Deep Reinforcement Learning for Continuous Control

Deep Reinforcement Learning fuer kontinuierliche Regelungen
Bachelor-Thesis von Simon Ramstedt aus Frankfurt am Main
Tag der Einreichung:

1. Gutachten: Prof. Dr. Gerhard Neumann
2. Gutachten: Prof. Dr. Jan Peters
3. Gutachten: MSc. Simone Parisi

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Deep Reinforcement Learning for Continuous Control
Deep Reinforcement Learning fuer kontinuierliche Regelungen

Vorgelegte Bachelor-Thesis von Simon Ramstedt aus Frankfurt am Main

1. Gutachten: Prof. Dr. Gerhard Neumann
2. Gutachten: Prof. Dr. Jan Peters
3. Gutachten: MSc. Simone Parisi

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 1. Mai 2016

_____

(Simon Ramstedt)

Abstract

Reinforcement learning is a mathematical framework for agents to interact intelligently with their environment. In this field, real-world control problems are particularly challenging because of the noise and the high-dimensionality of input data (e.g., visual input). In the last few years, deep neural networks have been successfully used to extract meaning from such data. Building on these advances, deep reinforcement learning achieved stunning results in the field of artificial intelligence, being able to solve complex problems like Atari games [1] and Go [2]. However, in order to apply the same methods to real-world control problems, deep reinforcement learning has to be able to deal with continuous action spaces. In this thesis, *Deep Deterministic Policy Gradients*, a deep reinforcement learning method for continuous control, has been implemented, evaluated and put into context to serve as a basis for further research in the field.

Zusammenfassung

Reinforcement-Learning ist ein mathematischer Rahmen, um intelligent mit ihrer Umgebung interagierende Agenten zu erzeugen. Regelungsprobleme in realen Umgebungen sind dabei wegen stark verrauschten, hochdimensionalen Eingabedaten (z.B. Video) besonders anspruchsvoll. In den letzten Jahren wurden dafür jedoch erfolgreich neuronale Netze benutzt. Deep-Reinforcement-Learning (Reinforcement-Learning mit neuronalen Netzen) hatte bereits große Erfolge in der künstlichen Intelligenz und war in der Lage komplexe Probleme wie Go [2] oder Atari-Spiele [1] zu lösen. Um diese Methoden aber in der echten Welt anwenden zu können, muss Deep-Reinforcement-Learning mit kontinuierlichen Handlungsräumen umgehen können. Deshalb wurde in dieser Thesis *Deep Deterministic Policy Gradients*, eine Deep-Reinforcement-Learning-Methode für kontinuierliche Regelungen implementiert, evaluiert und in Bezug zu anderen Methoden gesetzt.

Acknowledgments

I thank my supervisors Simone Parisi and Gerhard Neumann for their patience and helpful discussions, with special thanks to Simone for his support during time-critical periods.

Contents

Figures and Tables

List of Figures

List of Tables

1 Introduction

Reinforcement learning is a mathematical framework for agents to interact intelligently with their environment. Unlike supervised learning, where a system learns with the help of labeled data, reinforcement learning agents learn how to act by trial and error only receiving a reward signal from their environments. A field where reinforcement learning has been prominently successful is robotics [3]. However, real-world control problems are also particularly challenging because of the noise and high-dimensionality of input data (e.g., visual input). In recent years, in the field of supervised learning, deep neural networks have been successfully used to extract meaning from this kind of data. Building on these advances, deep reinforcement learning was used to solve complex problems like Atari games and Go. Mnih et al. [1] built a system with fixed hyperparameters able to learn to play 49 different Atari games only from raw pixel inputs. However, in order to apply the same methods to real-world control problems, deep reinforcement learning has to be able to deal with continuous action spaces. Discretizing continuous action spaces would scale poorly, since the number of discrete actions grows exponentially with the dimensionality of the action. Furthermore, having a parametrized policy can be advantageous because it can generalize in the action space. Therefore with this thesis we study state-of-the-art deep reinforcement learning algorithm, *Deep Deterministic Policy Gradients*. We provide a theoretical comparison to other popular methods, an evaluation of its performance, identify its limitations and investigate future directions of research.

The remainder of the thesis is organized as follows. We start by introducing the field of interest, machine learning, focusing our attention of deep learning and reinforcement learning. We continue by describing in details the two main algorithms, core of this study, namely Deep Q-Network (DQN) and Deep Deterministic Policy Gradients (DDPG). We then provide implementatory details of DDPG and our test environment, followed by a description of benchmark test cases. Finally, we discuss the results of our evaluation, identifying limitations of the current approach and proposing future avenues of research.

2 Foundations

Machine learning is an approach to design and optimize information processing systems by directly using data. As typically in real-world problems the training data is limited and does not cover all possible scenarios, the system has to learn how to behave also in the presence of data which it has not been trained on. In this regard, overfitting is one of the biggest challenges in machine learning and consists in having a system that strictly adapts its behavior to the training data, without being able to generalize to different unseen input data.
Machine learning is usually divided into supervised learning, reinforcement learning and unsupervised learning. Supervised learning systems learn input-output mappings from a dataset of desired input-output pairs, i.e. they are explicitly told how to behave. As we will see in the next section, most of deep learning and neural network research so far has been done in the supervised setting. Reinforcement learning systems, on the contrary, learn how to behave by receiving feedback from the environment encoding a specific goal. For example, a trash-collector robot would receive a reward for collecting trash or would be punished for hitting a wall. It is common for reinforcement learning to exploit supervised learning techniques. Unsupervised learning, on the other hand, is about finding patterns in data and will not be discussed further in this thesis. In the next sections we will describe in more detail one of the most prominent supervised learning technique, namely deep learning and neural networks.

2.1 Deep Learning and Neural Networks

Deep learning is an area of machine learning concerned with deep neural networks. Neural networks are non-linear parametric functions loosely inspired by the human brain. They are composed of layers of units called *neurons*, as shown in Figure 2.1.

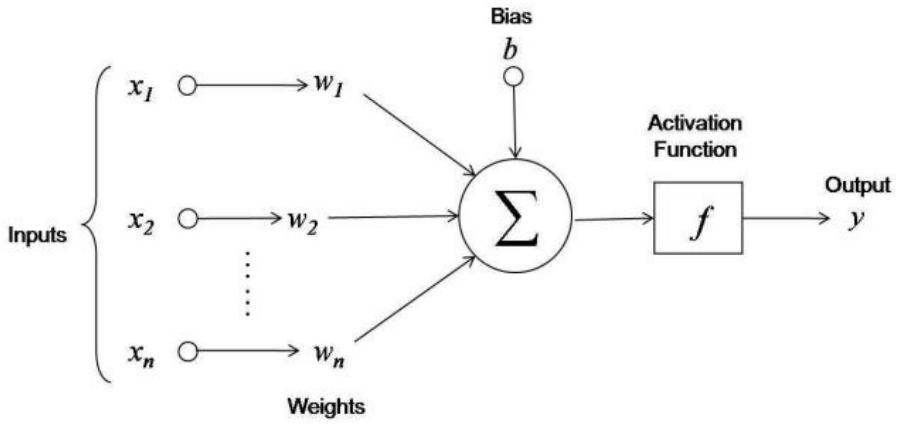
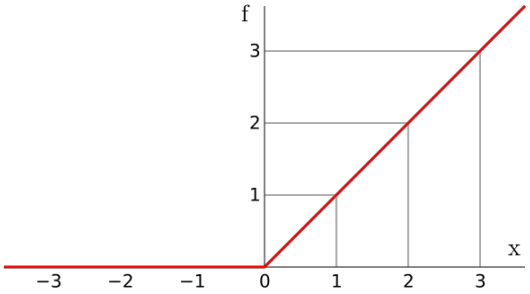
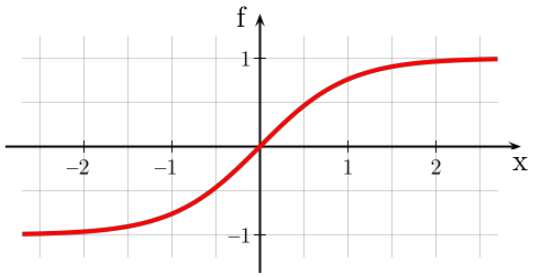


