ratio - as we have demonstrated.

Our results are inconclusive about λ however, as the low variance between budgets and the moderate parameter importance according to both measures might be due to model not exploring enough. Albeit, we will explore the influence of λ in the next chapter.

Additional Remarks Another crucial factor that is not evident from the mere averages presented in Table 7.3 and Table 7.4 is the massive volatility of these measures. For example, for one BOHB run the parameter importance of β_2 varied for as much as between 6.85 and 80.5 for two different budgets of the time-interpretation. Also, for the agents-interpretation the effect was similar, e.g. 20.1 and 70.5 at two different budgets. This extreme example once again highlights the big noise level in the performance of a single configuration. While the total magnitude of the noise is relatively low - at leas for the agents-interpretation of the budget - so is the actual performance difference between configurations, hence the signal. One of the central issues seems to be the performance metric which needs to capture both the learning speed and the stability of the performance. Finding such a metric is especially difficult as not all environments have a performance ceiling in a way that Cartpole has. A combined metric measuring both the learning speed up to a performance peak and the stability after that peak might help, but needs to trade-off between both according to the environment and the optimization methods. This again implies manual tinkering, which is what we try to eliminate in the first place. Our hope is that even simple metrics like the average episodic reward - that we chose - might work better in richer environments, such as ALE.

Generally, it is also important to keep in mind the influence of the strong parallelization of BOHB on the model-based configurations picks.

7.5 CONCLUSION

The application of BOHB yielded configurations that vastly outperformed both the default configurations and the best randomly sampled configurations for both optimizers. However, due to the insufficient difficulty of the environment, a large set of configurations could easily reach the performance ceiling. BOHB could still discriminate between configurations that quickly reach this performance ceiling and those who could not. But due to the performance metric it was difficult for it to discriminate between well performing configurations. As a consequence, it could not provide conclusive estimates of the hyperparameter importance values.

OPTIMIZER ANALYSIS

8

8.1 PROBLEM

In this chapter we make a closer analysis of Adam with and without weight decay, building on the results of the last two chapters. Our goal is to shed some light on the not well understood effects of weight decay on Adam in reinforcement learning. Furthermore we reexamine the evidence from our previous experiments to characterize the influence of Adam's other hyperparameters as well.

Additionally, we show how examining the development of the optimizers' moment tensors (and hence, their momentum) allows to contrast the stability of the optimizers. Finally, we compare the stability of the optima that the optimizers find.

8.2 HYPERPARAMETERS

8.2.1 Weight Decay

Experimental Setup

We executed some runs of AdamW in Cartpole and ALE (Breakout, Pong and Enduro) with different regularization strengths to study the effects of weight decay more closely. Remember, the default values are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\lambda = 2.5e{-}4$. Runs with the default values suggested that the default value for λ is in fact too high. Recall for example Figure 6.4 that shows how AdamW performs well worse than Adam. Therefore, we constructed a set of exponentially decreasing regularization strengths: $\{1e{-}4, 5e{-}5, 2.5e{-}5\}$ and tested these in both Cartpole and ALE. The configuration for DQN was the same as before, see Table 6.1.

Results

Weight decay in Cartpole The effect of weight decay in Cartpole can be seen in Figure 8.1. While a strong weight decay drastically decrease the anytime and final performance, a more mild weight decay has less of a negative impact. This is only up to a certain performance however: the weakest weight decay again only performed en par with the strongest. No configuration however surpasses the performance of vanilla Adam.

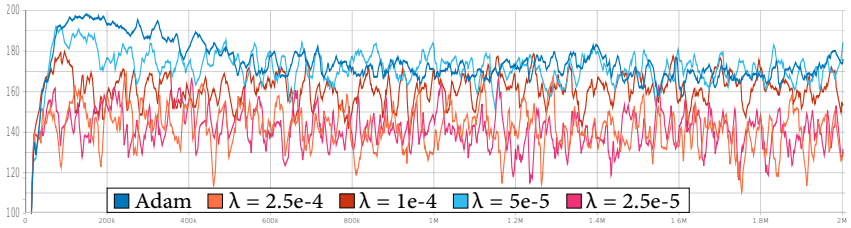
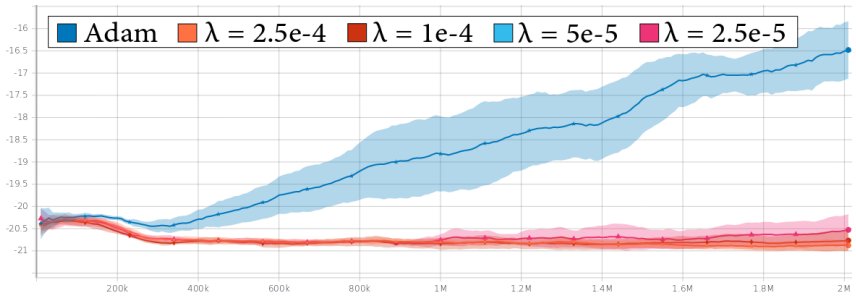


Figure 8.1: The mean episodic reward averaged over 10 runs in Cartpole for Adam and AdamW with different settings for λ (2.5e-4, 1e-4, 5e-5, 2.5e-5). We omitted the confidence intervals here for lucidity.

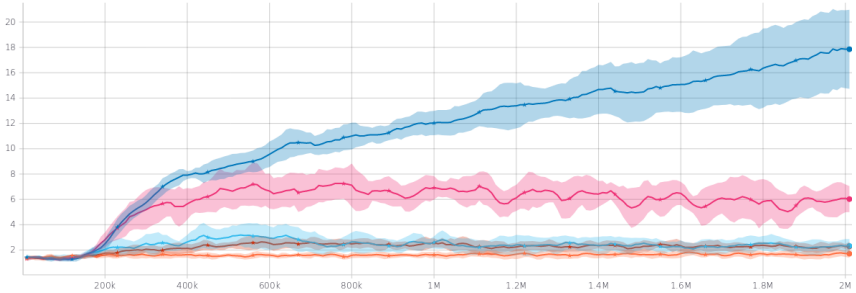
Weight decay in the Arcade Learning Environment For the ALE the influence is more complex, as can be seen in Figure 8.2. Especially in **Pong** (Figure 8.2a) the weight decay inhibits virtually any improvement in the reward. A lower regularization strength reduces this effect only marginally. All configurations of AdamW are by far outperformed by Adam. After two million time steps, Adam reaches an average reward of around −16.5, while all configurations of AdamW stay below −20.5. Additionally, the softest weight decay (λ = 2.5e−5) shows only a marginal improvements of less than 0.5 reward points over the strongest weight decay (λ = 2.5e−4).

