Deep Reinforcement Learning

Contents

Introduction

Machines can learn to play Atari games [1], and have beaten the world champion in Go [2]. They use the methods of *deep reinforcement learning*, which combine techniques from *reinforcement learning* and *deep learning*.

Reinforcement learning is about agents that carry out a task by controlling their environment. They learn strategies to solve the task solely by interacting with that environment. Typically, a certain strategy is adopted after repeatedly leading to a positive outcome – hence the name 'reinfrocement learning'. Deep learning, on the other hand, refers to the parts of statistical learning that involve deep neural networks.

These fields merged into deep reinforcement learning in 2013 [3]: agents that process their observations of the environment with a deep neural network initially achieved a major breakthrough in the Atari domain, and soon also dominated many other control tasks and games.

Chapter 1 introduces the mathematical framework of reinforcement learning and the model that underlies deep learning.

Chapter 2 provides a derivation of the breakthrough algorithm of 2013, which belongs to the class of value based algorithms. This derivation proceeds in two steps: We first derive a conventional reinforcement learning algorithm, and then turn it into a deep reinforcement learning algorithm.

Chapter 3 explores policy gradient methods, which belong to the class of policy based algorithms.

Finally, chapter 4 describes three original experiments, in which theoretical results from chapters 2 and 3 are applied in simple scenarios. The Github repository [4] contains the respective source code.

1 Framework and Model

The first part of this section describes the framework of reinforcement learning: a discrete time stochastic dynamical system models the environment, the agent interacts with it by sequentially choosing actions.

The second part of this section provides both a mathematical introduction and a heuristic discussion of deep neural networks.

1.1 The Framework of Reinforcement Learning

Consider an environment that evolves in time. At each time, the state of the environment is fully described by some element s of the *state space* $\mathcal{S}$. In general, $\mathcal{S}$ might be a finite or countably infinite set, or a compact subset of finite dimensional euclidean space. In this essay, we focus on countable $\mathcal{S}$; the lecture notes [5] elaborate on the continuous case. $\mathcal{P}(\mathcal{S})$ is the set of probability measures on $\mathcal{S}$.

The agent controls the dynamics of the environment by sequentially choosing actions from the *action space* $\mathcal{A}$, a space similar to $\mathcal{S}$. Together with $\mathcal{S}$ and $\mathcal{A}$, the transition probability function

$$P : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$$

defines a *discrete time stochastic controllable dynamical system* [5] .

The dynamics of the system are labeled by the discrete time $t \in \mathbb{N}_0$. At time t, taking the action a_t changes the state of the environment from s_t to s_{t+1} with probability $P(s_{t+1} \mid s_t, a_t)$. In this formalism, knowledge of the precise mechanics of the environment is not mandatory to frame it for reinforcement learning. The stochastic component of the framework absorbs uncertainties and insufficient knowledge.

We now leave the choice of actions to an autonomous agent, which thereby gains control over the environment. Explicitly, the agent provides a stochastic policy: a *stochastic policy* π is a map

$$\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A}),$$

where $\mathcal{P}(\mathcal{A})$ denotes the set of probability measures on $\mathcal{A}$. π encodes stochastic behaviour: after observing the current state of the environment s, the agent chooses an action by sampling from the distribution $\pi(\cdot \mid s)$. The agent's influence on the environment is complete encoded in the policy: as soon as π is fixed, the transition probability to the next state only depends on the current state. Specifying π turns the controllable dynamical system into a Markov model.

An autonomous agent that sequentially interacts with the environment gives rise to a stochastic process [6]. Let $(\mathcal{S}, \mathcal{A}, P)$ define a discrete time stochastic controllable dynamical system, let π be a corresponding stochastic policy, and consider T time steps. A sequence

$$\omega = (s_0, a_0, \ldots, s_{T-1}, a_{T-1}, s_T)$$

of successive states and actions is a *trajectory* or a *sample path*. The sample space Ω is the set of all trajectories. $S_t : \Omega \to \mathcal{S}$ and $A_t : \Omega \to \mathcal{A}$ are random variables satisfying

$$S_t(\omega) = s_t \qquad\qquad A_t(\omega) = a_t \tag{1.1}$$

Given a distribution for the initial state $\mu(s)$, the transition probability function P and the policy π together induce a probability distribution on Ω:

$$P^{\pi}_{\mu,T}(\omega) = \mu(s_0)\, \pi(a_0 \mid s_0)\, P(s_1 \mid s_0, a_0) \ldots \pi(a_{T-1} \mid s_{T-1})\, P(s_T \mid s_{T-1}, a_{T-1}) \tag{1.2}$$

$P^{\pi}_{\mu,T}(\omega)$ is the probability that the trajectory ω is realised by an agent that follows the policy π for T steps, starting from a state sampled from $\mu(\cdot)$. The expression mirrors how the trajectory is generated: at each time step, the agent observes the current state of the environment and chooses an action according to its policy. The environment responds to the action with a transition governed by its stochastic dynamics.

So far, the agent just blindly executes his policy, and has no means to improve it. To learn to carry out a task by itself, it needs feedback to gauge its performance. This feedback defines the task and is implemented as a *reward function*. The reward function is a map

$$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$$

The agent receives the numerical value $r(s, a)$ for taking the action a after observing the environment in state s. The received reward indicates how good the most recent decision was; it measures the agent's performance during a single time step.

The experiments 4.1 and 4.2 illustrate how to formalise environments in the framework of reinforcement learning, and how to specify tasks by defining a reward function.

An episodic task is a task with a fixed number $T \in \mathbb{N}$ of time steps. T is called the length of an episode, and must be specified in addition to the reward function to define an

episodic task. This essay considers episodic tasks only.

To evaluate its own behaviour, the agent should take the rewards of all time steps into account. The state-value function formalises this idea; it provides a performance measure for episodic tasks in deep reinforcement learning: Let $(\mathcal{S}, \mathcal{A}, P)$ define a discrete time stochastic controllable dynamical system, and let π be a corresponding stochastic policy. Let $T \in \mathbb{N}$ be the length of an episode, and r a suitable reward function. The *state-value function* or *expected return* v_T^π is a map $v_T^\pi : \mathcal{S} \to \mathbb{R}$ defined by

$$v_T^\pi(s) = \mathbb{E}_{P_{\mu,T}^\pi}\left[\sum_{t=0}^{T-1} r(S_t, A_t) \,\middle|\, S_0 = s\right] \tag{1.3}$$

$\mathbb{E}_{P_{\mu,T}^\pi}$ denotes an expectation with respect to the probability distribution in eq. 1.2 over all trajectories starting from the initial state s. An agent that controls the environment for T time steps according to the policy π, starting from state s, can expect to receive a sum of rewards equal to $v_T^\pi(s)$.

v_T^π contains the rewards for all decisions the agent made during an episode, and is thus a measure for its overall performance. More concretely, the state-value function allows to compare policies: for a fixed intial state s,

$$v_T^{\pi_1}(s) \geq v_T^{\pi_2}(s)$$

indicates that on average, the decisions derived from π_1 lead to a larger sum of rewards than those derived from π_2. Consequently, a policy $\pi^\star$ that satisfies

$$v_T^{\pi^\star}(s) \geq v_T^\pi(s) \,\forall \text{ policies } \pi,\ s \in \mathcal{S} \tag{1.4}$$

deserves to be called an *optimal policy*. All reinforcement learning methods aim at finding this optimal policy.

The optimality criterion defined in eq. 1.4 completes the framework of reinforcement learning: A discrete time stochastic controllable dynamical system formalises an agent observing and controlling its environment; a stochastic policy encodes the agent's behaviour. A reward function and the length of an episode together implement a task, and induce a notion of optimal behaviour.

In environments with large, complicated state spaces, optimal behaviour necessarily includes the extraction of relevant features from the description of the states. This essay focusses on agents that extract these features and relate them to a choice of action using multilayered, nonlinear, smoothly parametrised models called deep neural networks. The next section provides a detailed description of these models.

1.2 Deep Neural Networks

Neural networks are a specific class of *function approximators*, inspired by networks of neuron cells in the brain. This section starts with introducing function approximators, and continues with a rigorous mathematical description of neural networks. It concludes with a short, rather intuitive discussion.

Consider a function $f : X \times \mathbb{R}^n \to Y$ and the family of functions

$$\{f_\theta\} := \{f(\cdot, \theta) : \theta \in \mathbb{R}^n\},$$

which we call a parametrised function approximator. Now, assume that a function $F : X \to Y$ relates the two quantities X and Y. Consider the problem of modelling F. Further, assume that finding an analytic expression for F is hard, while providing samples of the form $(x, F(x))$ is easy.

The function approximator $\{f_\theta\}$ then serves as a general parametrised ansatz. Fitting this ansatz to a set of samples via the parameters θ yields an approximation of F. This process is called *training* [7].

Reinforcement learning algorithms often employ function approximators to extract relevant information from the agent's observations of the current state. These observations might have a complicated, highly redundant structure - for example, they could be images. Yet, typically, only certain aspects of an observation are relevant to successfully control the environment. Such aspects might include relative distances of objects depicted in an image. Function approximators can model the relation between raw observations and relevant key figures. In competitions like ILSVRC, neural networks prove to be the best known models for such tasks [8].

The functions f_θ that belong to a neural network consist of a sequence of affine linear transformations called layers, each directly followed by a nonlinearity. A single layer $L : \mathbb{R}^k \to \mathbb{R}^l$ encompasses a *weight matrix* $W \in \mathbb{R}^{k \times l}$ and a *bias vector* $b \in \mathbb{R}^l$:

$$L(x) = Wx + b$$

Before entering the next layer, each output component of the previous layer is passed trough a nonlinear function $\sigma : \mathbb{R} \to \mathbb{R}$:

$$(x')^a = \sigma(L^a(x))$$

The upper indices index the components of vectors, while lower indices enumerate the layers. Let $L_1, L_2, \ldots, L_n$ be layers:

$$\begin{array}{rccll} L_i : & \mathbb{R}^{k_{i-1}} & \to & \mathbb{R}^{k_i} & \\ & x & \mapsto & L_i(x) = W_i x + b_i & \forall i = 1, \ldots n \end{array} \tag{1.5}$$

Further let $\sigma_1, \ldots, \sigma_{n-1}$ be nonlinear functions. For $\theta = (W_1, b_1, \ldots, W_n, b_n)$, define the function f_θ by

$$\begin{array}{rccc} f_\theta : & \mathbb{R}^{k_0} & \to & \mathbb{R}^{k_n} \\ & x_0 & \mapsto & f_\theta(x_0) \\ & f_\theta(x_0) & = & L_n(x_{n-1}) \\ & (x_k)^a & = & \sigma_k\left(L_k^a(x_{k-1})\right) \quad \forall\, k = 1, \dots n-1 \end{array} \tag{1.6}$$

In words, the initial input x_0 undergoes n stages of processing; it *propagates forward* through the network. At each stage k, the *signal* x_{k-1} is transformed by the layer L_k before passing through the nonlinear function σ_k componentwise. Only the last stage spares the nonlinearity.