Figure 2.1: A canonical artificial neuron. The sum of the inputs x_i weighted by w_i and the bias b is fed into the activation function f, producing the neuron output $y = f(\sum_i w_i x_i + b)$. The weights w_i represent a pattern in the neurons input space. The closer the input is to that pattern, the higher the output of the neuron will be.

Multiple parallel neurons form a layer, each characterized by outputs called *activations* $y_j = f(\sum_i w_{i,j} x_i)$. Single layer (*shallow*) neural networks called *perceptron* have been first described by Rosenblatt in 1958 [4]. However in *deep* neural networks multiple layers are stacked on top of each other. In Krizhevsky et al. [5] achieved state-of-the-art computer vision results with an 8-layer, 500,000 neurons and 60-million parameter neural network. Since the outputs of a layer are non-linear features of the input, the more layers are in the network, the more non-linear and abstract those features are



(a) Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$.



(b) Hyperbolic Tangent (tanh): $f(x) = \tanh(x)$.

Figure 2.2: Examples of activation functions f. Usually, the function is monotonically increasing and saturates for low / negative inputs.

compared to the original input. This hierarchy of features enables neural networks to approximate complex functions.

Neural networks are usually initialized randomly and then trained on a dataset with regard to a *cost function*. Typically, cost functions are distance measures between the desired outputs (targets t_i) and the actual network outputs (y_i). A common and simple cost function is the mean squared error $\frac{1}{n}\sum_{i=1}^{n}(y_i - t_i)^2$. Training a neural network consists in optimizing the network parameters $\theta = (w_1, b_1, \cdots, w_n, b_n)$ to minimize the cost on the training dataset. However, training a neural network can be challenging. First, as the parameter space can be non-convex, neural networks are usually optimized with techniques guaranteeing convergence to a local optimum (e.g., gradient descent). Second, the number of parameters grows with the complexity of the network. As deep, complex networks are typically required to learn difficult functions, the training can be highly demanding, both in terms of computational time and data. Nevertheless, in practice neural networks can be optimized quite reliably by gradient descent, as we will see in the next section.

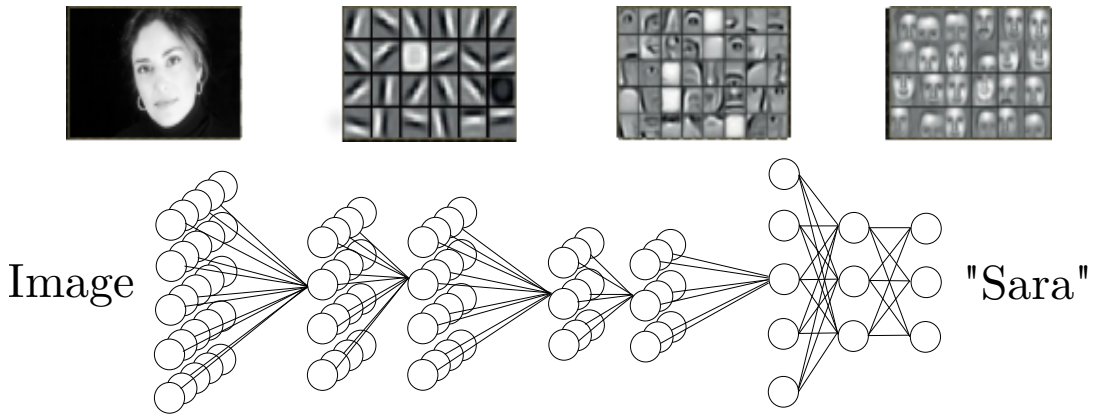


Figure 2.3: Multi-layer (deep) neural network. Neurons in higher (leftmost) layers represent more abstract features (top) of the network input.

2.1.1 Stochastic Gradient Descent

Using the gradient for optimization (i.e., first order optimization) is common to many machine learning algorithms with high number of parameters, since zeroth order methods (e.g., genetic algorithms) need several function evaluations and higher n-order methods are too expensive per evaluation. Therefore, gradient descent is a key element of deep learning. It consists in following the direction pointed by the gradient of the cost function with respect to the parameters of the system. In the case of neural networks, the gradient of the cost function with respect to the network parameters θ tells us how to

change the parameters in order to reduce the cost. More specifically, given the gradient of the cost with respect to the parameters $\frac{dC}{d\theta}$, we can update the parameters $\theta_{\text{new}} \leftarrow \theta_{\textbf{old}} - \alpha \frac{dC}{d\theta}$, where α is a *learning rate*. However, computing the cost and the gradient on large datasets is expensive. *Stochastic gradient descent* alleviates this issue by only computing the cost and the gradient for a small subset (*minibatch*) of the dataset. Like gradient descent, stochastic gradient descent is guaranteed to converge to a local minimum.

In high-dimensional optimization spaces like those of neural networks, optimization often gets stuck near saddle points or valleys where gradients are only high in directions orthogonal to the valley in which no progress can be made. This issue can be alleviated by having adaptive learning rates for each parameter. A recent version of stochastic gradient descent using these techniques and used in this thesis is ADAM [6]. Another interesting property of ADAM is that its stepsize is independent of the scale of the gradients.

To compute the gradient in a neural network, we can use *automatic differentiation*, a technique to compute derivatives in computational graphs in a modular way. It is based on the chain rule, which says that the derivative of a composition $y = f(g(x))$ of two functions $g : x \to w$ and $f : w \to y$ can be decomposed into $\frac{dy}{dx} = \frac{dy}{dw}\frac{dw}{dx} = g'(w) \cdot f'(x)$. In a computational graph, we can compute the derivative of any edge y with respect to any other edge x by first computing the derivative of the last node $g'(w)$ and then compute the derivative of all the previous nodes $f'(x)$ by decomposing them in the same way. Derivatives are then passed backwards through the computational graph. An example is depicted in Figure 2.4.