In **Breakout** (Figure 8.2b) the situation is similar: the weight decay drastically impairs the performance. However, mitigating the regularization strength has a far greater benefit here compared to Pong. While Adam reaches an average reward of around 18 after two million time steps, most configurations of AdamW cannot surpass the initial reward level of 2. Only the weakest regularization strength can keep with Adam during the first 250,000 or so time steps, but fails to further improve afterwards. Hence, it reaches only an average reward of six at the end of the training.

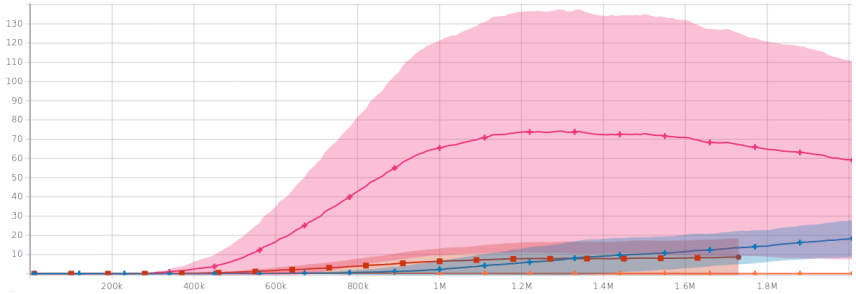
However, the situation is different for **Enduro** (Figure 8.2c). Strong weight decay (λ = 2.5e−4) also leads to the agent not improving its performance at all. A medium strength weight decay (λ = 1e−4) already performs about as well Adam. A weak weight decay (λ = 2.5e−5) even drastically improves the performance, reaching an average reward of 74 after 1.2 million time steps, while Adam and all other configurations of AdamW stay below a reward of 10 at this point. The effect vanished after this point however, with the configuration failing to improve the average reward further. Nevertheless, it is still by far outperforming Adam which only reaches a reward of below 20 after the full training of two million time steps. Albeit, the performance of the agent with weak weight decay is subject to large deviations ($\sigma \approx 60$ at the performance peak). Still, the effect seems to be a systematic one, as six out of the 10 runs performed well (reaching a reward of over 100) and four failing (not reaching a reward above 10). Hence, there are not simply one or two "lucky runs".



(a) The mean reward averaged over 10 runs in **Pong**, with confidence intervals.



(b) The mean reward averaged over 10 runs in **Breakout**, with confidence intervals.



(c) The mean reward averaged over 10 runs in **Enduro**, with confidence intervals.

Figure 8.2: The mean episodic reward averaged in Enduro and Breakout for Adam and AdamW with different settings for λ (2.5e-4, 1e-4, 5e-5, 2.5e-5). While in Pong even the slightest weight decay inhibits almost any learning progress and at also slows it down in Breakout, it does speed up the initial learning in Enduro. Nevertheless, the effect seems to diminish after some training time. Furthermore the regularization strength matters. Excessive weight decay also prevents learning here and only very faint weight decay gives a significant performance boost.

Discussion

Our results suggest that weight decay can in fact help with generalization in reinforcement learning. However the effectiveness is highly dependent on the environment at hand.

Weight decay helps generalization We found a mild weight decay (with λ = 2.5e−5) to drastically boost the performance of our DQN agent in Enduro. However, the same regularization strength significantly impaired the performance in Cartpole, Pong and Breakout. We think that this is the case because Enduro needs the agent to perform more abstraction and generalization. This is the case because of the changing visuals due to the changing background landscape and the

day-night cycle of the game. Weight decay is often applied to improve the generalization of neural networks. In DQN (and other reinforcement learning methods) however, the replay buffer already fulfills this role: by using experience from different points in time, the algorithm ensures not to overfit too much on single samples. We suspect that this is why after some time the positive effect of weight decay declines. See Figure 8.2c to see how after an initial speed up, the learning progress recedes after 1.2 million time steps. After a longer training time the replay buffer contains a more varied set of experience that help with generalization. Hence, the improvement in generalization brought forth by weight decay vanish. Analogously to learning rate scheduling, which is widely applied to improve the performance of machine learning system, decaying the regularization strength might combine the strengths of both methods.

Weight decay impairs strong action In other settings like Pong, however, the weight decay virtually hinders any learning. Rolling out the agents and watching their behavior reveals the reason. For the strong weight decay (with $\lambda = 2.5e{-}4$) for example, one can observe the following behavior. If the ball is about to cross the track on which the cart is moving with a big distance to the cart, the agent does not move the cart at all towards the ball. If the ball is about to cross the cart's track closely to the cart, the agent reacts by moving into the right direction, but reacts too slowly. The agent trained with only weak weight decay however (with $\lambda = 2.5e{-}5$) also misses the ball, but also reacts if the ball is farther away. This indicates that it has learned to predict the game state farther into the future and to take some action.

This matches with our results from the previous chapter. Recall that according to the configurations sampled by BOHB, well performing configurations in Cartpole had a λ-value one or two orders of magnitude below that of well performing configurations in MountainCar. Recall further, that in Cartpole the episode is terminated if the angle of the pole to the plummet exceeds 12° or the agent moves to far to the side. In Cartpole the reward is the episode length, in MountainCar it is the negative time until reaching the goal. In Cartpole however, the agent has to act quickly to stabilize the pole. If he acts slowly the pole will either exceed the maximum angle or he has to later move far on his track to catch it again. Hence, if he does not act fast, he is severely punished. In MountainCar however, the agent does experience not such a high pressure. Failing to act quickly which in MountainCar means the reversal of thrust upon reaching a certain height on either hill, might lengthen the time it takes to reach the goal but does not result in the episode terminating and giving him a catastrophic reward.

In conclusion, from our experience weight decay hurts the agent strongly whenever he has to act fast and vigorously to prevent an event that will take place in the farther future. Hence, it impairs his reaction time and capability to plan ahead.

