A Survey on Methods and Applications of Deep Reinforcement Learning

Siyi Li
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
sliay@cse.ust.hk
Supervisor : Dit-Yan Yeung

December 28, 2016

Abstract

While perception tasks such as visual image classification and object detection play an important role in human intelligence, the more sophisticated tasks built upon them that involve decision and planning require an even higher level of intelligence. The past few years have seen major advances in many low-level perceptual supervised learning problems by using deep learning models. For higher-level tasks, however, reinforcement learning offers a more powerful and flexible framework for the general sequential decision making problem. While reinforcement learning has achieved some successes in a variety of domains, their applicability has previously been limited to domains with low-dimensional state spaces. To derive efficient and powerful feature representations of the environment, it is naturally desirable to incorporate deep learning to the reinforcement learning domains, which we call *deep reinforcement learning*. In this survey, we start from research on general reinforcement learning methods. We then review the recent advances in *deep reinforcement learning*, including both the methods and its applications on game playing and robotics control. Finally, we discuss some possible research issues.

Contents

1 Introduction

Deep learning allows computational models with multiple processing layers to automatically learn representations of data with multiple levels of abstraction. These methods are making major advances in solving fundamental problems that have resisted the artificial intelligence community of many years. Deep learning has dramatically improved the state-of-the-art in visual object recognition [21], object detection [7], speech recognition [14] and many other domains such as drug analysis [28] and genomics [25]. The advances on these fundamental tasks have laid the foundation for building higher-level artificial intelligence system.

Ultimately, major progress in real artificial intelligence will come about through systems that combine representation learning with complex reasoning, where the latter one is apparently beyond the capability of current deep learning methods. The restriction comes from the formulation of supervised learning [24]. In supervised learning problems (including classification and regression), a learner's goal is to map observations (typically known as features) to actions which are usually a discrete set of classes or real value. The design and analysis of algorithms to address supervised learning problems rely on training and testing instances as independent and identical distributed random variables. This means that a decision made by the learner will have no impact on future observations. Therefore supervised learning algorithms are built to operate in a world in which every decision has no effect on the future data examples. Further, during a training phase an explicit answer is provided to the learner, so that there is no ambiguity about action choices. However, most real-world tasks involve an sequential interactive component and for some of them it's difficult even to define explicit labeled samples. These are major hurdles that hinder the application of deep learning to more complex tasks.

On the other hand, reinforcement learning (RL) offers a more general and flexible framework for a remarkable variety of problems. RL enables an agent to autonomously discover an optimal behavior through trial-and-error interactions with its environment. Instead of providing explicit labeled samples to a problem as in supervised learning, in RL the designer of a task provides feedback in terms of a scalar object function that measures the one-step performance of the agent. RL defines a more general scope of problems by requiring both interactive, sequential prediction as well as complex reward structures with only scalar feedback. It is this distinction that enables so many relevant real world tasks to be framed in these terms; it is this same distinction that makes the problem both theoretically and computationally hard.

While RL has achieved some successes in a variety of domains, the traditional approaches have been limited to domains in which useful features are handcrafted, or to domains with low-dimensional state spaces. To both derive the powerful feature representation from high-dimensional sensory inputs and embrace the generalization ability on complex tasks of RL framework, it is therefore a natural choice to combine recent deep learning advances with RL. With the tight integration of deep learning and RL, many sophisticated tasks can be trained in an end-to-end manner.

Nevertheless, applying deep learning to RL is also confronted with some unique challenges. First, RL is known to be unstable or even diverge when nonlinear function approximators such as neural networks are used. Traditional approaches find it hard to train such

complex policies. The second challenge is that deep learning requires a usually long, data-hungry training paradigm, which is not always available in certain real-world applications such as robotics. It is nontrivial to design such sample efficient algorithms with reasonable time complexity.

In this survey, we aim to give a comprehensive overview of recent deep reinforcement learning methods. We also present a wide variety of tasks to which deep reinforcement learning has been successfully applied. The rest of the survey is organized as follows: In Section 2, we provide a review of some basic deep learning models. Section 3 covers the main concepts and techniques for RL. These two sections serve as the background for deep reinforcement learning. Section 4 would survey the recent developments in deep reinforcement learning methods and the relevant applications are discussed in Section 5. Finally Section 6 discusses some future research issues and concludes the paper.

2 Deep Learning

When we look at the human brain, we see many levels of processing. It is believed that each level is learning representations at increasing level of abstraction. This observation has inspired the recent **deep learning** trend, which attempts to replicate this kind of architecture in a computer. In this section, we give a brief review of deep learning, starting from the simplest type of neural networks, multilayer perceptrons (MLPs), to several deep learning model variants based on MLP.

2.1 Multilayer Perceptrons

A multilayer perceptron is a feedforward neural network model that maps sets of input data to a set of appropriate outputs. Essentially an MLP consists of a sequences of parametric nonlinear transformations. Consider for example a regression task which maps a vector of M dimension to a vector of D dimension, where the input is denoted as a matrix $\mathbf{X}_0$ (0 means it is the 0-th layer of the MLP). The j-th row of $\mathbf{X}_0$, denotes as $\mathbf{X}_{0,j*}$, is a vector representing one data point. The target is denoted as $\mathbf{Y}$. Then the problem of learning an L-layer MLP can be formulated as the following optimization problem:

$$\begin{aligned} \min_{\{\mathbf{W}_l\},\{\mathbf{b}_l\}} \ & \|\mathbf{X}_L - \mathbf{Y}\|_F^2 + \lambda \sum_l \|\mathbf{W}_l\|_F^2 \\ \text{subject to } \ & \mathbf{X}_l = \sigma(\mathbf{X}_{l-1}\mathbf{W}_l + \mathbf{b}_l), l = 1, \ldots, L-1 \\ & \mathbf{X}_L = \mathbf{X}_{L-1}\mathbf{W}_L + \mathbf{b}_L, \end{aligned}$$