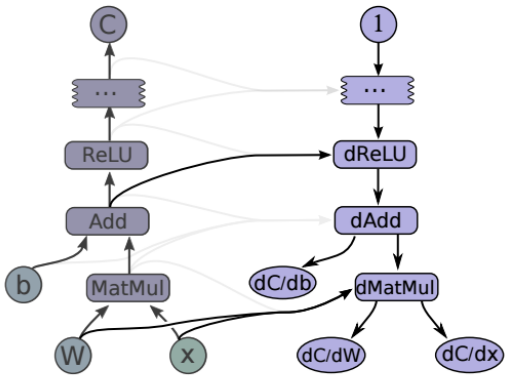
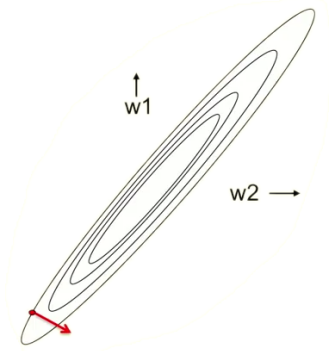


Figure 2.4: Learning iteration in a computational graph. First, the network output and cost C are computed (left). Subsequently, a backward pass is executed to compute derivatives (right).

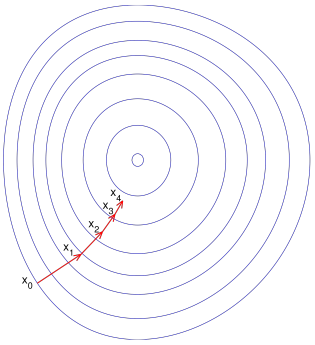
2.1.2 Normalization

Neural network training is highly sensitive to the mean and variance of the data. They affect the cost, gradients, activations and the operation region of the activation functions. This issue makes it hard to select good learning rates, parameter initializations and to set other hyperparameters. Also, scale differences between dimensions within a dataset result in skewing the cost surfaces (Figure 2.5a), thus resulting in a wrong direction of the gradients. Normalizing the data to zero mean and unit variance alleviates these issues (Figure 2.5b). Furthermore, normalization is not only important for the network inputs and targets but also for the activations inside the network. During training, each layer has to constantly adapt to its changing input distribution caused by the optimization of the previous layers. For alleviating this issue, Ioffe and Szegedy [7] proposed *batch normalization*, a techniques consisting in ensuring zero mean and unit variance for activations for each minibatch resulting in training time reduction by an order of magnitude.

In the next section we extend the discussion of deep learning techniques to reinforcement learning.



(a) Skewed cost surface due to unnormalized data. The gradient (red arrow) points in a direction almost orthogonal to the optimum.



(b) Normalized cost surface. The red arrows (gradients at each timesteps) correctly point towards the optimum.

Figure 2.5: Examples of contour plots of cost surfaces of one neuron with two weights.

2.2 Reinforcement Learning

Reinforcement learning is concerned with learning through interaction with an environment. At every timestep, a learner or decision-maker called *agent* executes an action and the environment in turn yields a new observation and a reward, as shown in Figure 2.6. The task of the agent is to maximize the sum of rewards received during the interaction with the environment.

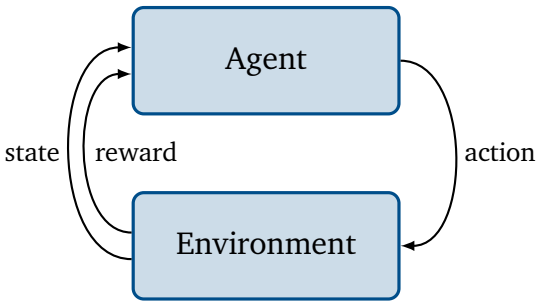


Figure 2.6: Agent-environment interaction in reinforcement learning. The agent can be anything interacting with an environment and able to improve its behavior (a human, an animal, a control system). Everything not learned is part of the environment (sensors, motors, rewards).

More formally, we can define a *state* $s \in S \subset \mathbb{R}^{d_s}$ as all the information the agent has about the environment at a given timestep. Generally, this information might not include the full state of the environment (e.g., in a card game the state would include the agent's hand but not all opponents hands even though they are part of the full state of the game).
An *action* $a \in A \subset \mathbb{R}^{d_a}$ encodes how the agents can interact with the environment. The mapping from states to actions is called *policy* $\pi : S \to A$. A policy can either be stochastic (e.g., a probability distribution of actions over states) or deterministic.
The *reward* $r \in \mathbb{R}$ is a feedback informing the agent about the immediate quality of its actions. Typically, the function generating the rewards is defined by an expert and can depend on the last state and action $r : S \times A \to \mathbb{R}$. The reward signal can encode the goal of the agent at different levels. For instance, in chess the agent can be rewarded at each capture or only at the end of the match.
The goal of the agent is to maximize the sum of the rewards received by the environment, namely the *return* $R = \sum_t \gamma^t r_t$. The discount factor $\gamma \in (0, 1]$ is to guarantee the convergence of the sum if the time horizon is not finite (*infinite horizon*). In the next section, we present a brief overview of classical approaches to solve reinforcement learning problems.

2.2.1 Learning Approaches

Reinforcement learning algorithms can be categorized as either *value-based*, *policy-based* or a combination of the two. Value-based methods consist in explicitly learning the value of all states and using it to select the action that leads to the highest-valued state. In this setting, the *value function* V is defined as the expected return of the agent being in state s_t and then following the policy π, while the *action-value function* Q is defined as the expected return of the agent being in state s_t, executing action a_t and then following the policy π. The *advantage function* A connects both.

$$\begin{aligned} V^{\pi}(s_t) &= \mathbb{E}_{r_{i\geq t}, s_{i>t}\sim E;\ a_{i\geq t}\sim\pi}\Big[\sum_{i=t}^{T}\gamma^{(i-t)}r(s_i,a_i)\Big], \\ Q^{\pi}(s_t,a_t) &= \mathbb{E}_{r_{i\geq t}, s_{i>t}\sim E;\ a_{i>t}\sim\pi}\Big[\sum_{i=t}^{T}\gamma^{(i-t)}r(s_i,a_i)\Big], \\ A^{\pi}(s_t,a_t) &= Q^{\pi}(s_t,a_t) - V^{\pi}(s_t). \end{aligned}$$

For deterministic policies, Q can be formulated recursively by the so-called *Bellman equation*

$$Q^{\pi}(s_t,a_t) = \mathbb{E}_{r_t, s_{t+1}\sim E}\left[r_t + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))\right].$$

Knowing $Q^{\pi^*}(s_t,a_t)$ for each state-action pair, the best policy π^* is to select the action with the highest action value at every timestep, i.e., $\pi^*(s_t) = \text{argmax}_a\ Q^{\pi^*}(s_t,a)$. In practice we neither know Q^{π^*} nor π^* in the first place. However, even when starting from a random Q, it is possible to iteratively update Q by exploiting the Bellman equation and to converge to Q^{π^*} (and therefore to π^*). This powerful approach is a key element in recent *Deep-Q-Networks* (DQN), which will be discussed further in Section 3.1.

In continuous action spaces, however, maximizing over Q is infeasible as there are infinite actions to consider. Even discretizing the action space becomes intractable for relatively low dimensional action spaces (*curse of dimensionality*). To overcome this issue, it might be better to directly learn a (parametrized) policy instead of a value function. Algorithms following this approach are called *policy-based*. One of the most successful class of policy-based algorithms is *policy gradient* [8, 9, 10]. Policy gradient approaches directly optimize the policy parameters ξ by following the direction of the gradient of the expected return with respect to the policy parameters ($\nabla_\xi R$), which can be directly estimated from samples.