Sensitivity towards weight decay The difference in sensitivity towards weight decay between Pong and Breakout that can be observed in Figure 8.2 seems to stem from the simple fact that in Breakout the ball sometimes arrives at the agent's starting position. This lengthens some episodes as the agent "accidentally" reflects the ball, giving him some reward, and allows the agent to collect more experience. In Pong however, this is not the case. Hence, the agent is stuck without gathering new experience. Although both Pong and Breakout have a similar game mechanic and both require the agent to react fast, this fact gives a good explanation for the different sensitivity towards weight decay.

There is a strong case for hyperparameter optimization Our results illustrate that different environments (e.g. the three games form the ALE) pose different challenges to the agent, leading to different optimal values of the hyperparameters like λ. Recall further from § 7.4 (p. 66) that there might be an interrelation between optimal settings for the learning rate and the regularization strength. Hence, there is a strong case in favor of automatically optimizing the hyperparameters of Adam and AdamW in these more complex environments.

8.2.2 Learning Rate

As we have seen in Figure 7.2 and discussed in § 7.4 (p. 66), the learning rate has a big influence on the performance of the agent. Increasing the learning rate from 2.40e−5 to somewhere within [3.69e−5, 4.55e−5] (roughly doubling it), did in fact halve the time that it took the agent to reach the performance ceiling in Cartpole. The effect itself is unsurprising. However, the fact that doubling the learning rate actually doubled the learning speed must only be seen as a lucky special case.

Further increasing the learning rate (e.g. to 6.7e−5) already leads to the instability in performance, exemplified in Figure 6.6 with the two blue graphs. For non-adaptive gradient based methods, this effect is well known; a too large learning rate can make the optimizer skip over optima and even lead to divergence of the Q-network and hence the policy. Figure 8.3 exemplifies this fact again. The agent with a learning rate of 3.67e−2 makes some initial learning progress but never even reaches the performance ceiling of the environment. Instead the policy fails to converge and the performance drops to an average reward of around 30. For comparison, a random policy achieves an average reward between 20 and 30.

The fact that we also observed this effect in the very simple example of Cartpole - that we found to be robust against other hyperparameters (e.g. the momentum parameters β_1 and β_2) - concurring with the results of Henderson et al. (2018b) who found that adaptive optimizers only have a small realm of effective learning rates (recall Figure 4.1).

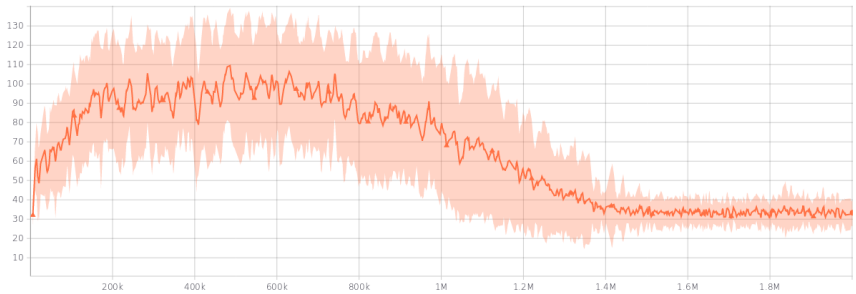


Figure 8.3: The mean episodic reward averaged over 10 runs in Cartpole for Adam with a learning rate of 3.67e−2. Notice how the large learning rates leads to some initial success and then not only fails to improve the performance further but makes the performance drop again to a level only slightly better than a random agent (that scores a reward of 20-30).

8.2.3 *Momentum Parameters*

As we have reported in § 7.3 (p. 64), illustrated in Figure 7.2 and discussed in § 7.4 (p. 66), even configurations of Adam and AdamW that show a performance difference below the noise level in Cartpole and MountainCar, can have have widely different momentum parameters β_1 and β_2. We found these variations to comprise almost the whole sample domain of $[0.1, 1]$ with observed values between 0.1 and 0.9. However, never did we observe momentum parameters > 0.9 in well performing configurations

Our results suggest that in Cartpole and MountainCar both Adam and AdamW are insensitive towards changes of the momentum parameters β_1 and β_2 within an interval of $[0.1, 0.9]$. We think that it is unsurprising that we have not observed any $\beta_1 > 0.9$ as this would imply that the current step with would have been decided almost entirely by the the past gradients and that the current gradient has virtually no influence. However, the great variance in well performing values is still surprising and implies that the strength of the influence of the momentum on the gradient update does not play a big role in these two environments.

We speculated that this might be due to the loss landscape being rather open instead of rough for both environments. However, as the momentum parameter β_1 also controls by how much the step width accelerates over flatter plains, this would imply that well performing configurations with small β_1 would either have to compensate for the missing speed up with a larger learning rate or as a result learn more slowly. We tested this hypothesis by correlating the values of β_1 with the learning rate α for the incumbent configurations found by BOHB - i.e. the best performing configurations for some budget run. We found a rank correlation of -0.434 for one of the BOHB runs but could not reproduce this result on the other runs. It remains unclear whether this is also due general noise level in the performance values or the absence of an actual effect.

Generally, the situation remains inconclusive about the influence of

the momentum parameters. We acknowledge that this is in part due to the noise level in our experiments and in part due to the environment-dependency of all discussed effect. Further research needs to be done here.

8.3 STABILITY

8.3.1 Development of the Momentum

Experimental Setup

The tracking of the momentum tensors drastically increases the computational burden of the computation, in our case increasing the wall clock time of a single training iteration of 1000 time steps from around ten seconds on a single processor core to around 20 minutes. More efficient implementations can surely speed up the process. For this thesis however, this meant that it was infeasible for us to track the momentum tensors in a usual run. To still be able to analyze the change of the tensors in some way, we increased the *sample batch size* of the agent from the default value four to 200. This parameter controls for how many time steps each current state of the agent interacts with the environment before being updated. Increasing it worsens the performance of the agent but saves computational cost. We then trained the agent with the default values for Adam and AdamW that we used before. For each optimizer we trained the agent ten times and aggregated the results as done in the previous chapters.

Results

Figure 8.4 shows the development of the mean episodic reward, the mean of the momentum tensors $\vec{m}$ and $\vec{v}$ and the Euclidean distance of the two to the last time step's values. Figure 8.4d and Figure 8.4e show how for both Adam and AdamW there is a lot of change in the momentum tensors between time steps 50,000 and 150,000 that then slowly vanishes. Figure 8.4c shows that the mean of $\vec{v}$ follows the same pattern, while Figure 8.4b illustrates that in the same time period the mean of $\vec{m}$ jumps frantically between positive and negative values. This indicates that the loss function is frequently changing signs in this area, resulting in a rough loss landscapes in which the optimizers slow down.