where $||\cdot||_F^2$ denotes the squared Frobenius norm, $\sigma(\cdot)$ is a certain kind of element-wise nonlinear activation function. By imposing the activation function, nonlinear transformations are introduced. Some common activations are the sigmoid function $\frac{1}{1+\exp(-x)}$ and the hyperbolic tangent $\tanh(x)$. Here the parameters $\{\mathbf{W}_l, \mathbf{b}_l\}$ ($l = 1, 2, \ldots, L$) are usually learned using backpropagation and stochastic gradient descent (SGD). The key is to compute the gradients of the object function with respect to $\mathbf{W}_l$ and $\mathbf{b}_l$. Let E denote the object

function value, the gradients can be computed recursively by the chain rule as follows:

$$\begin{aligned}
\frac{\partial E}{\partial \mathbf{X}_L} &= 2(\mathbf{X}_L - \mathbf{Y}) \\
\frac{\partial E}{\partial \mathbf{X}_l} &= (\frac{\partial E}{\partial \mathbf{X}_{l+1}} \circ \mathbf{X}_{l+1} \circ (1 - \mathbf{X}_{l+1}))\mathbf{W}_{l+1}, l = 1, \ldots, L-1 \\
\frac{\partial E}{\partial \mathbf{W}_l} &= \mathbf{X}_{l-1}^T(\frac{\partial E}{\partial \mathbf{X}_l} \circ \mathbf{X}_l \circ (1 - \mathbf{X}_l)), l = 1, \ldots, L \\
\frac{\partial E}{\partial \mathbf{b}_l} &= mean(\frac{\partial E}{\partial \mathbf{X}_l} \circ \mathbf{X}_l \circ (1 - \mathbf{X}_l), 1), l = 1, \ldots, L,
\end{aligned}$$

where we omit the regularization parameters for simplicity and $\circ$ denotes the element-wise product and $mean(\cdot, 1)$ is the matlab operation on matrices. In practice, we only use a small part of data to compute the gradient for each update. This is called stochastic gradient descent.

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) can be viewed as a variant of MLP, which is biologically inspired by cat's visual cortex [16]. The visual cortex contains a complex arrangement of cells. Two basic cell types have been identified: Simple cells respond maximally to specific edge-like patterns within their receptive field. Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern. CNNs model this kind of animal visual perception by two key ingredients: convolution and pooling.

Convolution: CNNs take advantage of the fact that the input consists of images. Unlike a regular MLP, the layers of CNNs have neurons arranges as 3D volumes (width, height, depth). In a convolutional layer, the output is the result of the convolution of the input and a linear filter, followed by some element-wise nonlinear transformation. Let $\mathbf{X}$ denote the input 3D tensor and $\mathbf{H}$ denote the 2D output feature map. We use $\mathbf{X}^k$ to denote the k-th matrix in the tensor $\mathbf{X}$. With the linear filter $\mathbf{W}$ and b, the output feature map can be obtained by 2D convolution operation as follows:

$$\mathbf{H}_{ij} = \sigma(\sum_k (\mathbf{W}^k * \mathbf{X}^k)_{ij} + b),$$

where $*$ denotes the 2D convolution operator. Note that with multiple different filters, we can have multiple feature maps which can then be stacked together. The convolutional layer with 4 input feature maps and 2 output feature maps is shown in Figure 1. Each neuron is connected to only a local region of the input volume and the filter weights is shared across the spatial locations. The local connectivity and parameter sharing scheme greatly improves the scalability of CNNs.

Pooling: It is common to insert a pooling layer in-between successive convolutional layers in CNNs. The intuition behind this is that once a feature has been detected, only its approximate position relative to other features is relevant. The pooling layer operates independently on every depth slice of the input volume and resizes it spatially, keeping only the maximum value in that region. Pooling layers can not only control overfitting

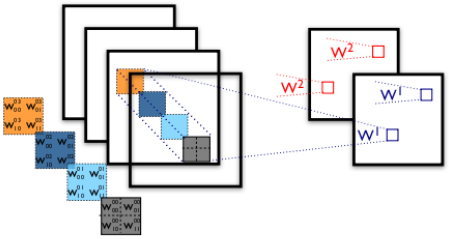

Figure 1: A convolutional layer with 4 input feature maps and 2 output feature maps.

by reducing the amout of parameters but also provides spatial invariance. Note that the pooling layer is optional in CNNs since in some applications local translational invariance and downsampling are not needed.

To build a complete CNN, the input would alternate between several convolutional layers and pooling layers before going into an regular MLP for classification or regression tasks. One recent famous example may be the AlexNet [21], which achieves a huge improvement over traditional methods on the image classification task.

2.3 Recurrent Neural Networks

When we read an article, we understand each word based on the understanding of previous words. This is a recurrent process that needs memory. Traditional feedforward neural networks fail to do this since the temporal sequential information is not modeled explicitly. To overcome this shortcoming, recurrent neural networks (RNNs) use recurrent computation to imitate human memory.

Figure 2(left) shows the vanilla recurrent neural network architecture. The computation of the hidden states $\mathbf{h}_t$ depends on both the current input $\mathbf{x}_t$ and the previous hidden states $\mathbf{h}_{t-1}$. This loop structure enables short-term memory in RNNs. By unrolling the RNNs as in Figure 2(right), the computation of output $\mathbf{o}_t$ can be obtained as follows:

$$\begin{aligned}\mathbf{a}_t &= \mathbf{W}\mathbf{h}_{t-1} + \mathbf{Y}\mathbf{x}_t + \mathbf{b} \\ \mathbf{h}_t &= \sigma(\mathbf{a}_t) \\ \mathbf{o}_t &= \mathbf{V}\mathbf{h}_t + \mathbf{c},\end{aligned}$$

where $\mathbf{Y}$, $\mathbf{W}$ and $\mathbf{V}$ are the weight matrices for input-to-hidden, hidden-to-hidden, hidden-to-output connections, respectively and $\mathbf{b}, \mathbf{c}$ are the corresponding biases. Note that RNNs adopt weight sharing so the above parameters are shared across different instances of the units corresponding to different time steps. The weight sharing scheme is important for processing variable-length sequences.