A mixture of the value-based and policy-based approaches are *actor-critic* algorithms. These approaches make use of both a parametric policy (actor) and a value function estimator (critic), used to improve the policy. A very recent actor-critic approach for learning deterministic policies is *Deterministic Policy Gradients* (DPG) [11]. DPG uses a differentiable action-value function approximator to obtain the policy gradient by taking the derivative of its output with respect to the action input dQ/da. The policy gradient is then computed as $\nabla_\xi Q = \nabla_a Q \cdot \nabla_\xi \pi$. Deep Deterministic Policy Gradient (DDPG), a DPG with neural network function approximators, will be discussed further in Section 3.2.

2.2.2 Exploration and Exploitation

One of the biggest issues in reinforcement learning is the tradeoff between exploration and exploitation. Acting greedily (exploitation) with respect to an approximated function (e.g., Q-function) and choosing the current best action might prevent the agent from discovering new better states and therefore prevent improvement of the policy. On the contrary, excessive exploration might slow down the learning or even results in harmful policies. A tradeoff is therefore necessary. Usually, noise is added to the actions during training. In the case of discrete actions, the ϵ-greedy policy is a common solution: the agent acts randomly with probably ϵ and greedily with probability $1-\epsilon$. In the case of continuous actions, Gaussian noise could instead be added. In this thesis, we will use these simple strategies. However, the exploration-exploitation tradeoff is still an open problem in reinforcement learning. Alternative exploration strategies might consist in artificially rewarding exploration or even leaving the decision about exploration completely to the agent. We will come back to this topic in Section 6.2.

3 Deep Reinforcement Learning

As discussed in the previous section, reinforcement learning heavily relies on either approximating the Q-function (in the case of value-based algorithms) or on the policy parameterization (in the case of policy-based algorithms). In both cases, the use of rich function approximators or policies allows reinforcement learning to scale to complex problems. Neural networks have been successfully used as function approximators in supervised learning and are differentiable, a useful quality for many reinforcement learning algorithms. In this section, we focus on two recent reinforcement learning algorithms. The first one, *Deep Q-Network (DQN)* [1, 12], is a value-based algorithm successfully able to play 49 Atari games from pixels better than human experts. The second one, *Deep Deterministic Policy Gradients (DDPG)* [13], is an actor-critic algorithm, an extension to DQN for continuous actions.

3.1 Deep Q-Network (DQN)

DQN is a value-based algorithm using a neural network $Q(s,a|\theta)$ to approximate the optimal action-value Q^* of each action in a given state. The training targets for Q are computed via the Bellman equation $y_j = r_j + \gamma Q(s_j, \pi(s_j|\theta')|\theta')$. The network is trained to minimize the mean squared error with respect to the Q-function, i.e.,

$$\begin{aligned} C(\theta\,|\theta') &= \frac{1}{m}\sum_j \Big(\underbrace{r_j + \gamma\, Q\big(s_{j+1}, \pi(s_{j+1}|\theta')\,\big|\theta'\big)}_{y_j} - Q\big(s_j, \pi(s_j|\theta)\,\big|\theta\big)\Big)^2 \\ &= \frac{1}{m}\sum_j \Big(\underbrace{r_j + \gamma\, \max_a Q\big(s_{j+1}, a\,\big|\theta'\big)}_{y_j} - \max_a Q\big(s_j, a\,\big|\theta'\big)\Big)^2. \end{aligned}$$

However, the dependence of the Q targets on Q itself can lead to instabilities or even divergence during learning. Having a second set of network parameters $\theta' = \mathrm{LP}(\theta)$, where LP is a low-pass filter (e.g., exponential moving average), stabilizes the learning.

Additional instabilities can arise from training directly on the incoming states and rewards, since, unlike supervised learning, the input data (state-action pairs) is highly correlated as they are part of a trajectory. Furthermore, the policy and therefore data distribution might change quickly as Q evolves. DQN solves both issues by storing all (s_t, a_t, r_t, s_{t+1}) transitions in a *replay memory* dataset D and then learning on minibatches consisting of random transitions from D. This trick breaks the correlation of the input data and smoothes the changes in the input distribution. It also increases data efficiency by allowing the agent to perform multiple gradient descend steps on the same transition.

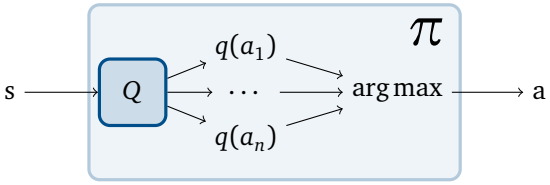


Figure 3.1: Architecture of DQN. The Q-Network outputs an estimate of the action-value function for each action. Subsequently, the policy π chooses the action with the highest value.

Finally, to ensure sufficient exploration, a simple ϵ-greedy policy, i.e., $\pi(s|\theta) = \arg\max_a Q(s,a|\theta)$. This off-policy approach is possible because the algorithm does not learn on full trajectories but only on isolated transitions. The complete algorithm is shown in Algorithm 1.

Algorithm 1: DQN

Initialize replay memory D
Initialize Q with random weights θ
Initialize target weights $\theta' \leftarrow \theta$
for $t = 1$ *to* T **do**
  Select action $a_t = \epsilon$-greedy $[\text{argmax}_a\, Q(s_t, a|\theta)]$
  Execute a_t and observe reward r_t and next state s_{t+1}
  Store transition (s_t, a_t, r_t, s_{t+1}) in D
  Sample random minibatch of m transitions from D
  Set targets $y_j = r_j + \gamma \max_a Q(a, s_{j+1}|\theta')$
  Perform gradient descent on cost $C = \frac{1}{m}\sum_j (y_j - Q(s_j, a_j|\theta))^2$ with respect to θ
  Update $\theta' \leftarrow \text{LP}(\theta)$
end

3.2 Deep Deterministic Policy Gradient (DDPG)

As discussed in the introduction, a parametrized policy is advantagous for control because it allows for learning in continuous actions spaces. Since DDPG is an actor-critic policy gradient algorithm, there is a policy network $\pi(s|\xi)$ with parameteters ξ in addition to the action-value network $Q(s, a|\theta)$ with parameters θ.
The training targets for Q are computed as in DQN, with the only difference that π now depends on ξ. Using the mean squared error, we derive the cost function for Q

$$C(\theta \mid \theta', \xi') = \frac{1}{m}\sum_j \Big(\underbrace{r_j + \gamma\, Q\big(s_{j+1}, \pi(s_{j+1}|\xi') \,\big|\theta'\big)}_{y_j} - Q\big(s_j, \pi(s_j|\xi) \,\big|\theta\big) \Big)^2.$$

As the targets depend on the explicit policy network, we also need the target policy parameters $\xi' = LP(\xi)$. Here, LP will be an exponential moving average with update rule $\xi' \leftarrow \tau\xi + (1-\tau)\xi'$ and $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$.