During this period there are only small but steady performance gains for both optimizers. As the turbulences dwindle, the gain in performance of both optimizers rise but become more instable.

The mean of the momentum tensors rise again for AdamW after around 400,000 time steps. Furthermore, is the standard deviation of the performance of the agent with AdamW (≈ 45) about twice as large as that for Adam (≈ 22) after 450,000 time steps.

Figure 8.5 shows the distribution plot and histogram for the momentum tensor $\vec{m}$ of Adam over the same time. The course of the distribution plot matches almost perfectly with that of the euclidean distance of $\vec{m}$ to the last time step. For the histogram, the bucket resolution is too low to give any useful information.

Discussion

Disclaimer on Conclusiveness Note that we only analyzed the development of the tensors for one specific hyperparameter setting. Therefore are our results not conclusive do discriminate between Adam and AdamW, but can be seen a primer for a more thorough investigation.

Stability The mean episodic reward (Figure 8.4a) has converged neither for Adam, nor AdamW. However, the larger variance in performance for AdamW suggests greater stability of Adam. While the Euclidean distance of $\vec{m}$ and $\vec{v}$ shrinks for both Adam and AdamW towards the end of the training, this is not the case for the mean of the of $\vec{m}$ and $\vec{v}$ for AdamW. The fact that the momentum tensors change less and less suggest that for all runs and both optimizers the Q-Networks gravitates towards a somewhat stable state. However, that the mean of $\vec{m}$ and $\vec{v}$ across different runs is in the end both higher for AdamW and shows a greater standard deviation, suggests that for AdamW the different runs reach different final states. This supports our hypothesis of the greater instability of AdamW.

Note though, that we have already found that the chosen value of $\lambda = 2.5\text{e} - 4$ is to large for Cartpole (remember for example Figure 6.4). Hence, we emphasize once more that this analysis serves more as a proof of concept on how tracking the momentum variables allows to compare the optimizers and is not conclusive to discriminate between Adam and AdamW.

Histograms and Distributions We find that both the histogram and distribution in Figure 8.5 do not add a lot of value to the analysis. The histograms proves the problem that we remarked in § 5.5 (p. 40): in order to aggregate the histograms across multiple runs, one has to fix the bins ex ante (i.e. before running the actual experiment). In order to chose more fitting bins, it seems necessary to run a pre-experiment to estimate the expected range of the values.

The distribution plot seems to simply confirm the trend shown by the development of the mean and Euclidean distance without adding further insight.

Implicit Learning Rate Scheduling The momentum introduces a mechanism akin to an *implicit learning rate schedule*. In flatter areas the momentum increases while shrinking in areas of strong volatility.

Calculating $\frac{\hat{m}}{\sqrt{\hat{v}}}$ allows to visualize this schedule.[1] We used the tracked mean of the momentum tensors to estimate the actual momentum. The results are illustrated in Figure 8.6.

First of all, the plot confirms the greater instability of AdamW as there are large changes in the momentum even after 400,000 time steps.

Second, it confirms out hypothesis on why there is only a small (but steady) learning progress between time steps 50,000 and 150,000: the volatility in $\vec{m}$ and the large magnitude of $\vec{v}$ (that hint at a rough loss landscape) lead to small momentum, hence smaller step sizes of the optimizers. Afterwards, the momentum increases slightly, making the performance gains larger but more volatile.

1. Note that we ignore ϵ here for convenience.

8.3.2 *Stability of the Optima*

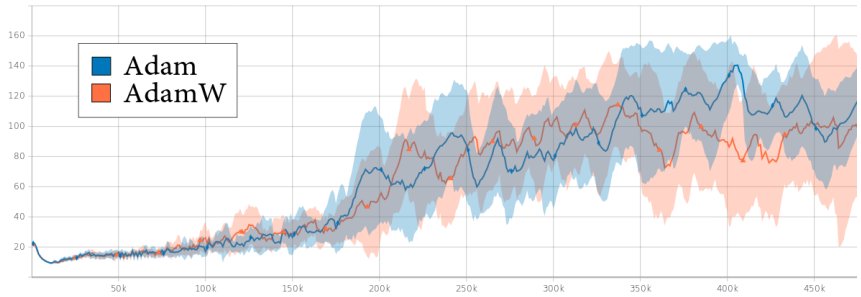
We trained the incumbent configurations (i.e. the best performing configurations) found by BOHB for Adam and AdamW in Cartpole for one million time steps. We trained for such a long time, so that instabilities of the configurations had time to arise. We then rolled out the trained agents for 100,000 time steps with different random initializations for each episode to judge the stability of the policies, hence the optima found by the optimizers.

Results

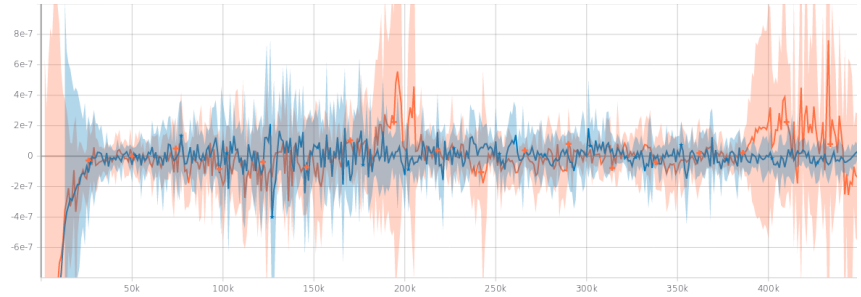
The incumbent configurations for Adam and AdamW found by BOHB both reached the performance ceiling after around 120,000 time steps and did not deviate again with increasing training duration. When rolling them out, both reached the maximum reward in each and every episode.