The family of functions $\{f_\theta\}$ constitutes a neural network with n layers, smoothly parametrised by the weights and biases that form its layers. All members of the family share the same number of layers, and the same choice of nonlinear functions. These specifications are called the *architecture* of the neural network. During training, the architecture of a neural network stays fixed, while the parameters change. The designation 'deep' is not a precise specification; typically, networks with more than four layers are called deep.

Several different types of layers and nonlinear functions occur in practice [9], some examples follow. The basic layer type is the *fully connected* layer:

$$L^a_{\mathrm{FC}}(x) = \sum_{b=0}^{l} W^{ab} x^b$$

Individual components of the weight matrix connect every output component with all input components.

A *convolutional* layer $L_{\mathrm{Conv}} : \mathbb{R}^{sl} \to \mathbb{R}^l$ convolutes a kernel W_{conv} with different parts of its input. Typically, it convolutes the kernel with blocks of some size n. Different output components arise as the kernel is shifted with respect to the input vector by a constant increment called *stride* s:

$$L^a_{\mathrm{Conv}}(x) = \sum_{b=as}^{as+n} (W_{\mathrm{conv}})^{ab}\, x^b$$

The architecture specifies the stride s and the size of the blocks n, while the components of the kernel W_{conv} serve as parameters.

Some choices for the nonlinear functions σ are $\sigma(x) = \tanh(x)$ or $\sigma(x) = 1/(1+\exp(-x))$. These functions are used to model the activation of neuron cells in the brain, and are therefore often called *activation functions*. Another popular choice of nonlinearity is the *rectifier linear unit* (ReLu) nonlinearity

$$\sigma(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Figure 1.1 depicts the architecture of the neural network featured by examples 2.3 and 3.4.

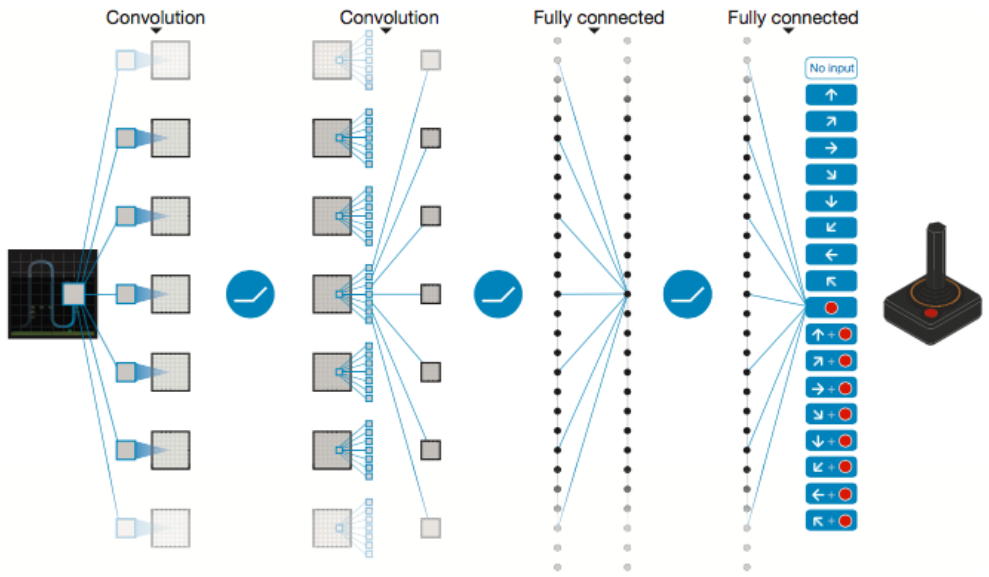

Figure 1.1 This diagramm, taken from the publication [1], shows the architecture of the first deep Q network, cf. chapter 2. It contains both convolutional and fully connected layers, joined by ReLu nonlinearities. The first convolutional layer features 16 independent kernels of size $8 \times 8 \times 4$ and uses stride 4, the second layer features 32 kernels of size $8 \times 8 \times 16$ and uses stride 2. The output of the second layer is thus a signal of dimension $9 \times 9 \times 32$, which is then reduced to 256 and finally to 4 to 18 dimensions by the remaining fully connected layers. The network contains roughly 7×10^5 parameters.

The above definitions specify what neural networks are, but do not explain why they work so well. One remarkable property of neural networks is *universality*: In 1989, G. Cybenko showed that a neural network can approximate any function $f : \mathbb{R}^n \to \mathbb{R}$ to arbitrary precision [10]. However, an excessive number of parameters is necessary to approximate a generic function. Furthermore, many less successful function approximators share that property.

Generally, a function approximator should represent the functions of interest efficiently, that is, incorporating only few parameters. To achieve this, the structure of the approximator must somehow match the structure of the represented functions - and thus, the structure of the data generated from those functions. Now, neural networks have very successfully been used to represent the relation of natural images to their content [8]. So apparently, the structure of deep neural networks somehow resembles the structure of natural images, or natural data in general. A detailed, widely accepted explanation for this hypothesis does not exist so far; it is the subject of current research [11].

2 From Q-learning to Deep Q-Learning

The *Deep Q Network* (DQN) algorithm, a method to train a reinforcement learning agent equipped with a deep neural network, was first published in 2013 [3]. Even though deep neural networks had been used in reinforcement learning before [12], the DQN algorithm was the first training method that enabled a single agent to learn successful strategies for many different tasks.

Deep Q-learning draws inspiration from the conceptually simple and well known Q-learning algorithm. Q-learning is a conventional reinforcement learning algorithm; its derivation, which follows [13], fills the first half of this chapter. In the second half, we will see how Q-learning can be modified in order to train an agent that employs a deep neural network.

2.1 Q-learning

The Q-learning algorithm approximates an auxiliary function called the *state-action value function* $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Given a policy π for an episodic reinforcement learning task with T time steps, the value Q^π of a state-action pair (s, a) is defined by

$$Q_T^\pi(s,a) = \mathbb{E}_{P_{\mu,T}^\pi}\left[\sum_{t=0}^{T} r(S_t, A_t) \,\middle|\, S_0 = s, A_0 = a\right]. \tag{2.1}$$

Q_T^π expresses the return the agent can expect from choosing some specific action a in state s before following the given policy, and thus provides a way to compare different actions.

An optimal policy $\pi^\star$ induces an optimal state-action value function $Q_T^{\pi^\star} := Q_T^\star$. Given only $Q_T^\star$, reconstructing $\pi^\star$ is straightforward: The definitions in eq. 1.3 and eq. 2.1 imply that the state value function and the state-action value function are linked by the policy:

$$v_T^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s)\, Q_T^\pi(s,a) \tag{2.2}$$

By definition, an optimal policy maximises the state value for each state. This means

$$v_T^{\pi^\star}(s) = \max_a Q_T^\star(s,a) \tag{2.3}$$

This suggests the optimal policy

$$\pi^\star(a \mid s) = \delta_{a a_{\max}} \quad \text{with} \quad a_{\max} = \arg\max_a Q_T^\star(s,a) \tag{2.4}$$

which is called the *greedy* policy with respect to $Q_T^\star$.

Q-learning provides a method to approximate $Q_T^\star$ directly: a slightly modified optimality criterion implies that $Q_T^\star$ is close to a solution of the *Bellman equation*. The Bellman equation is a fixed point equation for a contraction operator, which can be solved approximately by a well known iteration procedure [13].

Let π be a policy for an episodic reinforcement learning task with T time steps. The *discounted state value function* $v_{T,\gamma}^\pi : \mathcal{S} \to \mathbb{R}$ is defined by

$$v_{T,\gamma}^\pi(s) = \mathbb{E}_{P_{\mu,T}^\pi}\left[\sum_{t=0}^{T} r(S_t, A_t)\,\gamma^t \,\middle|\, S_0 = s\right], \tag{2.5}$$

where $\gamma \in (0,1]$ is the *discount factor*. If $\gamma < 1$, the contribution of a reward to the value of a state decreases exponentially with its delay. Q-learning approximates the optimal policy with respect to $v_{T,\gamma}^\pi$, which does not necessarly coincide with $\pi^\star$ defined in eq. 1.4. Yet, $\gamma < 1$ is needed to guarantee the convergence of the algorithm.

The Bellman equation is an approximate functional identity for the optimal *discounted state-action value function* $Q_{T,\gamma}^\star$, which is defined analogously to $v_{T,\gamma}^\pi$ in eq. 2.5. This functional identity emerges as an expansion of $Q_{T,\gamma}^\star$ along an optimal trajectory:

$$\begin{aligned}
Q_{T,\gamma}^\star(s,a) &= \mathbb{E}_{P_{\mu,T}^{\pi^\star}}\left[\sum_{t=0}^{T} r(S_t,A_t)\,\gamma^t \,\middle|\, S_0 = s, A_0 = a\right] \\
&= r(s,a) + \gamma \mathbb{E}_{P_{\mu,T}^{\pi^\star}}\left[\sum_{t=1}^{T} r(S_t,A_t)\,\gamma^{t-1} \,\middle|\, S_0 = s, A_0 = a\right] \\
&= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, \mathbb{E}_{P_{\mu,T-1}^{\pi^\star}}\left[\sum_{t=0}^{T-1} r(S_t,A_t)\,\gamma^t \,\middle|\, S_0 = s'\right] \\
&= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, v_{T-1,\gamma}^{\pi^\star}(s') \\
&\approx r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, v_{T,\gamma}^{\pi^\star}(s') \\
&= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q_{T,\gamma}^\star(s',a')
\end{aligned} \tag{2.6}$$

The approximation $v^{\pi^\star}_{T-1,\gamma}(s') \approx v^{\pi^\star}_{T,\gamma}(s')$ becomes an identity as T approaches infinity. We assume that T is large enough for eq. 2.6 to hold to high accuracy.

Let the Bellman operator $B^\star$ be an operator acting on functions $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, defined by

$$\left(B^\star Q\right)(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q(s',a') \tag{2.7}$$

Substituting $B^\star$ in eq. 2.6 yields

$$B^\star Q^\star_{T,\gamma} \approx Q^\star_{T,\gamma}$$

Therefore, $Q^\star_{T,\gamma}$ is close to a solution of the equation

$$B^\star Q = Q \tag{2.8}$$

The next step towards the Q-learning algorithm is the application of Banach's fixed point theorem to eq. 2.8. The theorem states that every contraction mapping $C : X \to X$ on a Banach space $(X, \|\cdot\|)$ yields a unique fixed point $x^\star \in X$. This fixed point is the limit of the series $(x_n)_{n \in \mathbb{N}}$, recursively defined by $x_{n+1} = Cx_n$.