Similar to feedforward neural networks, the parameters are trained by a generalized backpropagation algorithm called *backpropagation through time (BPTT)* [52].

The problem with the vanilla RNN is that the gradients propagated over many time steps are prone to vanish or explode [3], which makes it very difficult for RNNs to model long-term dependencies in practice. To address this problem, several gated RNN variants are proposed to capture long-range dependencies. Long short-term memory (LSTM) [15] has proven to be powerful and stable in practice.

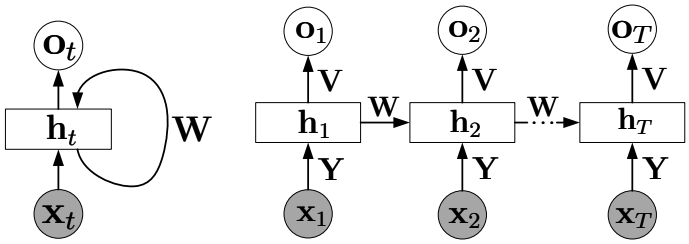

Figure 2: On the left is a recurrent neural network with input $\{\mathbf{x}_t\}_{t=1}^T$, hidden states $\{\mathbf{h}_t\}_{t=1}^T$, and output $\{\mathbf{o}_t\}_{t=1}^T$. On the right is an unrolled computation graph, where each node is associated with one particular time step.

3 Reinforcement Learning

One of the fundamental problems in artificial intelligence and control is that of sequential decision making in stochastic problems. Such problems can be formalized in the Markov decision process (MDP) framework. Briefly, MDP models a system that we are interested in controlling as being in some state at each step in time. A large and diverse set of problems can be modeled using the MDP formalism. Reinforcement learning (RL) offers a powerful set of tools for solving problems posed in MDP formalism and hence provides a more general framework than supervised learning. In this section, we begin by formalizing MDPs and then review some standard algorithms for solving MDPs.

3.1 Markov Decision Processes

Markov decision processes provide a formalism for reasoning about planning and acting in dynamics systems in the face of uncertainty. An MDP is a tuple $(\mathcal{S}, \mathcal{A}, f, R)$ consisting of:

- $\mathcal{S}$: The state space, a set of possible **states** in the world.
- $\mathcal{A}$: The action space, a set of possible **actions** from which we may choose on each time step.
- f: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, \infty)$: The **state transition probability** density function. For each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, this gives the distribution over to which state we will randomly transition if we take action a in state s.
- R: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$: The **reward** function.

In this survey, only stationary MDPs are considered, i.e., the elements of the tuple $(\mathcal{S}, \mathcal{A}, f, R)$ do not change over time.

Events in an MDP proceed as follows. The stochastic process is controlled by the state transition probability density function. Here we adopt the probability density function notation to extend to continuous state and action spaces. Since state space is continuous, it is only possible to define a probability of reaching a certain state *region* (the probability of reaching a particular state is zero.). The probability of reaching a state s_{k+1} in the region $\mathcal{S}_{k+1} \subseteq \mathcal{S}$ from state s_k after applying action a_k is

$$P(s_{k+1} \in \mathcal{S}_{k+1} | s_k, a_k) = \int_{\mathcal{S}_{k+1}} f(s_k, a_k, s') ds'.$$

After each transition to state s_{k+1}, the agent receives an immediate reward

$$r_k = R(s_k, a_k, s_{k+1}),$$

which depends on the previous state, the current state, and the action taken. Note that the reward function R is assumed to be bounded by $R_{\max}$. The action taken in a state s_k is drawn from a stochastic policy π: $\mathcal{S} \times \mathcal{A} \mapsto [0, \infty)$.

The goal of reinforcement learning is to find the policy π which maximizes the expected return J, which corresponds to the expected value of a certain function g of the immediate rewards received following the policy π. This expected return is the cost-to-go function

$$J(\pi) = \mathbb{E}\{g(r_0, r_1, \ldots)|\pi\}.$$

There are different models of optimal behavior [17] which result in different definitions of the expected return. In most cases, the function g is either the discounted sum of rewards or the average reward received, as explained next.

3.1.1 Discounted Reward

In the discounted reward setting, the expected return J is equal to the expected value of the discounted sum of rewards when starting from an initial state $s_0 \in \mathcal{S}$ drawn from an initial state distribution $s_0 \sim d_0(\cdot)$. This is also called the discounted return

$$\begin{aligned} J(\pi) &= \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \middle| d_0, \pi \right\} \\ &= \int_{\mathcal{S}} d^{\pi}(s) \int_{\mathcal{A}} \pi(s,a) \int_{\mathcal{S}} f(s,a,s') R(s,a,s') ds' da ds, \end{aligned} \tag{1}$$

where $d^{\pi}(s) = \sum_{k=0}^{\infty} \gamma^k p(s_k = s|d_0, \pi)$ is the discounted state distribution under the policy π, and $\gamma \in [0, 1)$ denotes the reward discount factor. Here $p(s_k = s)$ denote the probability density function.

Almost all reinforcement learning algorithms involve estimating the cost-to-go function J for a given policy π. This procedure is called *policy evaluation*. The resulting estimate of J is called the *value function* and two definitions exist for it. The state value function

$$V^{\pi}(s) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \middle| s_0 = s, \pi \right\} \tag{2}$$

only depends on the state and assume the policy π is followed starting from this state. The state-action value function

$$Q^{\pi}(s,a) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \middle| s_0 = s, a_0 = a, \pi \right\} \tag{3}$$

depends on both the state and the action, and treats the action a as a free variable. Once the first transition onto the next state has been made, π governs the rest of the action selection. The relationship between these two definitions for the value function is given by

$$V^{\pi}(s) = \mathbb{E}\left\{ Q^{\pi}(s,a) \middle| a \sim \pi(s, \cdot) \right\}$$

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy particular recursive relationships [44]. For the state value function, it is

$$V^\pi(s) = \mathbb{E}\left\{R(s,a,s') + \gamma V^\pi(s')\right\}, \tag{4}$$

with a drawn from the probability density function $\pi(s,\cdot)$ and s' drawn from $f(s,a,\cdot)$. For the state-action value function, the recursive form is

$$Q^\pi(s,a) = \mathbb{E}\left\{R(s,a,s') + \gamma Q^\pi(s',a')\right\}, \tag{5}$$

with s' drawn from the probability density function $f(x,u,\cdot)$ and a' drawn from the distribution $\pi(s',\cdot)$. These recursive forms are called Bellman expectation equations.