Discussion

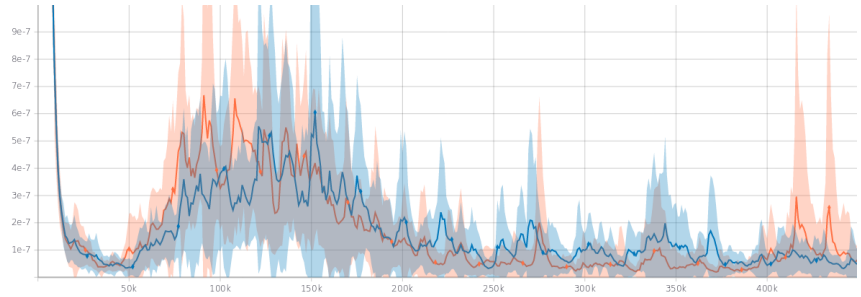
We found that in Cartpole the incumbent configuration of both Adam and AdamW achieve a constant perfect rollout score independently of the random initialization of the environment. We assume this to be due to simplicity of the environment. Unlike in more complex environments, the policy that the agent has to learn is simple and there probably are not many other policies acting as local optima in which the agent could get stuck. To see any differences, one would have to repeat these experiments in more complex environments, such as the Arcade Learning Environment. However, as we have seen are the optimizers sensitive towards the learning rate α and the regularization strength λ. Thus, a fair comparison is only possible after (automatically) tuning the hyperparameters of the optimizers. As discussed, this was not feasible for us due to a too high computational cost. Hence, further research is needed.



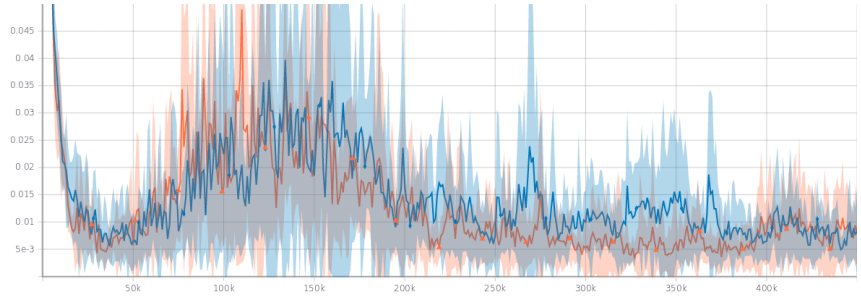
(a) The mean reward averaged over 10 runs with confidence intervals.



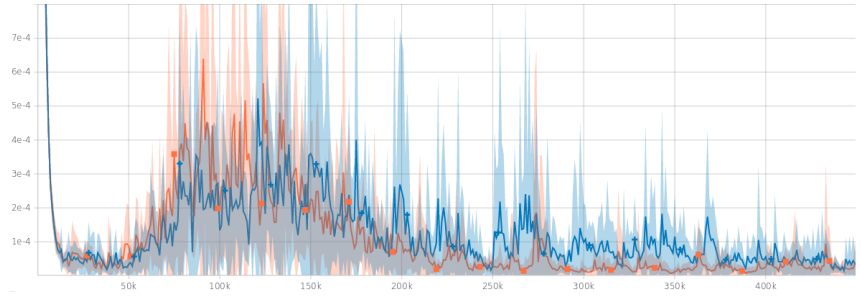
(b) The mean of the moment tensor $\vec{m}$.



(c) The mean of the moment tensor $\vec{v}$.

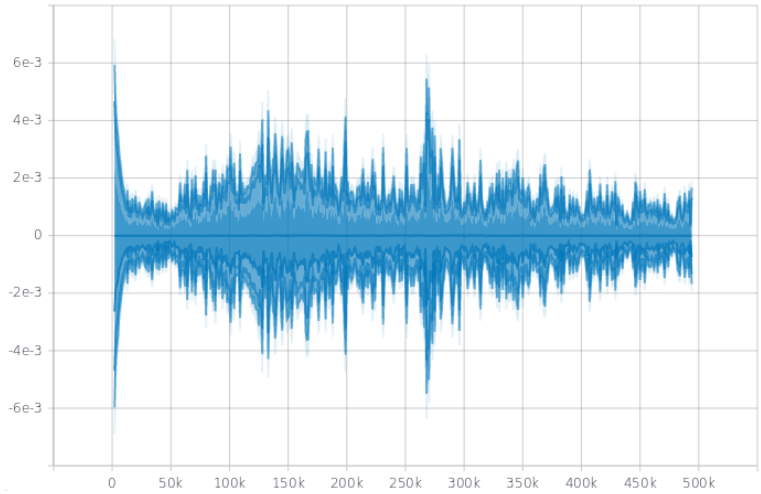


(d) The Euclidean distance of $\vec{m}$ to the last time step.

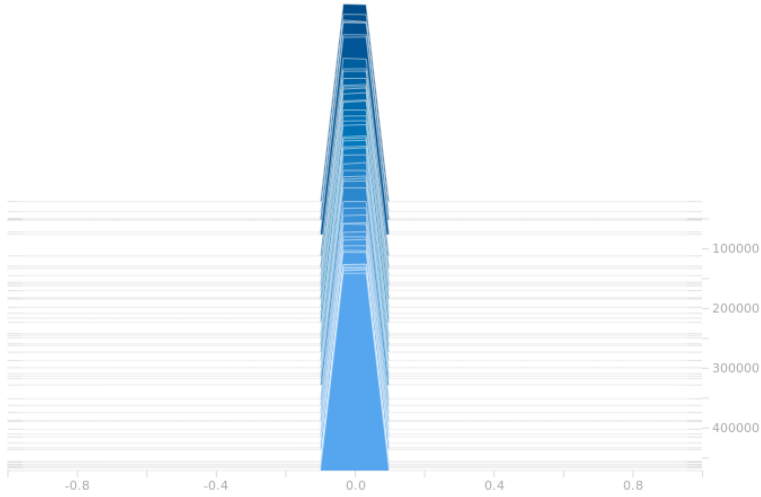


(e) The Euclidean distance of $\vec{v}$ to the last time step.

Figure 8.4: The development of the moment tensors $\vec{m}$ and $\vec{v}$ of the weights of the hidden layer of the Q-network. Aggregated for Adam and AdamW over ten runs each of the respective default configuration.



(a) The distribution of the scalar values of $\vec{m}$.



(b) The histogram of the scalar values of $\vec{m}$.

Figure 8.5: The development of the scalar values in the moment tensors $\vec{m}$ for Adam aggregated over ten runs of the default configuration.