A mapping $C : X \to X$ is a *contraction mapping* if $\exists \gamma \in [0,1)$ such that $\|Cx - Cy\| \leq \gamma \|x - y\| \; \forall x, y \in X$. The maximum norm $\|Q\|_\infty = \max_{(s,a) \in \mathcal{S} \times \mathcal{A}} |Q(s,a)|$ induces a metric on the space of functions $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that $B^\star$ acts on. Let Q and U be such functions.

$$\begin{aligned}
\|B^\star Q - B^\star U\|_\infty &= \max_{(s,a) \in \mathcal{S} \times \mathcal{A}} \left| \left(B^\star Q\right)(s,a) - \left(B^\star U\right)(s,a) \right| \\
&= \max_{(s,a) \in \mathcal{S} \times \mathcal{A}} \Bigg| r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q(s',a') \\
&\quad - r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} U(s',a') \Bigg| \\
&= \gamma \max_{(s,a) \in \mathcal{S} \times \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \left| \max_{a' \in \mathcal{A}} Q(s',a') - \max_{a' \in \mathcal{A}} U(s',a') \right| \\
&\leq \gamma \max_{(s,a) \in \mathcal{S} \times \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} \left| Q(s',a') - U(s',a') \right| \\
&\leq \gamma \max_{(s,a) \in \mathcal{S} \times \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \|Q - U\|_\infty \\
&= \gamma \|Q - U\|_\infty
\end{aligned} \tag{2.9}$$

Thus, the Bellman operator $B^\star$ is a contraction mapping if and only if $\gamma < 1$. The modified optimality criterion based on $v^\pi_{T,\gamma}$ guarantees a unique solution $Q^\star \approx Q^\star_{T,\gamma}$ of the Bellman equation. Furthermore, the series $(Q_n)_{n \in \mathbb{N}}$ defined by

$$Q_{n+1} = B^\star Q_n \tag{2.10}$$

approximates this solution; the relation eq. 2.10 gives rise to *Bellman iteration*. In Q-learning, the agent maintains an approximation Q of the optimal state-action value function $Q^\star_{T,\gamma}$ and steadily improves it according to eq. 2.10.

Still, one step of Bellman iteration requires updating the values of all state-action pairs $(s,a) \in \mathcal{S} \times \mathcal{A}$ - and is thus highly impractical. For this reason, in the final step of this derivation, we implement Bellman iteration as an efficient *stochastic approximation* algorithm.

Stochastic approximation provides a method to approximate the roots $\{x : f(x) = 0\}$ of a function f that can only be probed by noisy measurements [14]. Taking a noisy measurement of some value $f(x)$ means: sampling a random variable $F(x)$ that satisfies $(\mathbb{E}[F])(x) = f(x)$.

According to the theory of stochastic approximation, the series $(x_n)_{n\in\mathbb{N}}$ defined by

$$x_{n+1} = x_n + \alpha_n F(x_n) \tag{2.11}$$

converges to a root of f, provided that the ordinary differential equation (ODE) $\dot{x} = f(x)$ – which is asymptotically tracked by the series $(x_n)_{n\in\mathbb{N}}$ – features a stable equilibrium point $x^\star$. Two further technical conditions apply [14]:

1. The step size α_n must satisfy the *Robbins-Monro conditions*:

$$\sum_{n\in\mathbb{N}} \alpha_n = \infty \qquad\qquad \sum_{n\in\mathbb{N}} (\alpha_n)^2 < \infty \tag{2.12}$$

2. The series $(x_n)_{n\in\mathbb{N}}$ must be bounded.

This technique provably solves the Bellman equation eq. 2.8 for finite $\mathcal{S}$ and $\mathcal{A}$ [15]: A fixed point of $B^\star$ is a root of the function

$$f(Q)(s,a) = \left(B^\star Q\right)(s,a) - Q(s,a).$$

A transition in the respective reinforcement learning task allows a noisy measurement of f: Given a current state s and a selected action a, the agent receives the immediate reward $r(s,a)$ and observes the new state s', which is a sample of the random variable $S' \sim P(\cdot \mid s,a)$. The agent can then evaluate the expression

$$F(Q)(s,a) = r(s,a) + \gamma \max_{a'\in\mathcal{A}} Q(S',a') - Q(s,a)$$

According to the definition in eq. 2.7, $F(Q)$ satisfies $(\mathbb{E}F)(Q)(s,a) = f(Q)(s,a)$, and is therefore a noisy measurement of $f(Q)$. The stochastic approximation method eq. 2.11 now suggests the update

$$Q_{n+1}(s,a) = Q_n(s,a) + \alpha_n F(Q)(s,a) \tag{2.13}$$

The sequences $(Q_n(s,a))_{n\in\mathbb{N}}$ track a solution of the ODE system $\dot{Q}(s,a) = \left(B^\star Q\right)(s,a) - Q(s,a)$. The existence of a stable equilibrium point of that system is equivalent to the existence of a unique, stable fixed point of the operator $B^\star$. A discount factor $\gamma < 1$ guarantees both.

Using the update rule eq. 2.13, the agent improves its approximation of the optimal state-action value function after each transition. The more often some state-action pair (s,a) is visited, the more precise becomes the estimate $Q(s,a)$. In principle, the agent could just move through the environment randomly and eventually gain a good approximation of $Q^\star_{T,\gamma}$.

But to deduce the optimal strategy, the agent does not need a good approximation of $Q^\star_{T,\gamma}$ for every pair (s,a). In fact, it only needs a good approximation near an optimal trajectory. To systematically acquire that knowledge, the agent could use the best trajectory it has discovered so far, and try to improve it by exploring the surrounding trajectories. Such behaviour is implemented in the *epsilon-greedy* policy:

$$\pi^g_\epsilon(a \mid s) = \begin{cases} \delta_{a a_{\max}} \text{with } a_{\max} = \arg\max_a Q^\star_T(s,a) & \text{with prob. } 1-\epsilon \\ \text{const.} & \text{with prob. } \epsilon \end{cases} \tag{2.14}$$

Finally, this epsilon-greedy strategy and the update rule eq. 2.13 merge into the Q-learning algorithm [16]:

Algorithm 1 Q-Learning

Initialise $Q(s,a) = 0 \,\forall (s,a) \in \mathcal{S} \times \mathcal{A}$
loop forever
  Sample starting state s from $\mu(\cdot)$
  $t \leftarrow 0$
  repeat
    Sample action a from $\pi^g_\epsilon(a \mid s)$
    Execute a, observe r and s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$
    $t \leftarrow t+1$
  until $t = T$

The Q-learning algorithm is useful to learn tasks in environments with small $\mathcal{S}$ and $\mathcal{A}$. Experiment 4.1 describes how to apply Q-learning.

2.2 The Deep Q-Network algorithm

The Q-learning algorithm requires a table of size $|\mathcal{S}| \times |\mathcal{A}|$ that contains an estimate of the optimal state-action value function. For big, high dimensional state spaces, a

simple table is an inefficient representation of $Q_T^\star$: a state space of dimension d with n discretisation steps in each dimension requires a table of size $O\left(n^d\right)$. The number of parameters necessary to represent $Q_T^\star$ with a table thus grows exponentially in d.

Researchers of the company *DeepMind Technologies* instead used a deep neural network $\{Q_\theta\}$, the *deep Q network*, to represent $Q_T^\star$ [3]:

$$Q(s,a) \approx Q_\theta(s,a)$$

Rather than directly updating the value $Q(s,a)$ of single pairs (s,a), their agent improves its approximation $Q_\theta(s,a)$ by adapting the parameters θ of the neural network. It does so according to the DQN algorithm, which picks up the strategy of Q-learning.

The DQN algorithm resembles common supervised learning methods for neural networks. Section 1.2 introduced neural networks as function approximators. Consider the problem to approximate the function $F : \mathbb{R}^n \to \mathbb{R}^m$ with a neural network $\{f_\theta\}$.

Let $T = \{(x_1, F(x_1)), \ldots, (x_n, F(x_n))\}$ be a finite subset of the graph of F. T is called *training set*, while the pairs $(x_i, F(x_i))$ are referred to as *examples* consisting of an *input* x_i and a target $y := F(x_i)$.

Supervised learning means: determining the parameters of the neural network by fitting it to a training set. More formally, choose a *loss function* $L : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ to compare the output of the neural network with the targets. The formal objective of supervised learning is to find the parameter set $\theta^\star$ that minimises the loss averaged over the training set:

$$\theta^\star = \arg\min_\theta \frac{1}{n} \sum_{i=0}^{n} L(f_\theta(x_i), y_i) \tag{2.15}$$

Let (X, Y) be a tuple of uniformly distributed random variables that takes values in T. Then, equation 2.15 takes the form

$$\theta^\star = \arg\min_\theta \mathbb{E}\left[L(f_\theta(X), Y)\right] \tag{2.16}$$

Gradient descent is a standard technique to approach this optimisation problem. This technique provides an update rule

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta \mathbb{E}\left[L(f_\theta(X), Y)\right], \tag{2.17}$$

which generates a series of parameters $(\theta_k)_{k\in\mathbb{N}}$ that typically converges to $\theta^\star$ [7]. One gradient descent step requires the evaluation of the gradient for all examples in the potentially large training set. Just as in Q-learning in section 2.1, the theory of stochastic approximation provides an efficient implementation of gradient descent, called *stochastic gradient descent*.

$\nabla_\theta L(f_\theta(X), Y)$ is a noisy measurement of the averaged loss, since $\mathbb{E}[\nabla_\theta L(f_\theta(X), Y)] = \nabla_\theta \mathbb{E}[L(f_\theta(X), Y)]$. The fundamental update eq. 2.11 of stochastic approximation implies the iteration rule

$$\theta_{k+1} = \theta_k - \alpha_k \nabla_\theta L(f_\theta(X), Y) \tag{2.18}$$

The series generated by this iteration rule converges to $\theta^\star$ similarly to the series generated by eq. 2.17, provided that α_k satisfies the Robbins-Monro conditions eq. 2.12.

In practise, the method of choice is usually a mixture of gradient descent and stochastic gradient descent, called *batch gradient descent*. The update rule of batch gradient descent contains the gradient of the loss averaged over a small batch of examples sampled from the training set.

The deep Q network is meant to approximate $Q_T^\star$. Approaching this problem with supervised learning would require a training set, consisting of examples $\Big((s,a), Q_T^\star(s,a)\Big)$. Yet, the agent cannot access $Q_T^\star$ to provide the targets of such examples. However, it can use its current estimate Q_θ, together with the data it generated in previous transitions, to construct a better estimate of $Q_T^\star$ at some positions (s,a) using the Q-learning method.

