DERSTANDING OPTIMIZATION *in* EINFORCEMENT 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. 15. 2019

Examiner

Prof. Dr. Frank Hutter

Supervisor

Raghu Rajan

DECLARATION

This thesis was typest using the incredible template created and released into the public domain by Eivind Uggedal, found at https://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 **?**.

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

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

ABSTRACT

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

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

TODO

<div align="right">

Jan Ole von Hartz
Freiburg, Germany
May 2019

</div>

INTRODUCTION

According to **?** the field of *reinforcement learning* studies learning "how to map situations to actions—so as to maximize a numerical reward signal", and refers - besides the field of study - to both to the problem class itself, as well as a class of solution methods. The usually used formalism is that of an *agent* interacting with an *environment* via a fixed set of *actions* and only observing 1. a numerical reward signal and 2. the state of the environment or an observation which describes part of it. In recent years, *reinforcement learning* hast not only seen a surge of research, but has also produced a number of remarkable results, such as programs learning to control a helicopter (**?**), locomotion tasks (**?**), in-hand manipulation of objects (**?**) and playing games such backgammon (**?**) Atari video games (**?**) and Go (**??**). 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. While allowing for great scientific progress, the use of deep neural networks also introduces many new and opaque problems, some of which we try to tackle in thesis. For one, solving machine learning problems usually involves the selection of a as well suited as possible function from a class of functions by minimizing some loss function defined over the output of said function for a set of data points. Hence, the training of a deep neural network has in its core a *numerical optimization problem*: finding a suited parametrization of the network to achieve the best possible performance (the lowest possible loss) on unseen data by using a given data set as an approximation of the general data distribution and optimizing the networks performance on it. This is usually done via *gradient-based methods*, such *gradient descent* and variations of it. Reinforcement learning however introduces additional problems; unlike in *supervised learning*, the training data is not given, but must be produced by agent by engaging with the environment in a trial-and-error fashion. Since the deep neural networks are used by the agent to e.g. approximate a function that estimates the reward of a certain action in a certain state and these estimations change drastically over time, the numerical optimization of these networks is very unstable. A systematic problem of gradient-based methods is that unlike analytical methods they are only suited to find local optima, not global optima. In reinforcement learning the loss landscape is usually very rough and provides plenty of local optima to get stuck in. Furthermore, it is not well understood, how different gradient-based methods compare in reinforcement learning, especially since these are also very sensitive to the settings of their hyperparameters. A common call thus is for the development of hyperparameter agnostic algorithms - algorithms that adapt their hyperparameters themselves during runtime (**?**). For this thesis we

1. compared different gradient-based optimizers in terms of

    a) the final performance of their produced network configuration

    b) the stability of these optima

    c) their sensitivity towards their hyperparameters

    d) their stability in training

    to try to get an understanding of their characteristics in a in deep reinforcement learning setting.

2. Using a hyperparameter optimization method, automatically find well performing hyperparameter settings for the optimizers to make a step towards hyperparameter agnosticism.

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

Next, we will introduce the necessary background in a logical order; starting with the mathematical foundation, followed by the used algorithms and finally their implementation.

PART I

BACKGROUND

REINFORCEMENT LEARNING

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

2.1 THE MATHEMATICAL MODEL

In large parts of the current research, as well as this thesis, instances of this problem class are formalized using *Markov Decision Processes* (**?**). We thus now define this and related notions.

2.1.1 *Markov Decision Processes and Policies*

Definition 1 (Markov Process) *A* Markov Process *is a 2-tuple* $(\mathcal{S}, \mathcal{P})$, *where* $\mathcal{S}$ *is a finite 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* $\mathbf{P}[s_t \mid s_{t-1}, s_{t-2}, \ldots, s_0] = \mathbf{P}[s_t \mid s_{t-1}]$.

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* $\mathcal{R}_a(s, s') = \mathsf{E}[R_t \mid s_{t-1} = s, s_t = s', a_{t-1} = a]$, *with* R_t *being the reward received at point t, which can be a function mapping* s_{t-1}, s_{t-1} *and* s_t *or* s_{t-1} *and* s_t *and* a_{t-1} *to some real number.*

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* $s \in \mathcal{S}$: $s \mapsto p(\mathcal{A} = a \mid s)$.