The policy is trained via policy gradient

$$\nabla_\xi\, Q\big(s_j, \pi(s_j|\xi) \,\big|\theta\big) = \nabla_a Q\big(s_j, \pi(s_j|\xi) \,\big|\theta\big) \cdot \nabla_\xi\, \pi(s_j|\xi),$$

that is, Q is the cost function for π

$$C(\xi \mid \theta) \;=\; -\,Q\big(s_j, \pi(s_j|\xi) \,\big|\theta\big).$$

As actions are continuous, correlated Gaussian noise $\mathcal{M}_t$ is added to the actions to ensure exploration. More specifically, $\mathcal{M}_{t+1} \leftarrow \vartheta \cdot \mathcal{M}_t + \mathcal{N}(0, \sigma))$, where $\mathcal{N}(0, \sigma)$ a normal distributed random variable and ϑ a hyperparameter controlling the frequency of the noise. Again, this off-policy approach is possible because the algorithm does not learn on trajectories but only on isolated transitions. Algorithm 2 shows the complete learning procedure.

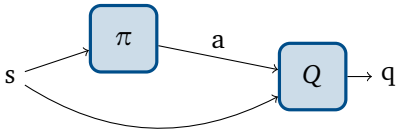


Figure 3.2: Architecture of DDPG. The policy π is trained by back-propagating the q-gradient with respect to the action a.

Algorithm 2: DDPG

Initialize replay memory D
Initialize π with random weights ξ and target weights $\xi' \leftarrow \xi$
Initialize Q with random weights θ and target weights $\theta' \leftarrow \theta$
for $t = 1$ *to* T **do**
    Select action $a_t = \pi(s_t) + \mathscr{M}_t$
    Execute a_t and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample random minibatch of m transitions from D
    Set Q targets $y_j = r_j + \gamma Q(s_{j+1}, \pi(s_{j+1}|\xi')\,|\theta')$
    Perform gradient descent on cost $C = \frac{1}{m}\sum_j (y_j - Q(s_j, a_j|\theta))^2$ with respect to θ
    Perform gradient ascent on $Q(s_j, \pi(s_j|\xi)\,|\theta)$ with respect to ξ
    Update $\theta' \leftarrow \mathrm{LP}(\theta)$
    Update $\xi' \leftarrow \mathrm{LP}(\xi)$
end

4 Implementation

A big challenge in deep reinforcement learning implementations are efficient neural network routines. The most critical part are the inner products of inputs and weights and the respective derivatives in each layer. We first coded in *MATLAB*, a machine learning popular tool providing efficient linear algebra routines. Due to the lack of automatic differentiation, we implemented gradient propagation routines, as well as the RMSProp and ADAM optimizers. The code can be found on the CD of this thesis.
However, due to the high computational demands of neural networks, we also developed a *TensorFlow* implementation. TensorFlow [14] is an open source Python / C++ library providing fast routines ($\sim$ 30 times faster than MATLAB from our experience) for deep learning, accomplished through GPU support. It features automatic differentiation and therefore allows for much more flexible network and algorithm design. Additionally, TensorFlow provides a wide variety of built-in optimizers (e.g., ADAM) and operations (e.g., batch normalization). The results shown in this theses have been produced by the TensorFlow implementation, which can be found on the CD of this thesis.
Another concern in setting up the testing framework regarded the ability to run several experiments on computing clusters with job schedulers while having a changing codebase. For this purpose, we wrote *ezex*, a small framework that provides basic operations (e.g., starting and aborting jobs) and a visualization of running and finished LSF (or SLURM) jobs via Jupyter Notebooks.

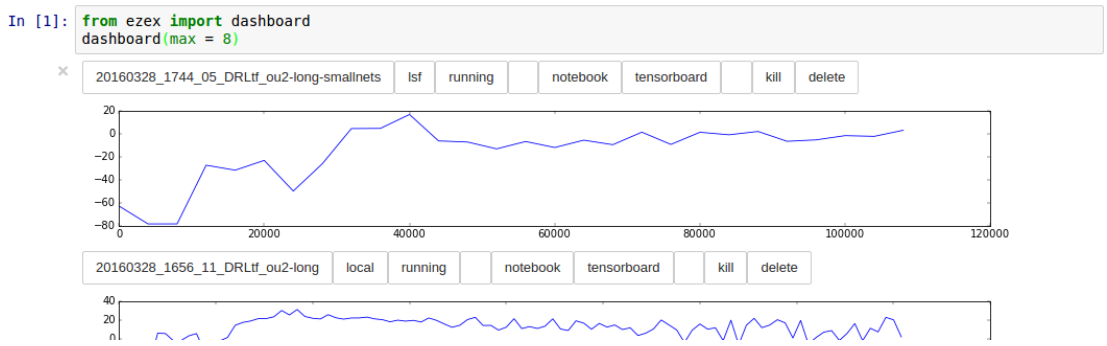
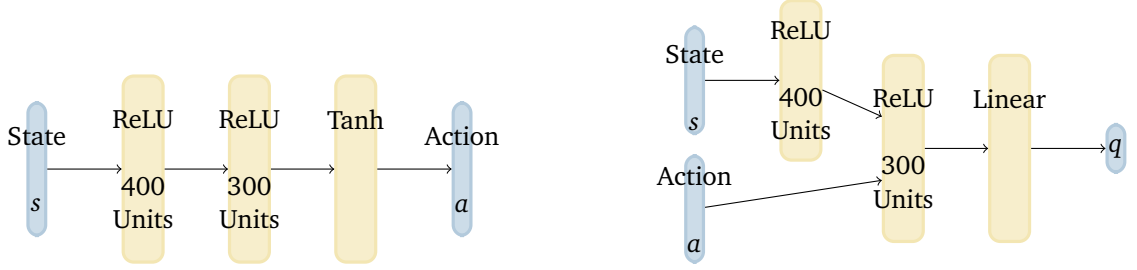


Figure 4.1: With ezex it is possible to visualize experiments located in a central folder on a (shared) file system. It also provides routines for aborting jobs and deleting experiments.

4.1 DDPG

Our DDPG implementation is as close as possible to the original paper. Its evaluation is performed on low-dimensional true state inputs and not on visual inputs due to time and software limitations. Figure 4.2 shows the layouts of the neural networks used, while Table 4.1 the hyperparameters used. The layers are initialized with small random weights.



(a) Layout of the policy network.

(b) Layout of the action-value network.

Figure 4.2: DDPG networks layout. Each network has about 130,000 parameters.

Hyperparameter	Value	Note
Minibatch size m	32	
Policy learning rate	10^{-4}	Step size for ADAM
Critic learning rate	10^{-3}	Step size for ADAM
Policy weight decay	0	
Critic weight decay	10^{-2}	L2 cost for each parameter
Target update rate τ	10^{-3}	
Replay memory size	500,000	Older transitions are discarded
Exploration noise σ	0.2	
Exploration noise reversion rate ϑ	0.15	The overall noise variance is 0.13
Reward discount γ	0.99	
Warm-up time	50,000	Timesteps until training starts

Table 4.1: Hyperparameters for DDPG.