Formally, the objective of the DQN algorithm is to minimise the difference between $Q_T^\star$ and Q_θ, which corresponds to finding a set of parameters $\theta^\star$ such that

$$\theta^\star = \arg\min_\theta \frac{1}{|\mathcal{S}||\mathcal{A}|} \sum_{(s,a)\in\mathcal{S}\times\mathcal{A}} L\Big(Q_\theta(s,a), Q_T^\star(s,a)\Big) \tag{2.19}$$

For $L(x,y) = (x-y)^2$, the update rule of stochastic gradient descent becomes

$$\theta_{k+1} = \theta_k - \alpha_k \Big(Q_\theta(s,a) - Q_T^\star(s,a)\Big) \nabla_\theta Q_\theta(s,a) \tag{2.20}$$

$Q_T^\star$ is unknown, but any previously experienced transition (s,a,r,s') enables the construction of an improved estimate U, inspired by the Q-learning update eq. 2.13:

$$U(s,a) = r + \gamma \max_{a'\in\mathcal{A}} Q(s',a') \tag{2.21}$$

Substituting the exact target $Q_T^\star(s,a)$ in eq. 2.20 by the improved estimate U yields the update rule of the DQN algorithm:

$$\theta_{k+1} = \theta_k - \alpha_k \left(Q_\theta(s,a) - U\right) \nabla_\theta Q_\theta(s,a) \tag{2.22}$$

Unlike Q-learning, the DQN algorithm does not necessarily converge; the nonlinearity of the neural network prohibits any such guarantees [13]. Yet, some tricks enable stable learning even without mathematically certain convergence. Among these tricks, *experience replay* enabled the 2013 breakthrough [3].

Strongly correlated training data, generated from consecutive transitions, contains almost redundant copies of information that cover only a tiny part of the system. Repeated updates using the almost same data do not only slow down learning, but may even lead to divergence.

Experience replay circumvents these difficulties; it exploits that any previously recorded transition is suitable to execute the update rule. Rather than updating the network parameters with the most recent transition, experience replay suggests to store the data of that transition in a *experience buffer* and instead update the network parameters with some past experience sampled from that buffer. This way, experience replay breaks correlations in the training data.

The DQN algorithm combines experience replay with the update rule eq. 2.22 and the ϵ-greedy policy:

Algorithm 2 Deep Q-Learning with Experience Replay

Initialise experience buffer $\mathcal{D}$
Initialise $Q_\theta(s,a)$ with random parameters θ
loop forever
    Sample starting state s from $\mu(\cdot)$
    $t \leftarrow 0$
    repeat
        Sample action a from $\pi_\epsilon^g(a \mid s)$
        Execute a, observe r and s'
        Store (s, a, r, s') in $\mathcal{D}$
        Sample $(\tilde{s}, \tilde{a}, \tilde{r}, \tilde{s}')$ from $\mathcal{D}$ at random
        $U \leftarrow \tilde{r} + \gamma \max_{\tilde{a}' \in \mathcal{A}} Q(\tilde{s}', \tilde{a}')$
        $\theta \leftarrow \theta - \alpha \left(Q_\theta(\tilde{s}, \tilde{a}) - U\right) \nabla_\theta Q_\theta(\tilde{s}, \tilde{a})$
        $s \leftarrow s'$
        $t \leftarrow t + 1$
    until $t = T$

Two examples illustrate deep Q-learning: experiment 4.2 features an implementation of the DQN algorithm designed to learn how to steer a car to the top of a mountain. Example 2.3 sketches the famous experiments in the Atari domain, published by the inventors of deep Q-learning [1].

2.3 Example: Learning to play Atari with Deep Q-Learning

The great potential of deep reinforcement learning became apparent through experiments with Atari games [1]. Agents that maintained deep, convolutional neural networks were trained to play a variety of different games, and reached remarkable scores. Even more remarkably, they only observed screened images.

The architecture of the neural network used by these agents is depicted in Figure 1.1. It roughly divides into a section for feature extraction from the input images, consisting of two convolutional layers, and followed by a section that relates the relevant features to according actions. This second section yields two fully connected layers.

The game emulations provided a screened image that did not fully characterise the state of the game. Therefore, the complete history of screens, starting from the beginning of the game, served as the description of the state. For practical reasons, the history was truncated after four frames.

Without any adaption of the algorithm or the architecture of the network, the same agent achieved better results than any previous reinforcement learning agent on 6 out of 7 games. On 5 out of 7 games, the agent reached a superhuman level.

These results were unprecedented at the time they were published. Since then, a rapid development in deep reinforcement learning lead to much more powerful algorithms [17]. Example 3.4 sketches one of the most successful algorithms at the time of writing.

3 Policy Gradient Methods

This chapter investigates a class of deep reinforcement learning algorithms called *policy gradient methods*. This class contains one of the most successful algorithms for deep reinforcement learning: the *Asynchronous Advantage Actor-Critic* (A3C) algorithm, which builds on the *actor-critic* concept.

The first section of this chapter contains a derivation of the basic technique underlying all policy gradient methods. The second section considers different ways to improve this basic technique. Finally, the third section presents the actor-critic concept.

3.1 The Policy Gradient

In every policy gradient method, a function approximator $\{\pi_\theta\}$, $\pi_\theta : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, represents the policy. In deep reinforcement learning, this function approximator is a deep neural network.

The parameters θ of π_θ provide direct access to the behaviour of the agent. Policy gradient methods optimise these parameters to reach an approximately optimal policy $\pi^\star \approx \pi_{\theta^\star}$. They employ gradient ascent; a practical expression of the respective gradient follows from the score function method.

Consider the problem of maximising the expression $\mathbb{E}_{p(\cdot|\theta)}[f]$ with respect to θ. Here, x is a real valued random variable with distribution $p(\cdot \mid \theta)$, and f is a real valued function.

Gradient ascent is a suitable approach to this problem. However, it requires the evaluation of the gradient

$$\nabla_\theta \mathbb{E}_{p(\cdot|\theta)}[f] \tag{3.1}$$

A different expression for that gradient suggests a way to do this evaluation.

$$\begin{aligned}\nabla_\theta \mathbb{E}_{p(\cdot|\theta)}[f] &= \nabla_\theta \int \mathrm{d}x\, p(x \mid \theta) f(x) \\ &= \int \mathrm{d}x\, f(x) \nabla_\theta p(x \mid \theta) \\ &= \int \mathrm{d}x\, f(x) p(x \mid \theta) \nabla_\theta \log(p(x \mid \theta)) \\ &= \mathbb{E}_{p(\cdot|\theta)}[f \nabla_\theta \log(p(\cdot \mid \theta))]\end{aligned} \tag{3.2}$$

$\nabla_\theta \log(p(x \mid \theta))$ is the *score function* of the distribution $p(x \mid \theta)$; it yields the direction of the fastest growth of p in the space of parameters θ at each x. Taking the product of the score function with f amplifies the growth of p in domains where f is large and suppresses it in domains of small f. The overall expectation thus yields a direction in parameter space that pushes p into domains of large f.

Evaluating the gradient in eq.. 3.1 by calculating or estimating the expectation in the last step of eq. 3.2 is referred to as the score function method [18]. To illustrate this method with a short example, let $p(x \mid \theta) = \exp\left(-(x-\theta)^2\right)$ and

$$f(x) = \begin{cases} 2 & \text{if } x \geq 0 \\ 1/2 & \text{otherwise} \end{cases}$$

Figure 3.1 shows the gradient field of $p(x \mid \theta)$, the product of the score function with f and the resulting direction in parameter space.

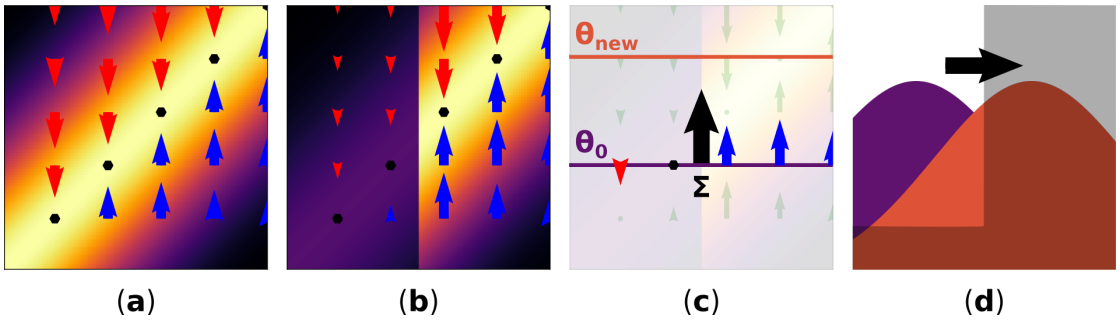

Figure 3.1 Diagram (a) shows $p(x \mid \theta)$ as a colour plot. The superimposed arrows represent the gradient with respect to θ. Diagram (b) depicts the product of the score function $\nabla_\theta \log(p(x \mid \theta))$ with f . Diagram (c) singles out the section of that product that belongs to a specific parameter value θ_0. The expectation over that section provides the direction in which θ should be shifted to enhance the probability in domains of high f. The result of that shift is the parameter θ_{new}. That shift of the distributions caused by $\theta_0 \rightarrow \theta_{new}$ is shown in diagram (d) Here, the shaded area depicts f.

Now, let us return to an episodic reinforcement learning task, which starts from an initial state sampled from the distribution $\mu(s)$. Policy gradient methods try to maximise

the expectation of the state value of the initial state, which is just the expected *return* $R := \sum_{t=0}^{T-1} r(S_t, A_t)$:

$$\begin{aligned} \mathbb{E}_\mu \left[v_T^{\pi_\theta} \right] &= \mathbb{E}_{P_{\mu,T}^{\pi_\theta}} \left[\sum_{t=0}^{T-1} r(S_t, A_t) \right] \\ &= \mathbb{E}_{P_{\mu,T}^{\pi_\theta}} [R] \end{aligned} \tag{3.3}$$

According to the score function method, the gradient of the expected return is an expectation:

$$\begin{aligned} \nabla_\theta \mathbb{E}_\mu \left[v_T^{\pi_\theta} \right] &= \mathbb{E}_{P_{\mu,T}^{\pi_\theta}} \left[R \nabla_\theta \log P_{s,T}^{\pi_\theta} \right] \\ &= \mathbb{E}_{P_{\mu,T}^{\pi_\theta}} \left[R \nabla_\theta \sum_{t=0}^{T-1} \log P(S_{t+1} \mid S_t, A_t) + \log \pi_\theta(A_t \mid S_t) \right] \\ &= \mathbb{E}_{P_{\mu,T}^{\pi_\theta}} \left[R \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t \mid S_t) \right] \end{aligned} \tag{3.4}$$

For gradient ascent, the exact value of this expectation is still necessary. However, any unbiased estimator of the expectation suffices for stochastic gradient ascent, a method previously introduced in eq. 2.18. Analogous to the example depicted in figure 3.1, multiplication with R amplifies the growth of the probability of trajectories that earned much reward. Thus, updating the parameters of the policy using the gradient from def. 3.4 reinforces successful sequences of actions.