Value functions define a partial ordering over policies. There is always at least one policy that is better than or equal to all other policies. This is an optimal policy. Optimality for both the state value function V^π and the state-action value function Q^π is governed by the Bellman optimality equation. Denoting the optimal state value function with $V^*(s)$ and the optimal state-action value function with $Q^*(s,a)$, the corresponding Bellman optimality equations for the discounted reward setting are

$$V^*(s) = \max_a \mathbb{E}\left\{R(s,a,s') + \gamma V^*(s')\right\}, \tag{6a}$$

$$Q^*(s,a) = \mathbb{E}\left\{R(s,a,s') + \max_{a'} \gamma Q^*(s',a')\right\}. \tag{6b}$$

3.1.2 Average Reward

As an alternative to the discounted reward setting, the average reward case are often more suitable in a robotic setting as we do not have to choose a discount factor and we do not have to explicitly consider time in the derivation. In this setting, a distribution d_0 does not need to be chosen, under the assumption that the process is ergodic [44] and thus that J does not depend on the starting state. The cost-to-go function is now

$$\begin{aligned} J(\pi) &= \lim_{n\to\infty} \frac{1}{n}\mathbb{E}\left\{\sum_{k=0}^{n-1} r_k \,\middle|\, \pi\right\} \\ &= \int_{\mathcal{S}} d^\pi(s) \int_{\mathcal{A}} \pi(s,a) \int_{\mathcal{S}} f(s,a,s') R(s,a,s') ds' da ds. \end{aligned} \tag{7}$$

Equation (7) is very similar to (1), except that the definition for the state distribution changed to $d^\pi(s) = \lim_{k\to\infty} p(s_k = s, \pi)$. For a given policy, the state value function $V^\pi(s)$ and state-action value function $Q^\pi(s,a)$ are then defined as

$$V^\pi(s) = \mathbb{E}\left\{\sum_{k=0}^{\infty} (r_k - J(\pi)) \,\middle|\, s_0 = s, \pi\right\},$$

$$Q^\pi(s,a) = \mathbb{E}\left\{\sum_{k=0}^{\infty} (r_k - J(\pi)) \,\middle|\, s_0 = s, a_0 = a, \pi\right\}.$$

The Bellman expectation equations for the average reward — in this case also called the Possion equations [4] — are

$$V^\pi(s) + J(\pi) = \mathbb{E}\left\{R(s,a,s') + V^\pi(s')\right\}, \tag{8a}$$
$$Q^\pi(s,a) + J(\pi) = \mathbb{E}\left\{R(s,a,s') + Q^\pi(s',a')\right\}. \tag{8b}$$

Note that (8a) and (8b) require the value $J(\pi)$, which is unknown and hence needs to be estimated in some way. The Bellman optimality equations, describing an optimum for the average reward case, are

$$V^*(s) + J^* = \max_a \mathbb{E}\left\{R(s,a,s') + V^*(s')\right\}, \tag{9a}$$
$$Q^*(s,a) + J^* = \mathbb{E}\left\{R(s,a,s') + \max_{a'} Q^*(s',a')\right\}, \tag{9b}$$

where J^* is the optimal average reward as defined by (7) when an optimal policy π^* is used.

3.2 MDP Algorithms

Over the course of time, several types of RL algorithms have been introduced, and they can be divided into three groups [20]: critic-only, actor-only, and actor-critic methods, where the words actor and critic are synonyms for the policy and value function, respectively. We now briefly review some standard algorithms from the above three categories. Throughout this survey, we assume the discounted reward setting without special explanation, though many similar derivations exist for the average reward setting.

3.2.1 Critic-Only Methods

Since a value function defines an optimal policy, many algorithms attempt to find either V^* or Q^*. A straightforward way of deriving a policy in critic-only methods is by selecting *greedy actions*: actions for which the value function indicates that the expected return is the highest. Specifically, π^* is given by either of the following equations:

$$\pi^*(s) = \arg\max_a \mathbb{E}\left\{R(s,a,s') + \gamma V^*(s')\right\} \tag{10a}$$
$$\pi^*(s) = \arg\max_a Q^*(s,a). \tag{10b}$$

Note that while knowing Q^* can easily compute the optimal policy, finding optimal policy from V^* is slightly more complicated and also requires knowledge of the state transition density function $f(s,a,s')$ (required to compute the expectation).

A wide variety of value function based RL algorithms have been developed and can be split mainly into three classes: (i) dynamic programming-based optimal control approaches such as policy iteration and value iteration, (ii) rollout-based Monte Carlo methods and (iii) temporal difference (TD) learning methods such as TD(λ), Q-learning [51], and SARSA (State-Action-Reward-State-Action) [38].

Dynamic Programming-Based Methods Dynamic programming-based methods, require a model of the state transition density function $f(s, a, s')$ and the reward function $R(s, a, s')$ to calculate the value function. The model does not necessarily need to be predetermined but can also be estimated from data. Such methods are called *model-based*. Typical methods include policy iteration and value iteration.

Policy iteration alternates between the two phases of policy evaluation and policy improvement. Policy evaluation determines the value function for the current policy. Each state is visited and its value is updated based on equation (4). This procedure is repeated until the value function converges to a fixed point. Policy improvement then greedily selects the best action in every state according to the above estimated value function. The two steps of policy evaluation and policy improvement are iterated until the policy does not change any longer.

Policy iteration only updates the policy once after the policy evaluation step has converged. In contrast, value iteration combines the steps of policy evaluation and policy improvement by directly updating the value function every time a state is updated.