4.2 The Cart-Pole Problem

Although the final goal is to apply deep reinforcement learning in real-world settings (e.g., robotics), testing in these environments can be dangerous and expensive both in terms of money and human expert labor. Simulated environments are well suited for testing reinforcement learning systems extensively as the testing process is cheap and can be easily automated. For these reasons, both DQN and DDPG have been originally evaluated in simulated environments. In the original works DQN was tested on several Atari games using the simulator ALE [15], while DDPG has been evaluated in more than 20 physical environments using the MuJoCo [16] simulator. The authors showed that the algorithms and the same hyperparameters work on a wide variety of tasks without the need of hand-tuning them for each individual task. In our implementation, we also used MuJoCo but, because of time constraints, we evaluated DDPG only on the cart-pole (Figure 4.3), a standard reinforcement learning benchmark problem. The task is a standard non-linear control task. The goal is to indirectly swing-up and balance a pole placed on a cart by a applying control force to the cart. Environment details (e.g., masses, friction coefficients, etc.) can be found in the MuJoCo model file `cartpole.xml`.

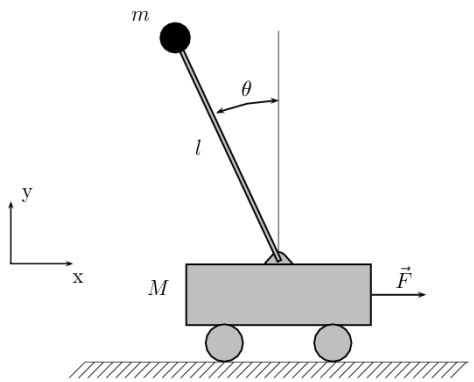


Figure 4.3: The cart-pole environment. The state consists of four dimensions: the vertical cart position $s_0 \in [-1, 1]$, the angle of the pole $s_1 \in [-\pi, \pi)$ and their derivatives $s_2 = \dot{s_0}$ and $s_3 = \dot{s_1}$. In our case the agent is able to fully observe the state.

5 Evaluation

Even though DDPG turned out to be highly sensitive to the reward functions, they have not been included in the original paper. Among the several reward functions we tried, the most reliable one is

$$r = 0.5 \cdot \cos(s_1) \; - 0.03 \cdot a^2 \; - 0.015 \cdot |s_0| \; - 0.2 \cdot s_3^2.$$

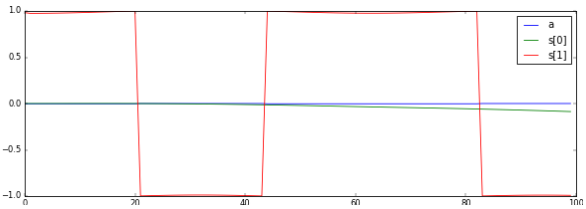
Here, $\cos(s_1)$ has the highest influence, giving -1 when the pole is hanging down and 1 when it is standing up. The additional terms consist in penalties for the action a, the cart position s_0 (for not being in the middle) and the rotation speed of the pole s_3. However, we noticed that even slight changes to the parameters of the reward functions impaired the learning dramatically.

Below, we report results on varying the reward function and the hyperparameters. In particular, we evaluate the effects of batch normalization, experience replay and the use of a target network. All experiments have been averaged over ten trials.

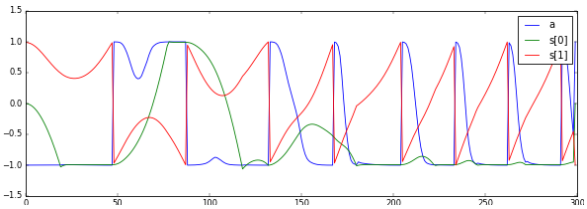
5.1 Reference Results Without Batch Normalization

Here, we report the best results we achieved. The reward function used it the handcrafted one described above and we did *not* use batch normalization. With this settings, the agent was consistently able to learn the optimal swing-up and balance policy. The returns and some sample trajectories over the course of the learning process are depicted in Figure 5.1 and Figure 5.2, respectively.

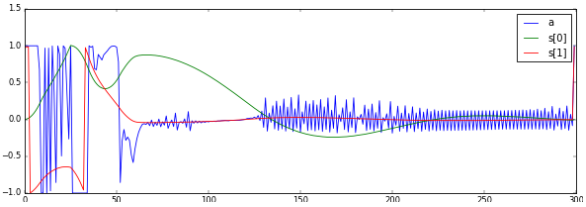
We stress that, even though we used constant episode lengths, shifting the rewards (i.e., adding a constant) had an impact on the learning stability. At first we added a bias to ensure the returns for the initial policy were zero to match the initial Q values, but having a negative bias (introduced by $\cos(s_1)$) improved the learning consistency significantly.



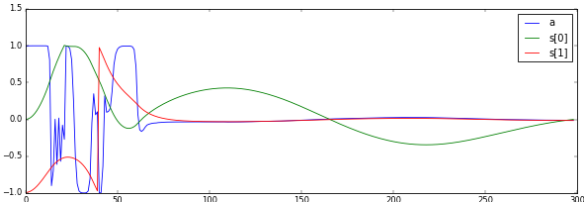
(a) Trajectory after 0 timesteps of learning.



(b) Swing-up after $\sim 10,000$ timesteps.



(c) Balance after $\sim 100,000$ timesteps.



(d) Near optimal after $\sim 200,000$ timesteps.

Figure 5.1: Cart pole test trajectories, showing the action a (blue), the cart position s_0 (green) and the normalized pole angle s_1 (red). An angle of -1 or 1 indicates that the pole is hanging down, while 0 that it is standing up.

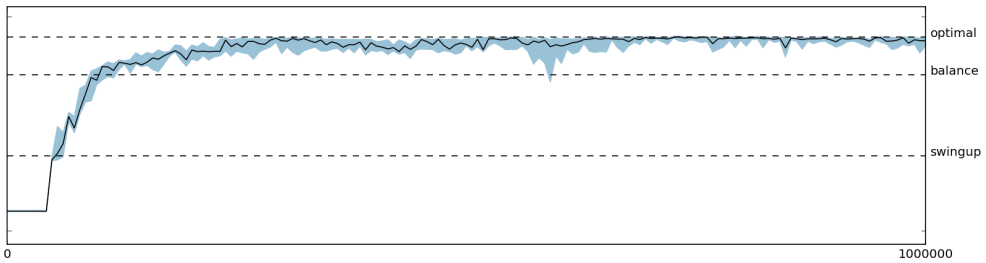


Figure 5.2: Expected return with the handcrafted reward function and without batch normalization. The black line denotes the mean, while the blue area the interquartile range. During the warm-up time the agent is acting according to the initial policy (i.e., randomly) and is only filling the replay memory, therefore the expected return does not improve.

5.2 Applying Batch Normalization

We then evaluated the same reward function *with* batch normalization. Surprisingly, we observed much worse results, as shown in Figure 5.3. Even though we tried the canonical as well as the non-standard batch normalization used in the DDPG paper, the agent was never able to balance after swinging up. This behavior might be due to the cart-pole environment, as the original paper also reported slightly inferior results for batch normalization on cart-pole. It could also be due to the fact that batch normalization introduces additional noise during learning. A very recent method claiming to alleviate this issue and that could be interesting for future research is *weight normalization* [17].