Policy methods all draw on the gradient from def. 3.4. Yet, they use different techniques to estimate it, and incorporate it in different stochastic approximation schemes. For example, replacing the expectation over all trajectories with an average over a batch of trajectories sampled from $P_T^{\pi_\theta}(\omega)$ yields a *Monte Carlo* estimator G of the gradient:

$$G = \frac{1}{n} \sum_{i=1}^{n} \left(R(\omega_i) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t(\omega_i) \mid S_t(\omega_i)) \right) \tag{3.5}$$

The Monte Carlo estimator combined with stochastic gradient ascent constitutes the REINFORCE algorithm, which was invented by R. Williams in 1992 [19].

Algorithm 3 REINFORCE

1: Initialise $\pi_\theta(s \mid a)$ with random parameters θ
2: **loop** forever
3: Generate batch of trajectories $\{\omega_i = (S_{0,i}, A_{0,i}, \ldots, S_{T,i})\}$ using $\pi_\theta(s \mid a)$
4: $R_i \leftarrow \sum_{t=0}^{T-1} r_{t,i} \; \forall i$
5: $G \leftarrow \frac{1}{n} \sum_{i=1}^{n} R_i \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_{t,i} \mid S_{t,i})$
6: $\theta \leftarrow \theta + \alpha G$

Policy gradient methods have two big advantages over the Q-learning method described in chapter 2:

1. A strongly stochastic policy is beneficial in the beginning of the learning process: the agent should explore many different strategies. As the agent gathers more knowledge, its policy should ideally collapse to the optimal policy, which might be deterministic. Q-learning is implemented with a fixed stochastic element in the ϵ-greedy policy, which makes such adaption impossible. In contrast, policy gradient methods have access to every aspect of the policy; an adaption from highly stochastic to deterministic is possible without further modifications.

2. Policy gradient methods are remarkably compatible with deep learning, they incorporate deep neural networks in a straightforward way. As opposed to Q-learning, policy gradient methods directly turn into deep reinforcement learning methods.

Still, a simple stochastic gradient ascent based on the Monte Carlo estimator defined in def. 3.5 will likely converge slower than the Q-learning algorithm. The reason for this is the high variance of that estimator. The next section provides various ways to improve the estimator. With these improvements, policy gradient methods have developed into the most powerful class of deep reinforcement learning algorithms so far.

3.2 Variance Reduction

Three modifications of the gradient formula reduce the variance of the Monte Carlo estimator. Firstly, a significant part of the expression def. 3.4 is identically zero. Therefore, it does only contribute to the variance of the Monte Carlo estimator, but not to the estimate [20]. Leaving it out will reduce the variance without damaging the estimator.

When considering the causal structure of the estimator, we will find that parts of the estimator are obsolete: an action taken at time step t only affects the rewards at time steps $t' \geq t$. Earlier rewards are not affected, and the respective terms vanish. The following calculation proves this:

$$
\begin{aligned}
&\mathbb{E}_{P_{\mu,T}^{\pi_\theta}}\left[R(\omega)\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \mathbb{E}_{P_{\mu,T}^{\pi_\theta}}\left[\sum_{v=0}^{T-1} r(S_v, A_v)\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \sum_{t=0}^{T-1}\mathbb{E}_{s_0:s_t, a_0:a_{t-1}}\mathbb{E}_{s_{t+1}:s_T, a_t:a_{T-1}}\left[\left(\sum_{v=0}^{t-1} r(S_v, A_v) + \sum_{v=t}^{T-1} r(S_v, A_v)\right)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \sum_{t=0}^{T-1}\mathbb{E}_{s_0:s_t, a_0:a_{t-1}}\mathbb{E}_{s_{t+1}:s_T, a_t:a_{T-1}}\left[\sum_{v=0}^{t-1} r(S_v, A_v)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&+ \mathbb{E}_{P_T^{\pi_\theta}}\left[\sum_{t=0}^{T-1}\sum_{v=t}^{T-1} r(S_v, A_v)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right]
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_0:s_t, a_0:a_{t-1}} \left[\sum_{v=0}^{t-1} r(S_v, A_v) \mathbb{E}_{s_{t+1}:s_T, a_t:a_{T-1}} \nabla_\theta \log \pi_\theta(A_t \mid S_t) \right] \\
&+ \mathbb{E}_{P_{\mu,T}^{\pi_\theta}} \left[\sum_{t=0}^{T-1} \sum_{v=t}^{T-1} r(S_v, A_v) \nabla_\theta \log \pi_\theta(A_t \mid S_t) \right] \\
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_0:s_t, a_0:a_{t-1}} \left[\sum_{v=0}^{t-1} r(S_v, A_v) \sum_{a_t \in \mathcal{A}} \pi_\theta(a_t \mid S_t) \frac{\nabla_\theta \pi_\theta(a_t \mid S_t)}{\pi_\theta(a_t \mid S_t)} \right] \\
&+ \mathbb{E}_{P_T^{\pi_\theta}} \left[\sum_{t=0}^{T-1} \sum_{v=t}^{T-1} r(S_v, A_v) \nabla_\theta \log \pi_\theta(A_t \mid S_t) \right] \\
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_0:s_t, a_0:a_{t-1}} \left[\sum_{v=0}^{t-1} r(S_v, A_v) \nabla_\theta \sum_{a_t \in \mathcal{A}} \pi_\theta(a_t \mid S_t) \right] \\
&+ \mathbb{E}_{P_{\mu,T}^{\pi_\theta}} \left[\sum_{t=0}^{T-1} \sum_{v=t}^{T-1} r(S_v, A_v) \nabla_\theta \log \pi_\theta(A_t \mid S_t) \right] \\
&= \mathbb{E}_{P_{\mu,T}^{\pi_\theta}} \left[\sum_{t=0}^{T-1} \sum_{v=t}^{T-1} r(S_v, A_v) \nabla_\theta \log \pi_\theta(A_t \mid S_t) \right]
\end{aligned}
$$

The notation $\mathbb{E}_{s_0:s_t, a_0:a_{t-1}}$ means

$$
\begin{aligned}
\mathbb{E}_{s_0:s_t, a_0:a_{t-1}} f = \int & \mathrm{d}s_0 \mathrm{d}a_0 \cdots \mathrm{d}s_{t-1} \mathrm{d}a_{t-1} \mathrm{d}s_t \\
& \times \mu(s_0)\, \pi(a_0 \mid s_0) P(s_1 \mid s_0, a_0) \ldots \pi(a_{t-1} \mid s_{t-1}) P(s_t \mid s_{t-1}, a_{t-1}) \\
& \times f(s_0, a_0, \cdots, s_{t-1}, a_{t-1}, s_t)
\end{aligned}
\tag{3.6}
$$

For each time step t, the return is split into the sum of rewards earned earlier than t and the sum of rewards earned at step t and later. Since the rewards earned earlier than t do not depend on the action at step t, they are not affected by the respective expectation. That expectation then only contains the score function of the policy, and thus vanishes.

The policy gradient becomes

$$
\begin{aligned}
&\mathbb{E}_{P^{\pi_\theta}_{\mu,T}}\left[\sum_{t=0}^{T-1}\sum_{v=t}^{T-1} r(S_v, A_v)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_0:s_t, a_0:a_{t-1}} \mathbb{E}_{s_{t+1}:s_T, a_t:a_{T-1}}\left[\sum_{v=t}^{T-1} r(S_v, A_v)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_t, a_t}\left[\mathbb{E}_{s_{t+1}:s_T, a_{t+1}:a_{T-1}}\left[\sum_{v=t}^{T-1} r(S_v, A_v)\right]\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_t, a_t}\left[Q^{\pi_\theta}_{T-t}(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \mathbb{E}_{P^{\pi_\theta}_{\mu,T}}\left[\sum_{t=0}^{T-1} Q^{\pi_\theta}_{T-t}(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right]
\end{aligned}
\tag{3.7}
$$

The state-action value function, which was introduced for Q-learning in section 2.1, reappears in the purified expression for the policy gradient.

The final result of the above calculations,

$$
\nabla_\theta \mathbb{E}_\mu\left[v^{\pi_\theta}_T\right] = \mathbb{E}_{P^{\pi_\theta}_{\mu,T}}\left[\sum_{t=0}^{T-1} Q^{\pi_\theta}_{T-t}(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right], \tag{3.8}
$$

is the *policy gradient theorem* [16].

The second modification of the policy gradient is due to a *control variate*. Control variates exploit further knowledge about an unknown quantity m that is estimated. Another strongly correlated and known quantity c represents that knowledge. Let M and C be estimators such that $\mathbb{E}[M] = m$ and $\mathbb{E}[C] = c$. If M and C are strongly correlated, their estimation errors are correlated, too. Explicitly, the known error $\Delta_c = C - c$ then resembles the unknown estimation error $\Delta_m = M - m$. The estimator M might be corrected accordingly:

$$
m = M - \Delta_m \approx M - \Delta_c = M - C + c := M_{\mathrm{CV}}
$$

Since $\mathbb{E}[M_{\mathrm{CV}}] = m$, M_{CV} is an unbiased estimator of m. Yet,

$$
\mathrm{Var}[M_{\mathrm{CV}}] = \mathrm{Var}[M - C] = \mathrm{Var}[M] + \mathrm{Var}[C] - 2\mathrm{Cov}[M, C]
$$

For $\mathrm{Var}[C] \leq 2\mathrm{Cov}[M, C]$, M_{CV} has less variance than M. For the Monte Carlo estimator of the policy gradient, consider the expression

$$
\sum_{t=0}^{T-1} b(S_t)\nabla_\theta \log \pi_\theta(A_t \mid S_t)
$$

The expectation of this expression is known - it vanishes identically:

$$
\begin{aligned}
&\mathbb{E}_{P_{\mu,T}^{\pi_\theta}}\left[\sum_{t=0}^{T-1} b(S_t)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_0:s_t, a_0:a_{t-1}} \mathbb{E}_{s_{t+1}:s_T, a_t:a_{T-1}} \left[b(S_t)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] \\
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_t}\left[\mathbb{E}_{a_t}\left[\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] b(S_t)\right] \\
&= \sum_{t=0}^{T-1} \mathbb{E}_{s_t} b(S_t)\left[\nabla_\theta \sum_{a_t \in \mathcal{A}} \pi_\theta(a_t \mid S_t)\right] \\
&= 0
\end{aligned}
\tag{3.9}
$$

Implementing this control variate in the policy gradient yields

$$
\nabla_\theta \mathbb{E}_\mu\left[v_T^{\pi_\theta}\right] = \mathbb{E}_{P_{\mu,T}^{\pi_\theta}}\left[\sum_{t=0}^{T-1}\left(Q_{T-t}^{\pi_\theta}(S_t, A_t) - b(S_t)\right)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right], \tag{3.10}
$$

where b is called the *baseline*.