Monte Carlo Methods Monte Carlo methods are typically *model-free* RL algorithms in the sense that they do not need an explicit state transition function. They use sampling to estimate the value functions. By performing rollouts on the current policy, the transitions and rewards are kept track of and are used to form unbiased value function estimates. However, such methods typically suffer the high variance problem.

Temporal Difference Learning Methods TD learning methods generally have a lower variance in the estimates of the value function, compared to Monte Carlo methods. On the other hand, estimates of value function in TD methods are biased. Unlike Monte Carlo methods, they do not have to wait until an estimate of the return is available (i.e., at the end of the episode) to update the value function. Instead, they use temporal errors, which are the differences between the old estimate and a new estimate of the value function. These updates are done iteratively and, in contrast to dynamic programming-based methods, only take into account the sampled successor states rather than the complete distributions over successor states. TD learning methods are also model-free.

For example, given a transition tuple (s, a, s'), the state value function could be updated iteratively by

$$V'(s) = V(s) + \alpha \left(R(s, a, s') + \gamma V(s') - V(s) \right),$$

where $V(s)$ is the old estimate of the value function, $V'(s)$ is the updated one, and α is the learning rate. The equivalent TD learning algorithm for state-action value function is

$$Q'(s, a) = Q(s, a) + \alpha \left(R(s, a, s') + \gamma Q(s', a') - Q(s, a) \right),$$

where $Q(s, a)$ is the old estimate of the state-action value function, $Q'(s, a)$ is the updated one. This algorithm is known as SARSA, which is an on-policy method. On-policy methods collect sample information about the environment using the current policy while off-policy methods learn independent of the employed policy. The off-policy variant is called Q-learning, with the updates

$$Q'(s, a) = Q(s, a) + \alpha \left(R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a) \right). \tag{11}$$

For discrete spaces, the value function can be represented by tables. For large or continuous spaces, function approximation is employed to find a lower dimensional representation that matches the real value function as closely as possible. While selecting greedy actions in discrete action space is rather straightforward, it can be computationally intensive when the action space is continuous. Therefore critic-only methods usually discretize the continuous action space, after which the optimization over the action space becomes a matter of enumeration. Obviously, this approach undermines the ability of using continuous actions.

3.2.2 Actor-Only Methods

Actor-only methods typically work with a parameterized family of policies over which optimization procedures can be used directly. The benefit of a parameterized policy is that a spectrum of continuous actions can be generated. These methods are called policy gradient methods (for example, the SRV [11] and REINFORCE algorithms [53]) and do not use any form of a stored value function. The policy parameterizations are directly optimized by the cost defined in (1) or (7).

A policy gradient method is generally obtained by parameterizing the policy π by the parameter vector $\theta \in \mathbb{R}^p$. The expected returns defined in (1) and (7) are in fact functions of θ. Assuming the parameterization is differentiable with respect to θ, the gradient of the cost function with respect to θ is described by

$$\nabla_\theta J(\theta) = \frac{\partial J}{\partial \pi_\theta} \frac{\partial \pi_\theta}{\partial \theta}. \tag{12}$$

By using standard optimization methods, a locally optimal solution of the cost J can be found by the following iterative updates:

$$\theta_{k+1} = \theta_k + \alpha_{a,k} \nabla_\theta J(\theta_k), \tag{13}$$

where $\alpha_{a,k}$ is the learning rate for the actor. If the gradient estimate is unbiased and learning rates fulfill

$$\sum_{k=0}^{\infty} \alpha_{a,k} = \infty, \qquad \sum_{k=0}^{\infty} \alpha_{a,k}^2 < \infty,$$

then the learning process is guaranteed to converge to at least a local minimum. The main problem in policy gradient methods is obtaining a good estimator of the policy gradient $\nabla_\theta J(\theta)$. The most prominent approaches are finite-difference and likelihood ratio methods [48, 8], the latter more well known as REINFORCE methods in RL.

Finite-Difference Methods Finite-difference methods are among the oldest policy gradient approaches dating back to 1950s. The idea is straightforward to understand. The policy parameterization is varied by small increments $\Delta\theta_i$ and for each policy parameter parameterization $\theta_k + \Delta\theta_i$ rollouts are performed which generate estimates $\hat{J}_i = J(\theta_k + \Delta\theta_i)$ and $\Delta\hat{J}_i \approx J(\theta_k + \Delta\theta_i) - J_{\text{ref}}$ of the expected return. There are different ways of choosing

the reference value J_{ref}, e.g. forward-difference estimators as $J_{ref} = J(\theta_k)$ and central-difference estimators as $J_{\text{ref}} = J(\theta_k - \Delta\theta_i)$. The gradient can now be estimated by solving the following linear regression problem

$$\left[\nabla_\theta J^T, J_{\text{ref}}\right] = \left(\Delta\boldsymbol{\Theta}^T \Delta\boldsymbol{\Theta}\right)^{-1} \Delta\boldsymbol{\Theta}^T \hat{\mathbf{J}},$$

where

$$\Delta\boldsymbol{\Theta} = \begin{bmatrix} \Delta\theta_1, & \cdots, & \Delta\theta_I \\ 1, & \cdots, & 1 \end{bmatrix}^T, \qquad \hat{\mathbf{J}} = \left[\hat{J}_1, \cdots, \hat{J}_I\right],$$

denote the I samples. This approach is very straightforward and even applicable to policies that are not differentiable. However, it is usually considered to be very noisy and inefficient. Also, they do not work for parameter spaces of very high dimensionality (e.g., deep learning models).