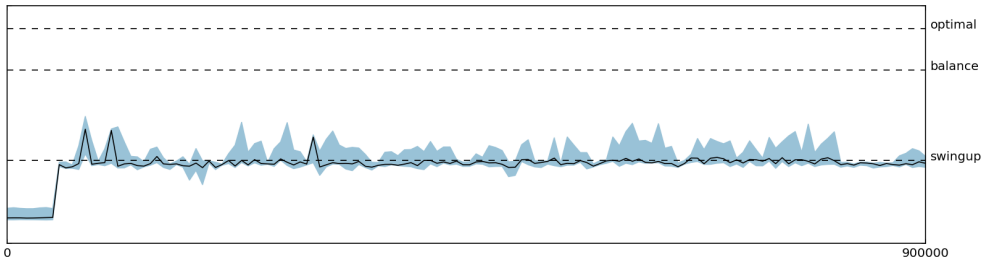


Figure 5.3: Expected return with batch normalization. The agent only learns to swing up and fails at balancing.

5.3 Disabling Target Network and Replay Memory

Consistent with the findings from the original paper, we observed reduced performance when disabling the target networks (Figure 5.4) or the replay memory (Figure 5.5).

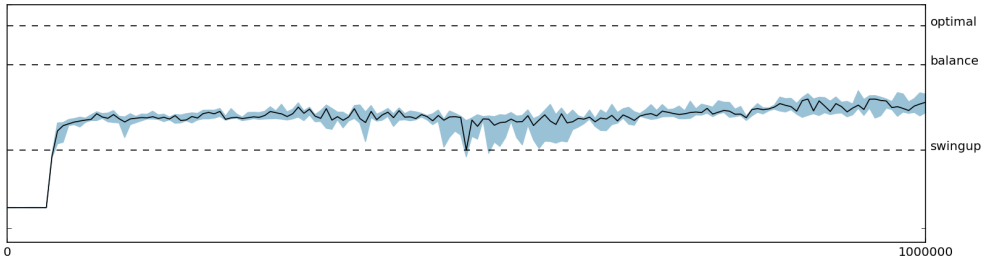


Figure 5.4: Expected return without target networks (i.e., setting the target update rate $\tau = 1$).

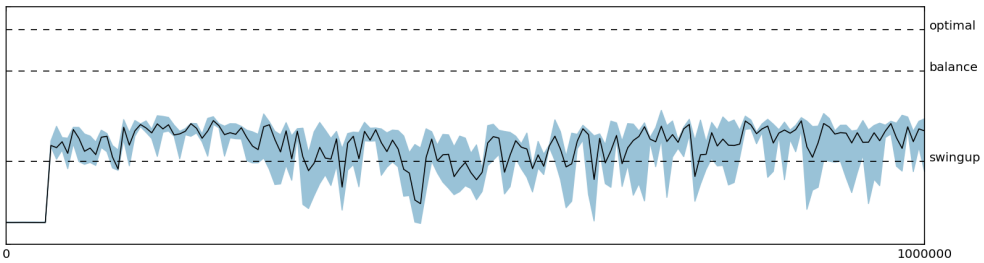


Figure 5.5: Expected return without experience replay (i.e., the neural network is trained on the last collected 32 transitions).

5.4 Sparse Reward Functions

As already discussed, we found DDPG to be very sensitive to the reward function. Initially, we were able to learn only with smooth reward functions, while using a sparse reward function encoding only the actual goal (i.e., rewarding the agent only when the pole is standing upright $r = (|s_1| < 0.01)$), there was no improvement over the initial policy at all, as shown in Figure 5.6. At first, we thought that this behavior might be due to poor exploration of the state space and that the agent, never reaching the goal state, does not know where to get positive rewards. While this might in fact be an issue in more complicated environments, it was not the issue in the cart-pole. Instead, we found that increasing the magnitude of the reward and increasing the warm-up time helped the agent to learn good action value function and policy, as shown in Figures 5.7, 5.8 and 5.9. In general, extending the warm-up time stabilized learning tremendously. This behavior might be due to the fact that transitions from initially diverging policies have a lower chance to get sampled from the replay memory since the replay memory is already filled with many transitions from the warm-up time.

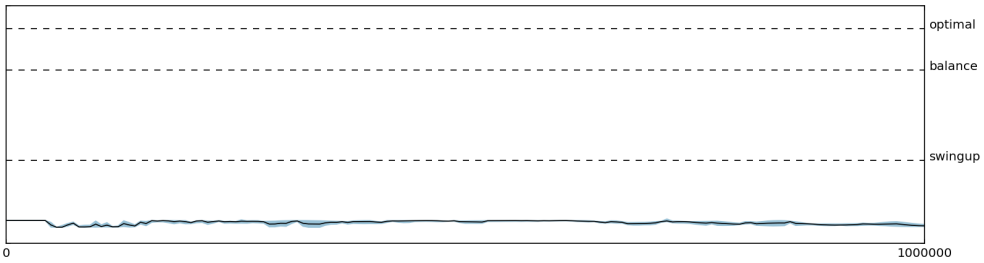


Figure 5.6: Expected return with $r = (|s_1| < 0.01) - 0.03 \cdot a^2$ and $t_{warump} = 10,000$. With a sparse reward function and a short warm-up time, there is no learning at all.

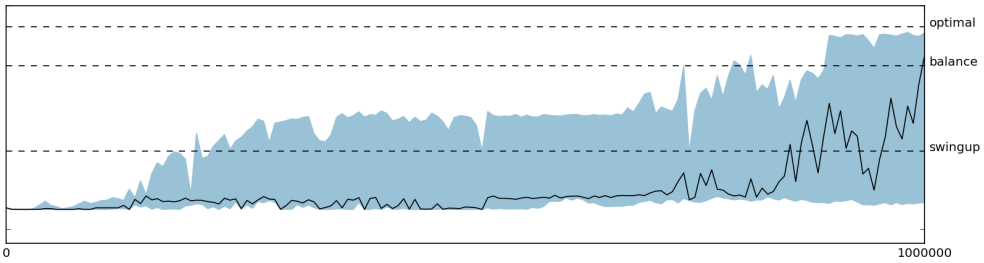


Figure 5.7: Expected return with $r = \mathbf{4} \cdot (|s_1| < 0.01) - 0.03 \cdot a^2$ and $t_{warump} = 10,000$. Increasing the magnitude of the positive reward helped the agent, although the learned policy is far from the optimal one.

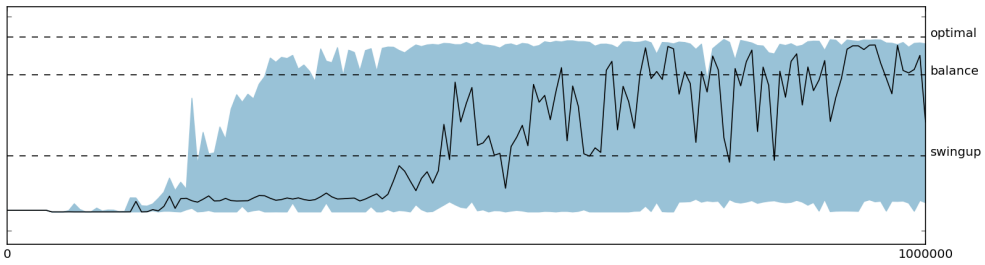


Figure 5.8: Expected return with $r = \mathbf{4} \cdot (|s_1| < 0.01) - 0.03 \cdot a^2$ and $t_{warump} = \mathbf{50,000}$. Increasing the warm-up time further increased the performance.