According to [21], a nearly optimal and very common choice for b is the state value $v_{T-t}^{\pi_\theta}$. That quantity is indeed strongly correlated with the state-action value $Q_{T-t}^{\pi_\theta}$. The difference $Q_{T-t}^{\pi_\theta}(s,a) - v_{T-t}^{\pi_\theta}(s) := a_{T-t}^{\pi_\theta}(s,a)$ is called the *advantage* of the action a in the state s, the respective expression for the gradient reads

$$
\nabla_\theta \mathbb{E}_\mu\left[v_T^{\pi_\theta}\right] = \mathbb{E}_{P_{\mu,T}^{\pi_\theta}}\left[\sum_{t=0}^{T-1} a_{T-t}^{\pi_\theta}(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right], \tag{3.11}
$$

At this point, a comparison with supervised learning provides some intuition for the policy gradient method. The *cross entropy function* $L : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, $L(x,y) = \langle \log x, y\rangle$ is a common loss function used for supervised learning. Writing

$$
\mathbb{E}_{P_{\mu,T}^{\pi_\theta}}\left[\sum_{t=0}^{T-1} a_{T-t}^{\pi_\theta}(S_t, A_t)\log \pi_\theta(A_t \mid S_t)\right] = \mathbb{E}_{P_{\mu,T}^{\pi_\theta}}\left[\sum_{t=0}^{T-1} L\left(\pi_\theta(A_t \mid S_t), a_{T-t}^{\pi_\theta}(S_t, A_t)\right)\right]
$$

emphasises the similarity between policy gradient methods and supervised learning, as described in section 2.2: Neglecting the dependence of the $A_{T-t}^{\pi_\theta}(S_t, A_t)$ and $\mathbb{E}_{P_{\mu,T}^{\pi_\theta}}$ on θ, a gradient descent procedure to minimise that loss would rely on exactly the gradient from def. 3.11. Such a procedure would fit the policy to the advantage function of the initial policy. This advantage function, in turn, states how much more reward the agent can expect by choosing some specified action rather than following the underlying initial policy. Consequently, the policy would change in favour of the most promising action [22].

In contrast to that supervised learning process, policy gradient methods use an estimate of the advantage function, and update it at every step. They thus feature dynamical targets which keep track of recent changes of the policy.

The third means to reduce the variance of the Monte Carlo estimator is a discount factor γ. This discount factor, which already appeared in section 2.1, exponentially suppresses delayed rewards. It enters the policy gradient as the discounted state value function $v^{\pi_\theta}_{T,\gamma}(s)$ substitutes the undiscounted state value function $v^{\pi_\theta}_{T}(s)$ as the optimisation objective defined in def. 3.3. The discounted policy gradient reads

$$\nabla_\theta \mathbb{E}_\mu \left[v^{\pi_\theta}_{T,\gamma} \right] = \mathbb{E}_{P^{\pi_\theta}_{\mu,T}} \left[\sum_{t=0}^{T-1} a^{\pi_\theta}_{T-t,\gamma}(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t \mid S_t) \right], \tag{3.12}$$

with $a^{\pi_\theta}_{T-t,\gamma}(s,a) = Q^{\pi_\theta}_{T-t,\gamma}(s,a) - v^{\pi_\theta}_{T-t,\gamma}(s)$

The Monte Carlo estimator corresponding to that expectation is

$$G_{\text{Adv.},\gamma} = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t(\omega_i) \mid S_t(\omega_i)) A^{\pi_\theta}_{T-t,\gamma}(S_t(\omega_i), A_t(\omega_i)) \tag{3.13}$$

where

$$A^{\pi_\theta}_{T-t,\gamma}(S_t(\omega_i), A_t(\omega_i)) = \sum_{v=t}^{T-1} r_{t,i} \gamma^{v-t} - \sum_{j=1}^{n} \delta_{S_t(\omega_i)\, S_t(\omega_j)} \sum_{v=t}^{T-1} r_{t,j} \gamma^{v-t} \tag{3.14}$$

It is a biased estimator of the undiscounted gradient from def. 3.11, but typically, it has less variance [23]. Here, discounting exemplifies a common theme in statistics: the bias-variance tradeoff.

So far, this chapter only presented Monte Carlo estimators to approximate the policy gradient. Example 4.3 shows how to implement a Monte Carlo estimator to solve the Frozen Lake Environment of experiment 4.1. In the next section, a new type of estimator enters the discussion. That type defines the last class of learning algorithms in this essay, the so called actor-critic methods.

3.3 The Actor-Critic Concept

For policy based methods, a low noise estimate of the policy gradient is key to efficient learning. Even with different improvements, Monte Carlo estimation is a very noisy and inefficient way to approximate that gradient.

The *actor-critic* concept suggests: Instead of generating a completely new, noisy Monte Carlo estimate at each step, the agent should maintain – and successively improve – an approximation of a value function to estimate the advantage [24].

Chaper 2 contains a method to successively learn the optimal state-action value function. Analogous methods exist to learn the state value function or the action state value function for some given policy π. Such methods are called *value based* methods. Actor-critic methods merge policy gradient techniques and value based techniques.

More specifically, to estimate the gradient from eq. 3.11, the agent must estimate the advantage function $a^{\pi_\theta}_{T-t}(s,a) = Q^{\pi_\theta}_{T-t}(s,a) - v^{\pi_\theta}_{T-t}(s)$. Since

$$Q^{\pi_\theta}_{T-t}(s,a) = \mathbb{E}^{\pi}_{(s,a),T-t}\left[\sum_{v=0}^{T-t} r(S_v, A_v)\right] = r(s,a) + \mathbb{E}_{P(\cdot|s,a)}\left[v^{\pi_\theta}_{T-t-1}\right],$$

the expression

$$A^{\pi_\theta}_{T-t}(S_t, A_t) = r(S_t, A_t) + v^{\pi_\theta}_{T-t-1}(S_{t+1}) - v^{\pi_\theta}_{T-t}(S_t)$$

is an unbiased estimator for the advantage function. Now, assume that the agent can access an approximation V^{π}_t of v^{π}_t. It could then approximate the advantage function with the estimator

$$A^{\pi_\theta}_{T-t,(0)}(S_t, A_t) = r(S_t, A_t) + V^{\pi_\theta}_{T-t-1}(S_{t+1}) - V^{\pi_\theta}_{T-t}(S_t) \tag{3.15}$$

Only the uncertainty in the transition to the next step contributes to the variance of that estimator, which is therefore lower than that of any Monte Carlo estimator. On the other hand, the expectation of $A^{\pi_\theta}_{T-t,(0)}$ deviates from the exact advantage function if V^{π}_t deviates from the exact state value function – using V^{π}_t introduces bias.

Typically, the agent learns the approximation V^{π}_t with methods analogous to Q-learning and optimises the policy simultaneously. Therefore, V^{π}_t generically contains data generated with many different policies. In the literature of reinforcement learning, using previous estimates to construct new ones is called *bootstrapping*. Bootstrapping usually introduces bias and often prevents theoretical convergence guarantees, but enhances practical efficiency [16].

In experiments, a mixture in between the two extremes of pure Monte Carlo and the immediate bootstrap of def. 3.15, is most succesful [16]:

$$A^{\pi_\theta}_{T-t,(n)}(S_t, A_t) = \sum_{i=0}^{n} r(S_{t+i}, A_{t+i}) + V^{\pi_\theta}_{T-t-n-1}(S_{t+n+1}) - V^{\pi_\theta}_{T-t}(S_t) \tag{3.16}$$

This estimator cuts the Monte Carlo return after n steps and replaces the remainder with a bootstrapped approximation. This way, the effects of the current action a_t in combination with the current policy π^θ are simulated n steps into the future. According to that simulation, a_t and π^θ lead to a state S_{t+n+1} after $n+1$ steps. Finally, the agent evaluates that state using the approximation V^{π}_t that it learned from its previous experience.

The estimator defined in eq. 3.16 is at the heart of one of the most successful deep reinforcement learning algorithms at the time of writing: the A3C algorithm. To approximate v_t^π, the A3C agent maintains a deep neural network v_θ which it updates according to

$$\theta_{k+1} = \theta_k + \alpha \left(\sum_{i=0}^{n} r(S_{t+i}, A_{t+i}) \gamma - v_\theta(S_{t+n+1}) \gamma^{n+1} - v_\theta(S_t) \right) \nabla_\theta v_\theta$$

This update rule belongs to the value based $TD(n)$ algorithm [16]. The example 3.4 sketches the implementation and the results of the A3C algorithm from the original publication [17].

3.4 Example: The Asynchronous Advantage Actor-Critic Algorithm

Asynchronous algorithms use a trick to speed up computation [20]: Instead of only a single agent, asynchronous algorithms create multiple agents that interact with different copies of the environment. Still, they use and update the same shared policy network – they share their experience.

In the original implementation in [17], a single deep neural network represents both the policy and the state value function. These objects simply correspond to different outputs of the final layer, and therefore rely on the same feature extraction. Apart from that final layer, the architecture of the neural network is identical to that in example 2.3.

A3C and DQN directly competed in the Atari domain. On average, A3C performed significantly better then DQN, even with less training time. Furthermore, the A3C algorithm readily generalises to tasks with continuous action spaces: the finite number of outputs of the policy gradient network then provides means and covariances of a multivariate gaussian distribution over the action space. With that modification, the A3C algorithm learned successful strategies in a set of continuous control tasks.

This applicability to tasks with continuous action spaces is a remarkable advantage of the A3C over value based methods, which cannot straightforwardly be generalised for such tasks.

4 Experiments

This chapter describes three experiments, conducted to underpin the above presented theory. For each experiment, the respective section states the task and elaborates on how it is simulated in the computer. Then, it describes how the particular algorithm is implemented, and finally provides the results of the experiment.

4.1 Tabular Q-Learning in the Frozen Lake Environment

The agent navigates through a two-dimensional world in order to reach a goal position. There are two complications: Firstly, the world contains holes. These holes are parts of the world from which the agent cannot escape. Secondly, due to unpredictable external influences, the agent is not in full control of where it moves next.