Reinforcement learning problems are then often modeled as interpreting a *Markov Decision Process M* as an environment in which an Agent A can take Actions $a \in \mathcal{A}_M$ and observe the ensuing reward $r \in \mathbf{R}$ and state $s \in \mathcal{S}_{\mathcal{M}}$. Its goal then usually is to find an optimal policy $\pi^\star$ to maximize some cumulative function, usually the total discounted reward $\mathtt{R} = \sum_{t=0}^{T-1} \gamma^t R_{a_t}(s_t, s_{t+1})$, where T is called the *horizon* of the problem.

2.1.2 Value Functions and the Bellmann Equation

Notably, a policy implies a transition distribution over the state space of a Markov Decision Process M. This property allows one 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 4 (State-Value Function) *The* state-value 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:* $V^{\pi}(s) = \mathsf{E}_{\pi}[\sum_{t=0}^{T-1} \gamma^t R_{a_t}(s_t, s_{t+1}) \mid s_0 = s]$.

Definition 5 (State-Action-Value Function) *The* state-action-value function $V^{\pi}(s)$ *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* a *and following* π *thereafter:* $Q^{\pi}(s,a) = \mathsf{E}_{\pi}[\sum_{t=0}^{T-1} \gamma^t R_{a_t}(s_t, s_{t+1}) \mid s_0 = s, a_0 = a]$.

The *state-action-value function* (often called Q-function) fulfills the following recursive property (a similar property holds for the *action-value function*):

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

This famous property called *Bellmann Equation* allows to, given the reward function R, iteratively compute the Q-function (or V-function) (**?**). Remember, that the goal for the agent is to learn the optimal policy $\pi^{\star}$, which corresponds to an optimal *state-action-value function* $Q^{\star}$. The optimal policy $\pi^{\star}$ is the one that maximizes the corresponding Q-function $Q^{\star}$. As the *Bellmann Equation* illustrates, maximizing Q under policy π is achieved by greedily selecting the action a in state s that maximizes $R(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s,s') \sum_{a' \in \mathcal{A}} \pi(a' \mid s') \cdot Q^{\pi}(s',a')$, yielding an improved policy π': $\pi'(s,a) = \begin{cases} 1 & \text{, if } a = \arg\max_{\hat{a} \in \mathcal{A}} Q(s,\hat{a}) \\ 0 & , else \end{cases}$. Hence, just as $Q^{\star}$ can iteratively be computed, $\pi^{\star}$ can be iteratively computed as well, e.g. via *dynamic programming* (**?**).

2.2 APPLICATION OF THE MODEL

These lessons learned in the exact solution of the calculation 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 introduce two approaches that the algorithms that we introduce later make use of: *Value Base Methods* and *Actor-Critic Methods.*

2.2.1 *Value Based and Actor-Critic Methods*

To learn an approximation to the optimal policy $\pi^\star$ in an environment M an agent A can thus estimate $Q^\star$ via bootstrapping and iteratively improving its estimates, which is at the heart of so called *Value Based Methods*, such as *Q-learning* (**?**).

As the selection of the policy often happens in a greedy manner, 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 V-function and the policy (instead of producing the current policy via greedy selection from the current estimation of Q), which is at the heart of so called *actor-critic methods*. Another issue with continuous action spaces is that the runtime of the iterative update of the Q-function is at least linearly dependent on the size of the action space (see **??**) . 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.

2.2.2 *Practicality of the Markov Property*

As noted by **?**, the underlying assumption of the *Markov Property* does often not hold in practical applications, since it requires full observability of the states. While algorithms that use *Partly Observable Markov Decision Processes* exist, full observability is often simply assumed as an approximation to the actual partly observable environment. An example of such an approximation can be found in **?**; using the video output of an Atari game simulator the authors preprocessed the observations by combining four video frames into a single observation to approximate state features such as velocity of game objects.