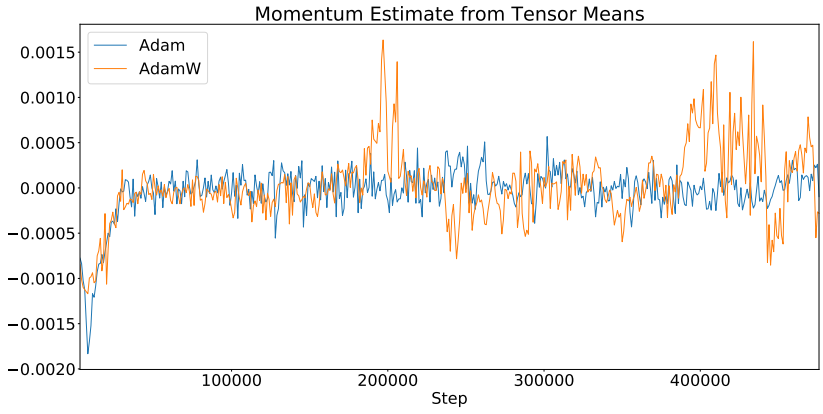


Figure 8.6: The mean momentum factor $\frac{\hat{m}}{\sqrt{\hat{v}}}$ as estimated from the mean of the moment tensors. While the value for Adam enters a somewhat stable state of 50,000 time steps, there is more volatility for AdamW.

8.4 CONCLUSION

Our results concur with the strong sensitivity of Adam towards the learning rate, attested by Henderson et al. (2018a). However, we did not find conclusive evidence for a significant influence of momentum parameters in Cartpole. For AdamW, the regularization strength has a big impact on the learning that depends on the environment at hand. A mild weight decay can help the generalization of the policy for environments with rich observation spaces like Enduro during initial learning when the agent's replay buffer does not yet contain diverse experiences. In environments like Cartpole, Pong and Breakout, where fast, vigorous actions and/or anticipation of a future event is needed, even a slight weight decay hurts the performance significantly.

Furthermore, we showed how tracking the moment tensors of Adam and AdamW can help to judge the stability of the optimization process. However, we did not find conclusive evidence about the characteristics of the optima preferred by either optimizer.

PART III

SUMMARY

CONCLUSION

9

Performance Estimation

Interpreting the budget of a hyperparameter optimization method as a number of agents to train and averaging the performance over these runs for a specific configuration, allows to reliably estimate the expected performance of that configuration. Interpreting the budget as the training duration on the other hand leads to very noisy estimates, unless tuned carefully. The anytime performance, i.e. the average episodic reward across the whole training duration, is a metric that allows to judge the quality of a configuration by respecting both the learning speed and the stability of the performance. However, it requires a trade-off between the emphasis on either of the two factors. Additionally, it fails to discriminate between well performing configurations in simple tasks with a performance ceiling like Cartpole. Hence, it is advised to select the metric by which to judge the quality of a configuration with respect to the environment at hand.

Automatic Hyperparameter Optimization

We found automatic hyperparameter optimization to reliably find well performing configurations for the examined optimizers. Due to the addressed difficulties regarding the performance metric, it was hard for the hyperparameter optimization algorithm to discriminate between well performing configurations. Nevertheless, in all investigated cases it did find configurations that vastly outperformed the default configuration. They also outperform the manually picked best configurations from a set of randomly sampled configurations by half an order of magnitude in terms of learning speed without sacrificing stability. Due to the good parallelization of state-of-the-art methods like BOHB, hyperparameter optimization in reinforcement learning can be feasible and brings great benefits. Due to it finding well performing configurations reliably, it takes off the burden of manual parameter tuning and thereby fosters the comparability of results across experiments.

Hyperparameter Importance

Our results concur with the sensitivity of adaptive gradient-based optimization methods towards their learning rate found by Henderson et al. (2018b). Due to fact that we only examined the simple Cartpole envi-

ronment in this regard, we did not find conclusive evidence about the importance of the momentum parameters. For AdamW we found the regularization strength to have an enormous impact on the performance that exact extend of which depends on the environment at hand.

Comparison of the Optimizers

The conducted experiments furthermore show that the general effectiveness of weight decay in reinforcement learning depends on the environment at hand. At the outset of training, when the agent does not yet posses diverse experience, a mild weight decay can help the generalization of the agent in environments that require abstraction from the observation, like Enduro. However, we found it to impair the agent's ability to react fast and vigorously and anticipate future events. This drastically hurts the agent's performance in environments like Pong and Breakout that require this kind of skill.

Although we did not find conclusive evidence on the difference in stability of both the optimization process and the preferred optima for the two optimizers, we showed how tracking the moment variables allows to comprehend and judge the optimization process.

9.1 FUTURE WORK

Comparing Adam and AdamW While we did not find conclusive evidence on the stability of either optimizer, we proposed ways by which to judge it. Deploying those on automatically optimized configurations of the optimizers might give answers to this question.

Environments Thus far, we only examined a limited set of environments. Cartpole and MountainCar in which we performed the hyperparameter optimization, however, do not require the agent to do a lot of abstraction and generalization. Our experiments should thus be repeated using more complex environments, such as the Arcade Learning Environment. We have demonstrated that the environments in ALE pose greater demands on the agent's generalization capabilities that might favor weight decay. However, our manual runs only give a snapshot picture of the situation. Automatically optimizing the hyperparameters in ALE requires a lot more computational resources but should give more insight. Especially, as the necessary policies are more complex than in Cartpole, allowing for more local optima, we hope for more insight into the optimizers' characteristics. Additionally we examined only a small subset of the ALE environments. Exploring the whole Arcade Learning Environment should give a more complete impression.

Due to the very active research community, new environments and challenges are put forward with great frequency. Recent examples include the games Rogue (Kanagawa and Kaneko, 2019) and Minecraft

(Guss et al., 2019). No single environment can give a complete picture of the challenges of reinforcement learning. Thus, repeating our experiments in a variety of them is important.

Optimizers Despite Adam's huge popularity in recent machine learning research, researchers were quick to propose other, even more sophisticated optimization methods. A very recent example is *AdaBound* (Luo et al., 2019), an extension of Adam that tries to combat Adam's poor generalization compared to vanilla stochastic gradient descent. An earlier attempt *AMSGrad* was put forward by Reddi et al. (2019). Zhang and Mitliagkas (2017) proposed *YellowFin*, a method to automatically tune momentum and learning rates for stochastic gradient descent. To gauge their value, rigorously comparing all these optimizers in a reinforcement learning setting, while keeping the recommendations of Henderson et al. (2018a) in mind, needs to be done.

As we did not find conclusive evidence on the influence of the momentum parameters, comparing Adam with optimizers without momentum, like vanilla stochastic gradient descent, in a similar setting might answer the emerging doubts about the benefits of using Adam.