A model divides the two-dimensional world into a grid of cells. The position of the agent is sufficiently characterised by the cell s that contains it. The set of all cells constitutes a suitable state space. Some of the cells are hole cells, and the cell containing the goal position is the goal cell s_{Goal}. Figure 4.1 displays a possible scenario. The agent controls the system by moving from cell to cell. Its options are gathered in the action space $A = \{\text{up}, \text{down}, \text{left}, \text{right}\}$.

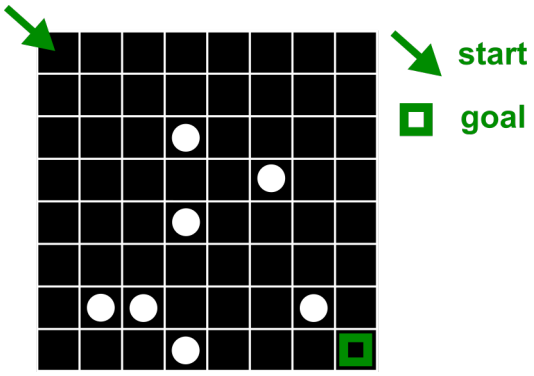

Figure 4.1 A square piece of the two-dimensional world is divided in 8×8 cells. Regular cells are black, hole cells contain a white dot. The goal cell is marked by a green square. The green arrow points to the cells through which the agent enters the world.

The dynamics of the environment entangle actions and states: From a regular cell, the agent moves to the neighbouring cell in the selected direction with probability $p \in [0, 1]$. Yet, with probability $1 - p$, it moves to a randomly chosen neighbouring cell. This stochastic component models any unpredictable, external influences. From a hole cell, the agent cannot move to any other cell; it will remain in the hole cell for all remaining time steps independent of the actions it chooses.

Finally, the reward function specifies the task. The task consists in reaching the goal cell, therefore

$$r(s, a) = \begin{cases} 1 & \text{if } s_{Goal} \text{ borders upon } s \text{ in direction } a \\ 0 & \text{otherwise} \end{cases}$$

is a suitable reward function: The agent gets rewarded for choosing to move onto the goal cell.

The gym package provided by the company OpenAI contains an implementation of this environment with the name FrozenLake-v0 [25].

A simple two-dimensional matrix suffices to represent the estimate $Q(s, a)$ of the optimal state-action value function. Four columns correspond to the actions {up, down, left, right}, and each row corresponds to a different cell.

Each episode begins with placing the agent at the starting position. From there, the agent moves through the grid of cells. The ϵ-greedy policy serves as a navigation system. Initially, this navigation system is informed by a very poor estimate of $Q_T^\star$, and generates a random walk. Eventually, the agent will coincidentally discover the goal cell and receive the respective reward. According to the update rule 2.13, this newly discovered value of the goal state induces increasing values of the surrounding cells. The increase in value continues to spread until the agent has worked out an estimate of $Q_T^\star$ that directly leads it to the goal state from its initial position.

Figure 4.2 depicts the estimated state value function $V(s) = \max_{a \in \mathcal{A}} Q(s, a)$ and the greedy policy the agent derives from its estimate Q. Both are shown after 400, 720 and 20000 episodes of learning. The Python script tabular_q_learning.py that generated the figure 4.2 is available at the repository [4].

4.2 Deep Q-Learning in the Mountain Car Environment

The agent controls a car that moves on a one-dimensional, mountainous track. The car can accelerate either forward or backward with some fixed acceleration, or roll freely. This acceleration competes with the gravitational force. The task consists in reaching some specified goal position as fast as possible. The scenario is illustrated in figure 4.3

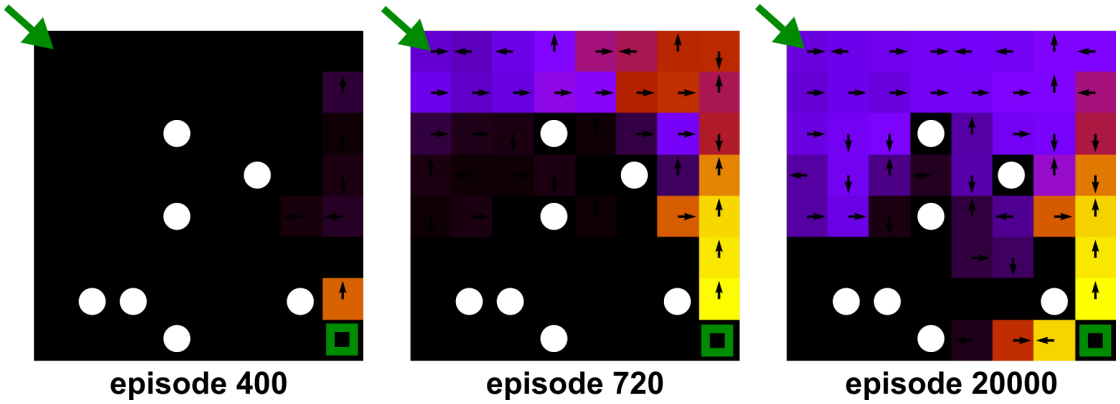

Figure 4.2 In the environment of Figure 4.1, the estimated state value V is represented by the colour of the cells. The arrows indicate the greedy choice of action $a = \arg\max_a Q(s,a)$ for every state s. Apparently, the greedy strategy does not indicate the direct route to the goal state; rather, it helps to avoid holes.

Two parameters describe the dynamical state of the system: the position $x \in \mathbb{R}$ and the velocity $\dot{x} \in \mathbb{R}$ take values within the compact intervals $\mathcal{I}_x \subset \mathbb{R}$ and $\mathcal{I}_{\dot{x}} \subset \mathbb{R}$. Thus, the state space is $\mathcal{S} = \mathcal{I}_x \times \mathcal{I}_{\dot{x}} \subset \mathbb{R}^2$. A computer discretises this compact subset of $\mathbb{R}^2$ and thus yields a two-dimensional, big but finite state space. The agent picks its actions from the action space $\mathcal{A} = \{+1, 0, -1\}$ where $+1$ corresponds to forward acceleration, while -1 corresponds to backward acceleration. 0 corresponds to no acceleration at all.

The dynamics of the environment are deterministic: the total acceleration of the car is the sum of the fixed acceleration $\ddot{x}_0$, controlled by the agent, and the gravitational acceleration $\ddot{x}_g = g\,m(x)$ which depends on the slope m at the position x of the car.

$$\ddot{x}_a = \ddot{x}_g + a\ddot{x}_0 \qquad\qquad a \in \mathcal{A} \tag{4.1}$$

Let Δt denote the time increment between two successive time steps. Assuming an acceleration $\ddot{x}_t$ and a velocity $\dot{x}_t$ at time step t, the position x_t and velocity of the car change to first order in Δt according to

$$x_{t+1} = x_t + \Delta t\,\dot{x}_t \qquad\qquad \dot{x}_{t+1} = \dot{x}_t + \Delta t\,\ddot{x}_t \tag{4.2}$$

The transition probability function P therefore reads

$$P\left((x',\dot{x}') \,|\, (x,\dot{x}), a\right) = \delta\left(x' - x - \Delta t\,\dot{x}\right)\,\delta\left(\dot{x}' - \dot{x} - \Delta t\ddot{x}_a\right)$$

The reward function implements the task of reaching the goal position by punishing the agent with a constant negative reward at every time step until the car reaches the goal.

The mountain car environment can be found in gym under the name MountainCar-v0.

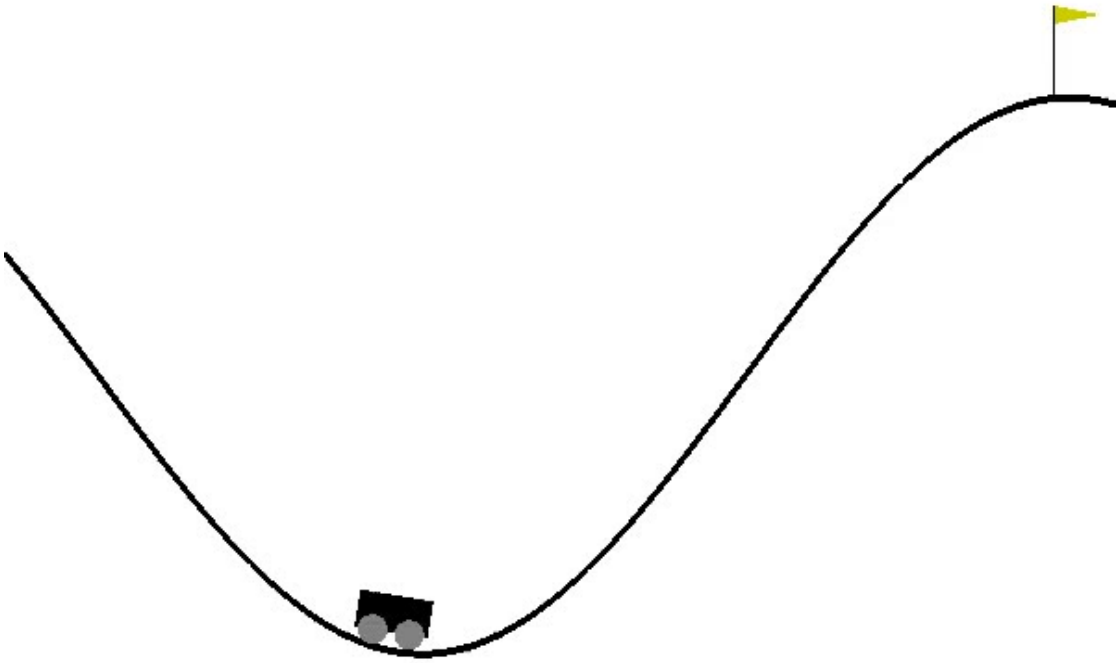

Figure 4.3 The little black cubic car is supposed to reach the flag on top of the hill.

To allow for an application of the *DQN* algorithm, the agent is equipped with a neural network $Q_\theta : \mathcal{S} \to \mathbb{R}^{|\mathcal{A}|}$ that features two layers. Both layers are fully connected and joined through a tanh-nonlinearity. The output dimension of the second layer must be equal to the number of elements in the action space and is thus three, while the output dimension m of the first layer can be varied.

For sufficiently large m, the agent finds strategies to quickly reach the goal position. Figure 4.4 shows estimates of the optimal state value function $v_T^{\pi^\star}$ derived from $Q_\theta(a \mid s)$, as well as the greedy choice of action. Those quantities are superimposed with the trajectory of the car. The trajectory reaches the goal position more efficiently as the precision of the estimate increases.