Likelihood Ratio Methods Likelihood ratio methods are driven by a different important insight. Assume that trajectories τ are generated from a system by rollouts, i.e., $\tau \sim p_\theta(\tau) = p(\tau|\theta)$ with the return of a trajectory $J^\tau = \sum_{k=0}^{H} \gamma^k r_k$. From the view of the trajectories, the expected return of the policy can be written as an expectation over all possible trajectories $\mathbb{T}$:

$$J(\theta) = \int_{\mathbb{T}} p(\tau|\theta) d\tau.$$

Subsequently, we can rewrite the gradient by

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_{\mathbb{T}} \nabla_\theta p(\tau|\theta) J^\tau d\tau \\ &= \int_{\mathbb{T}} p(\tau|\theta) \nabla_\theta \log p(\tau|\theta) J^\tau d\tau \\ &= \mathbb{E}\left\{\nabla_\theta \log p(\tau|\theta) J^\tau\right\}. \end{aligned}$$

If we have a stochastic policy $\pi_\theta(s, a)$, the derivative $\nabla_\theta \log p(\tau|\theta)$ can be computed without knowledge of the generating distribution $p(\tau|\theta)$ as

$$p(\tau|\theta) = p(s_0) \sum_{k=0}^{H} f(s_k, a_k, s_{k+1}) \pi_\theta(s_k, a_k)$$

implies that

$$\nabla_\theta \log p(\tau|\theta) = \sum_{k=0}^{H} \nabla_\theta \log \pi_\theta(s_k, a_k),$$

i.e., the derivatives with respect to the dynamics system do not have to be computed. Finally, the policy gradient can be estimated as

$$\nabla_\theta J(\theta) = \mathbb{E}\left\{\left(\sum_{k=0}^{H} \nabla_\theta \log \pi_\theta(s_k, a_k)\right) J^\tau\right\}. \tag{14}$$

If we now take into account that rewards at the beginning of a trajectory cannot be caused by actions at the end of a trajectory, we can replace the return of the trajectory J^π by the state-action value function $Q^\pi(s,a)$ and get [33]

$$\nabla_\theta J(\theta) = \mathbb{E}\left\{\sum_{k=0}^{H} \nabla \log \pi_\theta(s_k,a_k) Q^\pi(s_k,a_k)\right\}, \tag{15}$$

which is equivalent to the *policy gradient theorem* [45]. In practice, it is often advisable to subtract a reference J_b, also called the baseline, from the return of trajectory J^π or the state-action value function $Q^\pi(s,a)$ respectively to get better estimates with smaller variance (though still large).

A major drawback of the above actor-only approach is that the estimated gradient may have a large variance [45, 37].

3.2.3 Actor-Critic Methods

Actor-critic methods combine the advantages of actor-only and critic-only methods. While the parameterized actor brings the ability of producing continuous actions without the need for optimization procedures on a value function, the critic supplies the actor with low-variance policy gradient estimates. The role of the critic is to evaluate the current policy prescribed by the actor. In principle, this evaluation can be done by any policy evaluation methods such as TD(λ). The critic approximates and updates the value function using samples. The value function is then used to update the actor's policy parameters in the direction of policy improvement. In actor-critic methods, the policy is not inferred from the value function by using (10b) but generated by the actor instead.

Many actor-critic algorithms rely on the policy gradient theorem, a result obtained simultaneously in [45, 20], proving that an unbiased estimate of the policy gradient (12) can be obtained from experience using a value function approximator satisfying certain properties. In what follows we first introduce the policy gradient theorem, as well as compatible function approximators, and then formulate the standard actor-critic algorithm. Finally, we also briefly describe the actor-critic methods in the natural gradient setting. For a more comprehensive survey on actor-critic methods, reader are referred to [33, 9].

Policy Gradient Theorem Consider the case of an approximated stochastic policy π_θ, but with exact state-action value function Q^π, the policy gradient theorem is as follows.

Theorem 1 *(Policy Gradient): For any MDP, in either the discounted reward or average reward setting, the policy gradient is given by*

$$\nabla_\theta J(\theta) = \int_{\mathcal{S}} d^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(s,a) Q^\pi(s,a) da ds$$

with $d^\pi(s)$ defined for the appropriate reward setting.

The above theorem shows the relationship between the policy gradient $\nabla_\theta J(\theta)$ and the critic function $Q^\pi(s,a)$. For most applications, the state-action space is continuous and

thus infinite, which means that it is essential to approximate the state-action value function. The result in [45, 20] also shows that $Q^\pi(s,a)$ can be approximated with a certain function $h_w : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, parameterized by w, without affecting the unbiasedness of the policy gradient estimate.

Theorem 2 *(Policy Gradient with Function Approximation): If the following two conditions are satisfied:*

1. *Function approximator h_w is compatible to the policy*

$$\nabla_w h_w(s,a) = \nabla_\theta \log \pi_\theta(s,a),$$

2. *Function approximator h_w minimizes the following mean-squared error from the exact state-action value function $Q^\pi(s,a)$*

$$\varepsilon = \int_{\mathcal{S}} d^\pi(s) \int_{\mathcal{A}} \pi_\theta(s,a) \left\{ (Q^\pi(s,a) - h_w(s,a))^2 \right\},$$

where $\pi_\theta(s,a)$ denotes the stochastic policy, parameterized by θ, then

$$\nabla_\theta J(\theta) = \int_{\mathcal{S}} d^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(s,a) h_w(s,a) da ds. \tag{16}$$

As discussed in [45], using the compatible function approximation $h_w = w^T \nabla_\theta \log \pi_\theta(s,a)$ ($\nabla_\theta \log \pi_\theta(s,a)$ are known as *compatible features*) gives

$$\int_{\mathcal{A}} \pi_\theta(s,a) h_w(s,a) da = w^T \nabla_\theta \int_{\mathcal{A}} \pi_\theta(s,a) da = 0.$$

This shows that the expected value of $h_w(s,a)$ under the the policy π_θ is zero for each state. In this sense, it is better to think of h_w as an approximation of the *advantage function* $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. This means that using the above $h_w(s,a)$ results in an approximator that can only represent the relative value of an action a in some state s. Because of this difference, the policy gradient estimate produced by just the compatible approximation will still have a high variance. To lower the variance, extra features have to be added to model the difference between the advantage function $A^\pi(s,a)$ and the state-action value function $Q^\pi(s,a)$, i.e., the value function $V^\pi(s)$. This is often referred to as a reinforcement baseline. The policy gradient theorem actually generalizes to the case where a state-dependent baseline function is taken into account. Equation (16) is now

$$\nabla_\theta J(\theta) = \int_{\mathcal{S}} d^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(s,a) \left[h_w(s,a) + b(s) \right] da ds. \tag{17}$$

Standard Gradient Actor-Critic Algorithms For both reward settings, the value function is parameterized by the parameter vector $w \in \mathbb{R}^q$. This will be denoted with $V_w(s)$ or $Q_w(s,a)$. If the parameterization is linear, the features will be denoted as ϕ, i.e.,

$$V_w(s) = w^T \phi(s) \text{ or } Q_w(s,a) = w^T \phi(s,a). \tag{18}$$

The stochastic policy π is parameterized by $\theta \in \mathbb{R}^p$ and is denoted as $\pi_\theta(s, a)$. If the policy is denoted as $\pi_\theta(s)$, it is deterministic and a direct mapping from states to actions $a = \pi_\theta(s)$.