Learning Rate Schedule Loshchilov and Hutter (2017) found AdamW to benefit greatly from learning rate scheduling, despite its adaptivity. Hence, it would be interesting how learning rate scheduling plays out in a reinforcement learning setting. Additionally, we speculated that scheduling the regularization strength might also improve an agent's performance.

Agents Besides the environment and the optimizer, the learning algorithm (agent) itself plays an important role. Whereas we had to limit ourselves to DQN, using other algorithms like DDPG (Lillicrap et al., 2015) not only opens up the possibility of investigating continuous environments but also introduces new dynamics. Repeating our experiments with a variety of learning algorithms is important, as - just with environments - a singular one will unlikely depict the whole problem class of reinforcement learning properly.

General As a reinforcement learning problem is in its core a numerical optimization problem, it is important to keep the *no free lunch theorem* (Wolpert et al., 1997) in mind. One will hardly find a singular algorithm that is superior to all other methods. Instead it is required to investigate and identify problems and algorithms that work well together.

Furthermore, Thornton et al. (2013) proposed a method to automatically jointly optimize both algorithm selection and hyperparameter optimization for classification tasks. Adopting their method to reinforcement learning could open up new ways towards a fully closed reinforcement learning pipeline.

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. 35.

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. 15.

Barto, Andrew G; Sutton, Richard S; and Anderson, Charles W. 1983. *Neuronlike adaptive elements that can solve difficult learning control problems*. In IEEE transactions on systems, man, and cybernetics, , no. 5, pp. 834–846. Cited on p. 3.

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 pp. 3, 22, and 38.

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. 25, 26, and 104.

Bellman, Richard. 1957. *A Markovian decision process*. In Journal of Mathematics and Mechanics, vol. 6, no. 5, pp. 679–684. Cited on p. 13.

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. 29.

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. 31 and 41.

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. 36.

CartPole v0. 2019. *CartPole v0 – OpenAI Github Wiki*. URL https://github.com/openai/gym/wiki/CartPole-v0. Cited on p. 36.

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. 19.

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. 4, 27, 28, 29, 30, 47, and 62.

Faust, Aleksandra and Francis, Anthony. 2019. *Google AI Blog*. URL https://ai.googleblog.com/2019/02/long-range-robotic-navigation-via.html. Cited on p. 3.

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. 25.

Goodfellow, Ian; Bengio, Yoshua; and Courville, Aaron. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org. Cited on p. 8.

Guss, William H; Codel, Cayden; Hofmann, Katja; Houghton, Brandon; Kuno, Noboru; Milani, Stephanie; Mohanty, Sharada; Liebana, Diego Perez; Salakhutdinov, Ruslan; Topin, Nicholay; et al. 2019. *The MineRL Competition on Sample Efficient Reinforcement Learning using Human Priors*. In arXiv preprint arXiv:1904.10079. Cited on p. 87.

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. 1.

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. 2, 4, 21, 26, 35, 45, 46, 49, 50, 82, and 87.

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. 4, 19, 20, 21, 26, 49, 75, and 85.

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. 25.

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. 4, 18, and 26.

Hochreiter, Sepp and Schmidhuber, Jürgen. 1997. *Flat minima*. In Neural Computation, vol. 9, no. 1, pp. 1–42. Cited on pp. 19 and 26.

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. 30.

Howard, Ronald A. 1960. *Dynamic Programming and Markov Processes*. In . Cited on pp. 11 and 13.

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. 30.

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 pp. 30 and 31.

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. 27 and 28.

Jaderberg, Max; Dalibard, Valentin; Osindero, Simon; Czarnecki, Wojciech M; Donahue, Jeff; Razavi, Ali; Vinyals, Oriol; Green, Tim; Dunning, Iain; Simonyan, Karen; et al. 2017. *Population based training of neural networks*. In arXiv preprint arXiv:1711.09846. Cited on p. 4.

Kanagawa, Yuji and Kaneko, Tomoyuki. 2019. *Rogue-Gym: A New Challenge for Generalization in Reinforcement Learning*. In arXiv preprint arXiv:1904.08129. Cited on p. 86.

Karpathy, Andrej. 2017. *Medium*. URL https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106. Cited on p. 18.

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. 9, 19, and 26.

Kingma, Diederik P and Ba, Jimmy. 2014. *Adam: A method for stochastic optimization*. In arXiv preprint arXiv:1412.6980. Cited on pp. 2, 17, 18, 19, 35, and 103.

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 pp. 9 and 19.

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. 27.

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. 35.

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. 40.

Lillicrap, Timothy P; Hunt, Jonathan J; Pritzel, Alexander; Heess, Nicolas; Erez, Tom; Tassa, Yuval; Silver, David; and Wierstra, Daan. 2015. *Continuous control with deep reinforcement learning*. In arXiv preprint arXiv:1509.02971. Cited on pp. 4 and 87.

Loshchilov, Ilya and Hutter, Frank. 2017. *Fixing weight decay regularization in adam*. In arXiv preprint arXiv:1711.05101. Cited on pp. 2, 18, 19, 20, 21, 22, 35, 48, and 87.

Luo, Liangchen; Xiong, Yuanhao; Liu, Yan; and Sun, Xu. May 2019. *Adaptive Gradient Methods with Dynamic Bound of Learning Rate*. In Proceedings of the 7th International Conference on Learning Representations. New Orleans, Louisiana. Cited on p. 87.

Machado, Marlos C; Bellemare, Marc G; Talvitie, Erik; Veness, Joel; Hausknecht, Matthew; and Bowling, Michael. 2018. *Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents*. In Journal of Artificial Intelligence Research, vol. 61, pp. 523–562. Cited on p. 38.

Mnih, Volodymyr; Badia, Adria Puigdomenech; Mirza, Mehdi; Graves, Alex; Lillicrap, Timothy; Harley, Tim; Silver, David; and Kavukcuoglu, Koray. 2016. *Asynchronous methods for deep reinforcement learning*. In International conference on machine learning, pp. 1928–1937. Cited on p. 20.

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. 1, 3, 14, 15, 22, 24, and 38.

Moore, Andrew William. 1990. *Efficient memory-based learning for robot control*. In . Cited on pp. 3 and 37.

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. 35.