The behaviour depicted in figure 4.4 was produced by an agent that used an output dimension $m = 128$ of the first layer of its deep Q-network. The respective Python code `deep_q_learning.py` is accessible at the repository [4].

4.3 Discounted REINFORCE in the Frozen Lake Environment

The Frozen Lake environment, already subject of experiment 4.1, does not enforce a deep neural network model. its small state space allows for a linear model; a *softmax*

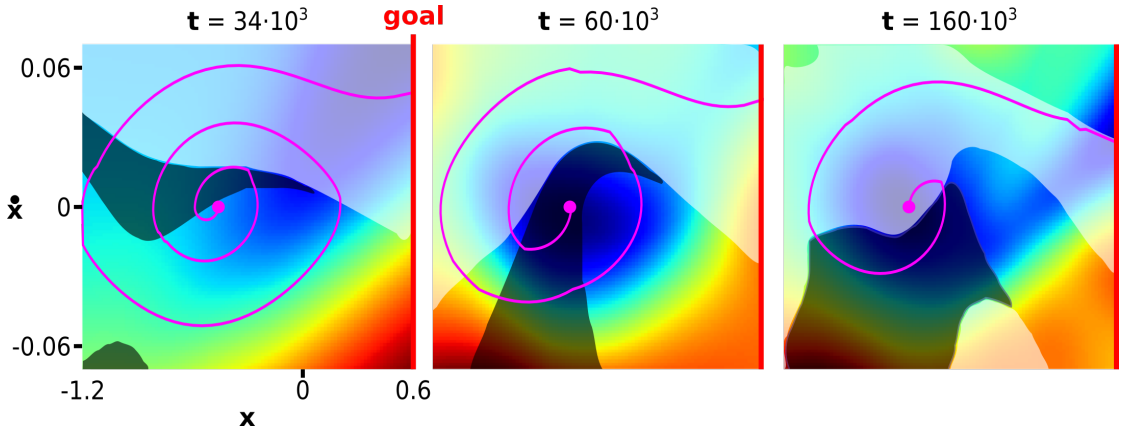

Figure 4.4 The colourmap represents the estimate of the optimal state value function, the shading encodes the greedy choice of action: In the dark areas, the greedy policy suggests acceleration to the left; in the light areas, it suggests acceleration to the right. In the neutral areas, the greedy policy suggest no acceleration. The most recent finished trajectory is depicted as a pink curve.

function turns the output of that linear model into a normalised probability distribution.

Explicitly, the model for the policy is

$$\pi_\theta : \mathbb{R}^{|\mathcal{S}|} \to \mathcal{P}(\mathcal{A}) \qquad \pi_\theta(v_s) = \text{softmax}(\Theta v_s) \tag{4.3}$$

where $v_s \in \mathbb{R}^{|\mathcal{S}|}$ is a vector that encodes the state $s \in \{s_1, s_2, \ldots, s_{|\mathcal{S}|}\} = \mathcal{S}$ according to

$$(v_s)^a = \begin{cases} 1 & \text{if } s = s_a \\ 0 & \text{otherwise} \end{cases}$$

The softmax function is defined by

$$\text{softmax}^a(x) = \frac{\exp(-x^a)}{\sum_b \exp\left(x^b\right)}$$

$\Theta \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$ is a linear transformation . The agent optimises the components of that transformation according to the REINFORCE algorithm 3, with two modifications:

1. The Monte Carlo estimator for the gradient is derived from the purified expression at the end of eq. 3.8
2. The estimator features a factor $\gamma \in (0, 1]$ that discounts the rewards.
3. In addition to the Monte Carlo estimator, the gradient also contains a contribution from a *total entropy regularisation*. The respective loss term is defined as

$$L_S = \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1}\sum_{a\in\mathcal{A}}\pi_\theta(a \mid S_t(\omega_i))\log\pi_\theta(a \mid S_t(\omega_i))$$

This term measures how deterministic the policy is. Using it as a loss term punishes highly deterministic policies, and thus prevents premature convergence.

Accordingly, the estimator reads

$$\begin{aligned} G &= \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(A_t(\omega_i) \mid S_t(\omega_i))\left(Q^{\pi_\theta}_{T-t,\gamma}\right)_i + \nabla_\theta L_S \\ \left(Q^{\pi_\theta}_{T-t,\gamma}\right)_i &= \sum_{v=t}^{T-1} R_{t,i}\gamma^{v-t} \end{aligned} \tag{4.4}$$

Using the above estimator in the REINFORCE algorithm, the agent successfully learns how to reach the goal state without falling into a hole. Figure 4.5 shows the learning curve, and the relation between the speed of learning and the discount factor γ. The respective source code files discounted_REINFORCE.py and evaluate_discounted_REINFORCE.py are available at the repository [4]. These programs are based on an implementation of the REINFORCE algorithm by John Schulman, which can be found at [26]

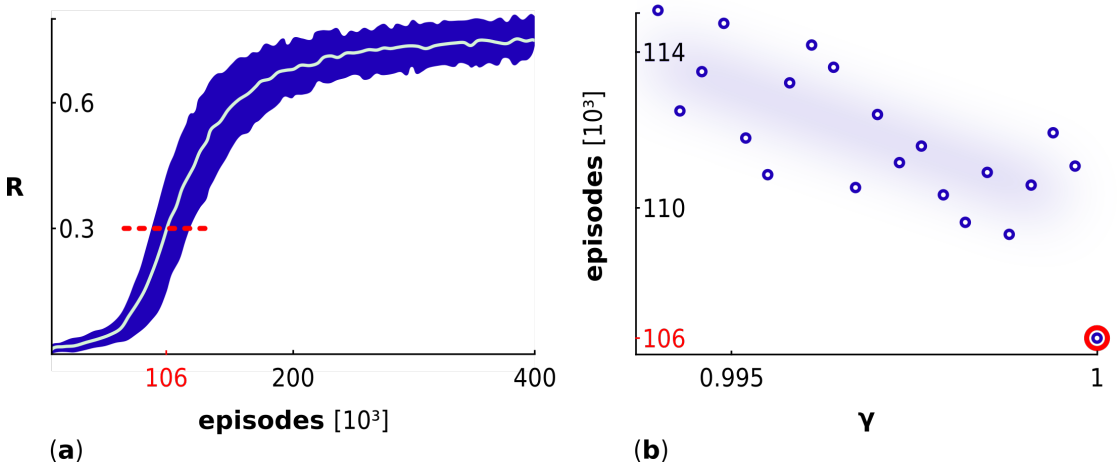

Figure 4.5 Diagram (a) shows the mean and standard deviation of the total return R as a function of the number of training episodes, extracted from 100 runs. Diagram (b) shows the number of training episodes necessary to achieve a total reward $R > 0.3$, averaged over 100 runs, and the respective parameter γ. The red circle marks the datapoint corresponding to $\gamma = 1$. In this environment, learning is fastest for $\gamma = 1$ – in contrast to the theory presented above.

Surprisingly, the data suggest that discounting slows down learning; $\gamma = 1$ features the quickest learning rate. This might be due to the discounted task being somewhat more difficult than the undiscounted task: since delayed rewards contribute less to the

objective, the agent needs to find a short path to the goal state. In the undiscounted setting, any path safely circumventing the holes is already optimal.

Conclusion

This essay describes how machines learn to carry out a task by interacting with the environment. The presented learning algorithms improve either an estimate of a value function, the policy, or both simultaneously.

Although these algorithms have been known for decades, they currently thrive through the support of deep learning. Deep neural networks take on the extraction of relevant features from complicated observations. The ability to learn feature extraction boosted the performance of reinforcement learning agents. Today, a single algorithm is capable to learn tasks as diverse as driving a racing car, escaping a randomly generated maze or playing Atari games – just from raw, visual input [17].

In practice, deep reinforcement learning is not easy to implement. For simple problems, other algorithms such as the cross entropy method might yield comparable results, while being far less complicated [27]. For hard tasks, experimental evidence suggest to choose a policy based algorithm like A3C rather than a value based method like DQN [17].

Despite its huge success, deep reinforcement learning resembles brute force try-and-error much more than abstract reasoning and understanding which characterises human learning. The presented algorithms do not subdivide their task and tackle the subtasks one by one, neither do they transfer previously learned concepts. The huge successes of recent years, including the mastership of Go, would not have been possible without the tremendous computing power we have at our disposal today.

Acknowledgements

I am grateful to Sergio Bacallado, Baptiste Barreau, Quentin Le Gall, Heino Möller, Marius Neuss and Dario Stein for inspiring discussions and valuable comments. Furthermore, I want to acknowledge Arthur Guiliani and John Schulman for the excellent programming resources they provide on the internet.

Bibliography

[1] V. Minh et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[2] D. Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[3] V. Mnih et al. Playing atari with deep reinforcement learning. *arXiv*, 2013.

[4] URL `https://github.com/MoritzMoeller/reinforcement_learning.git`.

[5] J. Norris. Optimization and control. Lecture Notes, 2007.

[6] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2005.

[7] L. Bottou. *Neural Networks, Tricks of the Trade, Reloaded*. Springer, 2012.

[8] K. He et al. Deep residual learning for image recognition. *arXiv*, 2015.

[9] J. Wu. Introduction to convolutional neural networks. Lecture Notes, 2016.

[10] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 1989.

[11] H. Lin and M. Tegmark. Why does deep and cheap learning work so well? *arXiv*, 2016.

[12] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 1995.

[13] C. Szepesvari. Algorithms for reinforcement learning. Lecture Notes, 2009.

[14] N. Shimkin. Learning in complex systems. Lecture Notes, 2011.

[15] C Watkins and P Dayan. Q-learning. *Machine Learning*, 1992.

[16] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2016.

[17] V. Minh. Asynchronous methods for deep reinforcement learning. *arXiv*, 2016.

[18] S. Mohamed. Machine learning trick of the day (5): Log derivative trick. Blog, 2015.

[19] R. Williams. Simple statistical gradient-following algorithms for reinforcement learning. *Machine Learning*, 1992.

[20] J. Schulman. Policy gradient methods. Lecture Notes, 2017.

[21] E. Greensmith et al. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 2004.

[22] A. Karpathy. Deep reinforcement learning: Pong from pixels. Blog, 2016.

[23] J. Schulman et al. High-dimensional continuous control using generalized advantage estimation. *ICLR*, 2016.

[24] R. Sutton et al. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, 2000.

[25] URL `https://gym.openai.com`.

[26] URL `http://rl-gym-doc.s3-website-us-west-2.amazonaws.com/mlss/pg-startercode.py`.

[27] J. Schulman. *Optimizing Expectations: From Deep Reinforcement Learning to stochastic computation graphs*. PhD thesis, University of California, Berkeley, 2016.