The critic is typically updated with TD learning methods. Using the function approximation for the critic and a transition sample (s_k, a_k, r_k, s_{k+1}), the TD error is estimated as

$$\delta_k = r_k + \gamma V_{w_k}(s_{k+1}) - V_{w_k}(s_k). \tag{19}$$

The most standard way of updating the critic is to apply gradient descent to minimize the squared TD error

$$w_{k+1} = w_k + \alpha_{c,k}\delta_k \nabla_w V_{w_k}(s_k), \tag{20}$$

where $\alpha_{c,k}$ is the learning rate for the critic. This TD method is also known as TD(0) learning, as no eligibility traces are used. The extension to the use of eligibility traces, resulting in TD(λ) methods and is explained next. Eligibility traces offer a way to assigning credit to states or state-action pairs visited several steps earlier. The eligibility trace vector at time instant k is denoted as $z_k \in \mathbb{R}^q$ and the update equation is

$$z_k = \lambda\gamma z_{k-1} + \nabla_w V_{w_k}(s_k).$$

It decays with time by a factor $\lambda\gamma$, where $\lambda \in [0, 1)$ is the trace decay rate. By using the eligibility trace vector z_k, the update of the critic (20) becomes

$$w_{k+1} = w_k + \alpha_{c,k}\delta_k z_k.$$

The state-action value function $Q_w(s, a)$ can be updated in a similar manner.

The actor is updated with the estimated policy gradient as in (13). Based on the policy gradient theorem, to obtain an unbiased policy gradient estimate of $\nabla_\theta J(\theta_k)$, we actually need to learn an compatible function approximator $h_w(s, a)$ and the parameter w must be determined by the linear regression problem that estimates $Q^\pi(s, a)$. In practice, the latter condition is usually relaxed and TD learning method are applied to learn the value function more efficiently.

For example, a TD(0) actor-critic algorithm can be described as follows:

$$\delta_k = r_k + \gamma V_{w_k}(s_{k+1}) - V_{w_k}(s_k), \tag{21a}$$
$$w_{k+1} = w_k + \alpha_{c,k}\delta_k \nabla_w V_{w_k}(s_k), \tag{21b}$$
$$\theta_{k+1} = \theta_k + \alpha_{a,k}\delta_k \nabla_\theta \log \pi_\theta(s_k, a_k). \tag{21c}$$

This approach only requires one set of critic parameters V_w. Alternatively, we can also estimate both the state value function $V^\pi(s)$ and the state-action value function $Q^\pi(s, a)$ by using two function approximators. While each value function is updated by TD learning, the actor can be updated with the policy gradient estimate given by the advantage function

$$A(s_k, a_k) = Q_{w_Q}(s_k, a_k) - V_{w_v}(s_k),$$
$$\theta_{k+1} = \theta_k + \alpha_{a,k} A(s_k, a_k) \nabla_\theta \log \pi_\theta(s_k, a_k).$$

Natural Gradient Actor-Critic Algorithms One of the main reasons for using policy gradient methods is that we intend to make a small change $\Delta\theta$ to the policy π_θ while improving the policy. Standard gradients methods use the Euclidean metric of $\sqrt{\Delta\theta^T\Delta\theta}$, then the gradient is different for every parameterization θ of the policy π_θ, even if these parameterizations are related to each other by a linear transformation [18]. The performance of standard gradients methods thus rely heavily on the choice of a coordinate system over which the cost function is defined. This problem poses the question whether we can achieve a covariant gradient descent, i.e., gradient descent with respect to an invariant measure of the closeness between the current policy and the updated policy based upon the distribution of the paths generated by each of these [33]. In statistics, a variety of distance measures for the closeness of two distributions (e.g., $p(\tau|\theta)$ nad $p(\tau|\theta+\Delta\theta)$) have been suggested, e.g., the Kullback-Leibler (KL) divergence $D_{KL}(p(\tau|\theta)||p(\tau|\theta+\Delta\theta)$. The KL divergence can be approximated by the second-order Taylor expansion, i,e, by

$$D_{KL}(p(\tau|\theta)||p(\tau|\theta+\Delta\theta)) \approx \frac{1}{2}\Delta\theta^T\mathbf{F}_\theta\Delta\theta,$$

where

$$\begin{aligned}\mathbf{F}_\theta &= \int_{\mathbb{T}} p(\tau|\theta)\nabla_\theta \log p(\tau|\theta)\nabla_\theta \log p(\tau|\theta)^T d\tau \\ &= \mathbb{E}\left\{\nabla_\theta \log p(\tau|\theta)\nabla_\theta \log p(\tau|\theta)^T\right\}\end{aligned} \tag{22}$$

is known as the Fisher information matrix. Suppose we fix the amount of change in our policy using the step-size ϵ, we then have a restricted step-size gradient descent problem [6]. The optimization problem is formulated as follows:

$$\begin{aligned}&\max_{\Delta\theta}\ J(\theta+\Delta\theta) \approx J(\theta) + \Delta\theta^T\nabla_\theta J(\theta) \\ &\text{subject to}\ \ \epsilon = D_{KL}(p(\tau|\theta)||p(\tau|\theta+\Delta\theta)) \approx \frac{1}{2}\Delta\theta^T\mathbf{F}_\theta\Delta\theta\end{aligned}$$

The solution is given by

$$\Delta\theta = \alpha_n\mathbf{F}_\theta^{-1}\nabla_\theta J(\theta), \tag{23}$$

where $\alpha_n = \left[\epsilon(\nabla_\theta J(\theta)^T\mathbf{F}_\theta^{-1}\nabla_\theta J(\theta))^{-1}\right]^{1/2}$ [35]. The direction $\Delta\theta$ is called the natural gradient $\widetilde{\nabla}_\theta J(\theta) = \Delta\theta/\alpha_n$. It is not necessary to use the learning rate α_n as it does not affect the gradient direction.