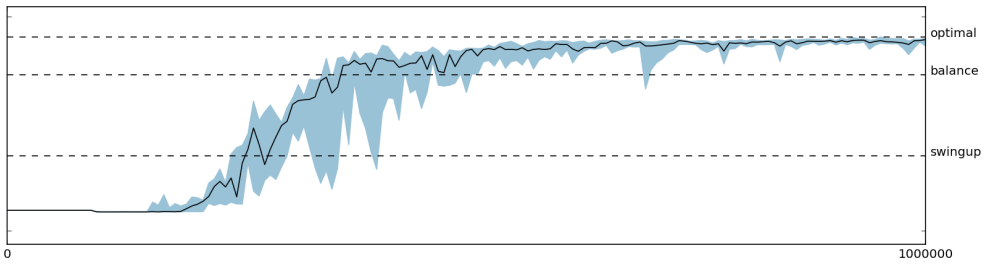


Figure 5.9: Expected return with $r = \mathbf{4} \cdot (|s_1| < 0.01) - 0.03 \cdot a^2$ and $t_{warump} = \mathbf{100,000}$. With both a high-magnitude positive reward function and a sufficiently large warm-up time, the agent is able to always learn a near optimal policy.

6 Conclusion and Future Work

With this thesis, we have provided an analysis of state-of-the-art the deep reinforcement learning algorithm DDPG. The results from the original papers have been reproduced and we successfully trained an agent for solving the cart-pole balance and swing-up tasks with continuous actions. We provided an analysis of the performance for sparse reward functions and evaluated the use of batch normalization, target network and replay memory.

We showed that DDPG was able to learn a non-trivial control policy, but there are several limitations that need to be solved before it will be possible to apply it to robotics. Below, we identify the major weaknesses and propose future avenues of research.

6.1 Data Efficiency

The first significant issue of deep reinforcement learning is data efficiency. In our test cases we experienced that in order to learn high-quality policies, the algorithm had to be fed with a high number of samples. Considering that real control problems are much harder that the simple task we evaluated, DDPG is not ready for real world problems.

There are, however, several papers improving DQN and some of the techniques proposed can be applied to DDPG with slight modifications. *Prioritized experience replay* [18] is for instance a technique for improving data efficiency by not sampling transitions uniformly from the replay memory but instead sampling according to the importance of a transition. Using the temporal difference error as a measure of importance, the authors achieved new state-of-the-art results on the DQN Atari benchmark games.

An alternative way to reduce the number of parameters (and therefore the number of required samples) is to move the first layers of the both the policy and the Q networks to a common *preprocessing network* ϕ, as shown in Figure 6.1. Since neurons in the first layers have to be useful to many other neurons in the following layers, they tend to learn task independent features anyway (e.g., for visual inputs these are typically edge filters as in Figure 2.3) and could therefore combined in a single network. Such a network would then be trained by the gradients from both π and Q. Having ϕ would also allow us to introduce additional function approximators for the value function $V(s)$ or to learn model $s_{t+1} = M(s_t, a_t)$ without much parameters overhead. Furthermore, learning a model and a value function could serve as a good regularizer for ϕ since transition data (model) and value data are currently unused in the DDPG framework.

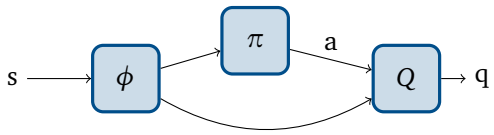


Figure 6.1: DDPG with a preprocessing network ϕ.

Moreover the model and the value function can be used to estimate another policy gradient, since the expected return can be also expressed via $\mathbb{E}[R] = r(s_t, \pi(s_t)) + \gamma\ V(M(s_t, \pi(s_t)))$. This approach is called *value gradient*. Because it assumes a deterministic model, it is only applicable in deterministic environments. However, there have been recent advances in expressing stochasticity in neural networks [19]. Building on this idea, Heess et al. [20] presented *Stochastic Value Gradients*, extending the value gradient approach to stochastic neural network models and policies. Even though these approaches have not been covered in this thesis, they seem a promising direction for further research.

6.2 Exploration

Another possible improvement to DDPG regards more efficient exploration. More specifically, stochastic policies that are able to encode multiple good actions instead of relying on ϵ-greedy exploration can speed up learning exponentially as Osband et al. [21] showed for DQN. As already mentioned in the introduction, rewarding exploration can be a way to enable the agent to actively explore. By learning a transition model for DQN and using the model error as a proxy for exploration (i.e., rewarding according to the model error) Stadie et al. [22] were able to improve performance especially where the original DQN performed poorly.

6.3 Imitation

In robotics, a simpler way to adopt complex behavior or to bootstrap the learning is imitation, consisting in using target trajectories for supervised learning of policies. In the case of DDPG, imitation could be employed by either initializing the networks by supervised training or by injecting target trajectories into the replay memory. For example, recently Silver et al. [2] were tremendously successful in the game of Go by first pre-training a policy network on human expert moves and subsequently improving it via reinforcement learning.
With regard to robotics, another helpful approach to avoid dangerous policies would be to initialize the Q-network with negative values for specific trajectories (e.g., for the violation of the joint limits).

6.4 Curriculum Learning

Curriculum learning [23] is a deep learning approach to learn complex functions by training on easier and more fundamental examples first and only gradually introducing more complicated examples. This could be also applied to reinforcment learning and especially robotics. For a robot the most fundamental concepts to learn are its own body dynamics and the skill to move without hurting itself. For example in table tennis, a robot could first learn to explore itself carefully (e.g., giving it rewards for exploration or punishing violation of joint limits) and subsequently learn striking movements and to win a match.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," in *Experimental Robotics IX*, pp. 363–372, Springer, 2006.

[4] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[6] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[7] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[8] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[9] J. Baxter and P. L. Bartlett, "Direct gradient-based reinforcement learning," in *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, vol. 3, pp. 271–274, IEEE, 2000.

[10] S.-I. Amari, "Natural gradient works efficiently in learning," *Neural computation*, vol. 10, no. 2, pp. 251–276, 1998.

[11] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[13] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[14] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous systems," *Software available from tensorflow.org*, 2015.

[15] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 06 2013.

[16] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 5026–5033, IEEE, 2012.

[17] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *arXiv preprint arXiv:1602.07868*, 2016.

[18] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[19] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[20] N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa, "Learning continuous control policies by stochastic value gradients," in *Advances in Neural Information Processing Systems*, pp. 2926–2934, 2015.

[21] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, "Deep exploration via bootstrapped dqn," *arXiv preprint arXiv:1602.04621*, 2016.

[22] B. C. Stadie, S. Levine, and P. Abbeel, "Incentivizing exploration in reinforcement learning with deep predictive models," *arXiv preprint arXiv:1507.00814*, 2015.

[23] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, ACM, 2009.