2.2.3 *Scope of this Thesis*

In this thesis we will examine *deep reinforcement learning* algorithms - reinforcement learning algorithms that make use of *deep neural networks* as function approximators - based on *value based methods* and *actor-critic methods*.

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

ALGORITHMS AND METHODS

3.1 NUMERICAL OPTIMIZATION

Deep reinforcement learning usually involves the *numerical optimization* of one or more *deep neural networks* as function approximators with regards to some loss function. Numerical optimization in deep reinforcement learning usually makes use of *gradient-based methods*, since gradients can be easily computed for deep neural networks. 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.

3.1.1 *Adam: Adaptive Moment Estimation*

Method

Adam (**?**) is a first-order gradient-base optimization methods that computes "individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients" (**?**). 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 **??**. Until reaching some stopping criterion (e.g. convergence), in each time step Adam computes the current gradients, which in machine learning is usually done on some minibatch, updates its moment estimates and uses them to update the parameters it optimizes.

Popularity

? 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 (**?**) in terms of convergence of the training cost. Adam hat hence quickly gained in popularity. According to a survey conducted by **?** of 28,303 machine learning papers published on arxiv.org between 2012 and 2017, Adam was used in about one in four papers, making it the most popular optimization method by far.

```
**Hyperparameters:** step size $\alpha \in \mathbf{R}$, decay rates $\beta_1, \beta_2 \in \mathbf{R}$,
                     division stabilization constant $\epsilon > 0$
**Given:** initial parameter vector $\theta_0 \in \mathbf{R}$,
           objective function series $\left(f_n\left(\cdot, \vec{\theta}\right)\right)_{n=0,1,\ldots}$
**Optional:** schedule multiplier series $(\eta_n)_{n=0,1,\ldots}$
**Init:** $t \leftarrow 0$; $\vec{m}_0, \vec{v}_0 \leftarrow \vec{0}$ ;  (time step and moment vectors)
1  **while** *stopping criterion not met* **do**
2      $t \leftarrow t + 1$;
3      $\vec{g}_t \leftarrow \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1})$ ;     (computed on minibatch $\vec{m}$)
4      $\vec{m}_t \leftarrow \beta_1 \cdot \vec{m}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t$;     (moment estimation)
5      $\vec{v}_t \leftarrow \beta_2 \cdot \vec{v}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t^2$;
6      $\hat{\vec{m}}_t \leftarrow \vec{m}_t / (1 - \beta_1^t)$;     (zero-bias correction)
7      $\hat{\vec{v}}_t \leftarrow \vec{v}_t / (1 - \beta_2^t)$;
8      $\vec{\theta}_t \leftarrow \vec{\theta}_{t-1} - \eta_t \cdot \alpha \cdot \hat{\vec{m}}_t / (\sqrt{\hat{\vec{v}}_t} + \epsilon)$;     (parameter update)
9  **end**
10 **return** $\theta_t$
```

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

Limitations

Despite its advantages and popularity, **?** 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. **?** have since proposed extensions of Adam to combat its shortcomings.

Open Questions

Despite these ongoing developments, the exact properties of Adam and the influence of its hyperparameters are not yet well understood. **?** 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. **?** 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". These findings seem contradictory to us, as flat minima are typically thought of to be more robust than steeper ones (**??**) and

Adam is advertised to be more robust against settings of its learning rate exactly due to the individual adaptation (**?**). **?** also challenged the idea of flat minima generalizing better. 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 **?**.

? 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* (**?**), two popular methods to regularize network parameters to avoid overfitting. While both are equivalent for stochastic gradient descent, they are not for Adam. And while naive implementations often just apply L_2-regularization, Adam benefits far more from weight decay.