Nesterov, Yurii. 1983. *A method for unconstrained convex minimization problem with the rate of convergence O (1/kˆ 2)*. In Doklady AN USSR, vol. 269, pp. 543–547. Cited on p. 4.

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. 1.

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. 36.

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. 1.

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. 18.

Reddi, Sashank J; Kale, Satyen; and Kumar, Sanjiv. 2019. *On the convergence of adam and beyond*. In arXiv preprint arXiv:1904.09237. Cited on p. 87.

Schaul, Tom; Quan, John; Antonoglou, Ioannis; and Silver, David. 2015. *Prioritized experience replay*. In arXiv preprint arXiv:1511.05952. Cited on p. 25.

Schulman, John; Levine, Sergey; Abbeel, Pieter; Jordan, Michael; and Moritz, Philipp. 2015a. *Trust region policy optimization*. In International Conference on Machine Learning, pp. 1889–1897. Cited on p. 20.

Schulman, John; Moritz, Philipp; Levine, Sergey; Jordan, Michael; and Abbeel, Pieter. 2015b. *High-dimensional continuous control using generalized advantage estimation*. In arXiv preprint arXiv:1506.02438. Cited on p. 20.

Schulman, John; Wolski, Filip; Dhariwal, Prafulla; Radford, Alec; and Klimov, Oleg. 2017. *Proximal policy optimization algorithms*. In arXiv preprint arXiv:1707.06347. Cited on p. 4.

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. 1.

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. 1.

Springenberg, Jost Tobias; Klein, Aaron; Falkner, Stefan; and Hutter, Frank. 2016. *Bayesian optimization with robust Bayesian neural networks*. In Advances in Neural Information Processing Systems, pp. 4134–4142. Cited on p. 4.

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. 1.

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. 1 and 8.

Tassa, Yuval; Doron, Yotam; Muldal, Alistair; Erez, Tom; Li, Yazhe; Casas, Diego de Las; Budden, David; Abdolmaleki, Abbas; Merel, Josh; Lefrancq, Andrew; et al. 2018. *Deepmind control suite*. In arXiv preprint arXiv:1801.00690. Cited on p. 20.

Tesauro, Gerald. 1995. *Temporal difference learning and TD-Gammon*. In Communications of the ACM, vol. 38, no. 3, pp. 58–69. Cited on p. 1.

Thornton, Chris; Hutter, Frank; Hoos, Holger H; and Leyton-Brown, Kevin. 2013. *Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms*. In Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 847–855. ACM. Cited on p. 87.

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. 17.

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. 25.

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. 25.

Watkins, Christopher JCH and Dayan, Peter. 1992. *Q-learning*. In Machine learning, vol. 8, no. 3-4, pp. 279–292. Cited on p. 14.

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. 18.

Wolpert, David H; Macready, William G; et al. 1997. *No free lunch theorems for optimization*. In IEEE transactions on evolutionary computation, vol. 1, no. 1, pp. 67–82. Cited on p. 87.

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. 24 and 25.

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. 21.

Zhang, Jian and Mitliagkas, Ioannis. 2017. *Yellowfin and the art of momentum tuning*. In arXiv preprint arXiv:1706.03471. Cited on p. 87.

INDEX

APPENDICES

THE ROLE OF EPSILON IN ADAM

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

— Anonymous

As discussed in § 4.3 (p. 26), 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 further argued that optimizing this hyperparameter without clear reasoning hampers the comparability of results across publications. More fundamentally, this is a prime example for a profound lack of basic understanding of the characteristics of Adam. This situation should be resolved by either accomplishing a deeper understanding of the situation at hand or demonstrating that the effect is in practice neglectable. 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, that $\hat{v}$, $\hat{m}$ denote the bias-corrected values of m, v.

Recall further the parameter update rule for Adam[1]:

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

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 $\Delta_{\theta_t} = \alpha \cdot \hat{m}_t / \sqrt{\hat{v}}$ of the parameter update is subject to an upper bound of

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

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

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

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

The Influence of Epsilon 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

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

which is dependent on $\hat{v}_t$. 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 v_t is small, a larger ϵ will shrink the effective steps size more strongly, whereas having a much smaller impact for larger momentum values. Thus, a larger epsilon will slow down the optimizer in flatter terrain, whereas 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.

To be precise, the step size is given by:

$$|\Delta_{\theta_t}| = \left| \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right| \tag{A.5}$$

$$= \alpha \cdot \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} \cdot \frac{|m_t|}{\sqrt{v_t} + \sqrt{1 - \beta_2^t} \cdot \epsilon} \tag{A.6}$$

As argued above, in smoother areas - where v_t is small - ϵ will have a stronger shrinking effect on the step size than in terrain where the gradients are large or even have rapidly changing signs - thus having large v_t. From the equation, we would also expect an interaction effect with the choice of β_2.

The Counterargument Against the empirically found benefits of tuning ϵ stands our initial apprehension that the tuning of this additional hyperparameter beyond its theoretical intention and without an understanding as to how this affects the performance, not only increases the risk of overfitting but also exacerbates the comparison of results.

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- and β-parameters.

Jointly optimizing ϵ together with the other hyperparameter using BOHB and then analyzing the parameter importance metrics might give some practical insight. However, we refrained from doing so for now, as in our experiments the estimates of the hyperparameter importance values were too noisy to be conclusive, anyway.

QUICK START GUIDE

In this chapter, we give a short introduction to how to recreate 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

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 Python3 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 targets `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 the customization the experiment's parameters. 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.

Custom environments should implement the interface provided by OpenAI Gym. They can then be passed to the agent instead of passing an environment name as a string. Alternatively, they can be registered in Ray [1] and then be passed using their name, which allows to set them via the runners parameters directly from the command line.

1. See ray.readthedocs.io/en/latest/rllib-env.html for instructions.

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`:

```
1 import CUSTOM_AGENT, DEFAULT_CONFIG
2
3 class CustomWorker(RllibWorker):
4   def __init__(self, **kwargs):
5     super().__init__(**kwargs)
6     self.agent = CUSTOM_AGENT
7     self.agent_name = CUSTOM_AGENT_AGENT
8     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 BOHB. 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 BOHB worker to pass the configuration to the optimizer.

Configuration Space To actually work with BOHB thought, the worker has to also implement the following method, generating a configuration space for BOHB to sample from.[2]

2. 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 BOHB worker however, will automatically do that.

B.3.3 Optimizers

To use a custom optimizer, the easiest way is 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.[3]

3. 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.