3.1.2 *AdamW: Adam with Weight Decay*

Method

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

$$\vec{\theta}_t = \vec{\theta}_{t-1} - \alpha \cdot \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1}) \tag{3.1}$$

weight decay introduces an additional factor λ that with time exponentially decays the parameter values to regularize their growth. We follow the notation of **?** here:

$$\vec{\theta}_t = (1 - \lambda) \cdot \vec{\theta}_{t-1} - \alpha \cdot \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1}) \tag{3.2}$$

Combining this technique with the Adam algorithm (see algorithm **??**), gives algorithm **??**, called *AdamW* by **?**. The authors showed that while for stochastic gradient descent L_2-regularization and weight decay are equivalent up to rescaling of the learning rate, this is not the case for Adam. This can easily be seen intuitively by changing line **??** in algorithm **??** to

$$\vec{g}_t \leftarrow \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1}) + \boxed{\lambda \cdot \theta_{t-1}} \tag{3.3}$$

as would to the case for L_2-regularization. Since the gradient is used afterwards for the moment estimation, $\lambda \cdot \theta_{t-1}$ has a non-linear influence on the parameter update. See **?** for a formal prove.

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, while still allowing the further improvements via the use of *scheduled learning rate multipliers*. **?** confirmed these results and offered mechanisms by which these improvements are achieved.

```
**Hyperparameters:** step size $\alpha \in \mathbf{R}$, decay rates $\beta_1, \beta_2 \in \mathbf{R}$,
                     division stabilization constant $\epsilon > 0$,
                     decay factor $\lambda \in [0, 1)$
**Given:** initial parameter vector $\theta_0 \in \mathbf{R}$,
           objective function series $\left(f_n\left(\cdot, \vec{\theta}\right)\right)_{n=0,1,\dots}$
**Optional:** schedule multiplier series $(\eta_n)_{n=0,1,\dots}$
**Init:** $t \leftarrow 0$; $\vec{m}_0, \vec{v}_0 \leftarrow \vec{0}$ ;  (time step and moment vectors)
1  **while** *stopping criterion not met* **do**
2      $t \leftarrow t + 1$;
3      $\vec{g}_t \leftarrow \nabla_{\vec{\theta}} f_t(\vec{m}, \vec{\theta}_{t-1})$ ;        (computed on minibatch $\vec{m}$)
4      $\vec{m}_t \leftarrow \beta_1 \cdot \vec{m}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t$;        (moment estimation)
5      $\vec{v}_t \leftarrow \beta_2 \cdot \vec{v}_{t-1} + (1 - \beta_1) \cdot \vec{g}_t^2$;
6      $\hat{\vec{m}}_t \leftarrow \vec{m}_t / (1 - \beta_1^t)$;        (zero-bias correction)
7      $\hat{\vec{v}}_t \leftarrow \vec{v}_t / (1 - \beta_2^t)$;
8      $\vec{\theta}_t \leftarrow \boxed{(1 - \lambda)} \cdot \vec{\theta}_{t-1} - \eta_t \cdot \alpha \cdot \hat{\vec{m}}_t / (\sqrt{\hat{\vec{v}}_t} + \epsilon)$;        (parameter
        update)
9  **end**
10 **return** $\theta_t$
```

Algorithm 2: AdamW. Yielded by combining Adam (see algorithm **??**) with weight decay (see equation **??**).

Decoupling of Parameters **?** further showed how the usage of weight decay over L_2-regularization decouples the learning rate and regularization hyperparameter. This is an incredibly valuable lesson. Recall that **?** found a strong dependence of the performance of reinforcement learning algorithms on the hyperparameter settings of their optimizers. **?** 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.

3.2 AGENTS

3.3 HYPERPARAMETER OPTIMIZATION

TOOLS AND LIBRARIES

Ref to henderson about code base influence etc.

4.1 NUMERICAL OPTIMIZATION

4.2 AGENTS

4.3 ENVIRONMENTS

4.4 HYPERPARAMETER OPTIMIZATION

4.5 VISUALIZATION AND EVALUATION

PART II

PERFORMANCE ESTIMATION

PROBLEM

EXPERIMENTAL SETUP

RESULTS

DISCUSSION

PART III

OPTIMIZER ANALYSIS

PROBLEM

EXPERIMENTAL SETUP

RESULTS

DISCUSSION

PART IV

SUMMARY

CONCLUSION

None

APPENDICES

none