The Fisher information matrix of paths can be estimated as in [1, 34]

$$\mathbf{F}_\theta = \int_{\mathcal{S}} d^\pi(s)\int_{\mathcal{A}} \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a)^T dads. \tag{24}$$

If a compatible function $h_w = w^T\nabla_\theta \log \pi_\theta(s,a)$ is used to approximate the value function, then the natural policy gradient is in fact the compatible feature parameter w, i.e.,

$$\widetilde{\nabla}_\theta J(\theta) = w. \tag{25}$$

Consequently, we can use a natural gradient without explicitly calculating the Fisher information matrix.

The strongest theoretical advantage of natural policy gradient methods is that the performance no longer depends on the parameterization of the policy and it is therefore safe to use for arbitrary policies. In practice, the learning process converges significantly faster in most practical cases and requires less manual parameter tuning of the learning algorithm.

4 Deep Reinforcement Learning Methods

With the background on deep learning and RL, we are now ready to introduce some concrete examples of deep reinforcement learning. To bring the success of deep learning to the RL domains, one question is to which part we can apply deep learning in the RL formulation. From the formulation of MDP in Section 3.1, it seems natural to use deep neural networks as function approximators for either the value function or the policy, which can be thought as value-based and policy-based approaches, respectively. In this section, we will introduce both the value-based and policy-based deep reinforcement learning methods. Some recent attempts to combine deep learning with model-based RL methods are also discussed.

4.1 Value-based Deep Reinforcement Learning

Value-based approaches, such as Q-learning, use a state-action value function and no explicit function for the policy. It is usually adopted in discrete action spaces since the equation (10b) can be then solved by simple enumeration. These methods learn the optimal value function by a function approximator. While it is straightforward to use a deep neural network as a function approximator, in practice naive Q-learning often oscillates or even diverges with neural networks. Much efforts have been devoted to address these instability issues in RL when a nonlinear function approximator is used.

4.1.1 Deep Q-Networks

RL is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the state-action function $Q(s, a)$. This instability has several causes: the correlations in the sequence of observations, the fact that small updates to $Q(s, a)$ may significantly change the policy and therefore change the data distribution, and the correlations between the current values $Q(s, a)$ and the target values $r + \gamma \max_{a'} Q(s', a')$. To address these issues, a novel variant of Q-learning, called deep Q-network (DQN) is proposed in [30]. DQN uses two key ideas. First, a biologically inspired mechanism termed experience replay is used to randomize the data, thereby removing the correlations in the observation sequence and smoothing over changes in the data distribution. Second, DQN use an iterative update that adjusts the state-action value $Q(s, a)$ towards target values that are only periodically updated, thereby reducing correlations with the target.

Model Architecture DQN parameterizes an approximate Q-value function $Q_{w_i}(s, a)$ using the deep convolutional neural network, where w_i are the parameters of the Q-network

at iteration i. There are several possible ways of parameterizing the Q-value function using a neural network. Previous approaches [36, 23] use both the states and actions as inputs to the neural network. The main drawback of this type of architecture is that a separate forward pass is required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions. DQN instead use an architecture in which there is a separate output unit for each action and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual actions for the input state. The main advantage of the proposed architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network.

Experience Replay DQN uses a technique known as experience replay in which the agent's experiences at each time step, $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$, are stored in a data set $D_t = \{e_1, e_2, \cdots, e_t\}$, pooled over many episodes (where the end of en episode occurs when a terminal state is reached) into a replay memory. During learning, Q-learning updates are then applied on samples (or minibatches) of experience $(s, a, r, s') \sim U(D)$, drawn uniformly at random from the replay memory.

Target Network A Q-network can be trained by adjusting the parameters w_i at iteration i to reduce the mean-squared error in the Bellman equation. Here introduces the other modification in DQN to standard online Q-learning, which is to use a separate network for generating the target values in the Q-learning update. Specifically, the Q-learning update at iteration i uses the following loss function:

$$L_i(w_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)} \left\{ \left(r + \gamma \max_{a'} Q_{\hat{w}_i}(s', a') - Q_{w_i}(s, a) \right)^2 \right\}, \tag{26}$$

in which w_i are the parameters of the Q-network at iteration i and $\hat{w}_i$ are the target network parameters used to compute the target at iteration i. The target network parameters $\hat{w}_i$ are only updated with the Q-network parameters (w_i) every C steps and are held fixed between individual updates. This modification makes the algorithm more stable compared to standard online Q-learning. Generating the targets using an older set of parameters adds a delay between the time an update to Q-network is made and the time the update affects the target values , making divergence or oscillations much more unlikely.

DQN is able to learn value functions using large, nonlinear function approximators in a stable and robust way. In the following text, we provide several extensions of DQN from different perspectives.

4.1.2 Double Deep Q-Networks

The Q-learning algorithm is known to overestimate state-action values under certain conditions. Formally, the max operator in both standard Q-learning and DQN, i.e., in equation (11) and (26), uses the same values both to select and evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, the selection can be decoupled from the evaluation.

With this motivation, an improved version of DQN, called Double DQN (DDQN) algorithm is proposed in [47]. DDQN proposes to evaluate the greedy policy according to the online network, but to use the target network to estimate its value. Its update is the same as for DQN, but with the following loss function,

$$L_i(w_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)} \left\{ \left(r + \gamma Q_{\hat{w}_i}(s', \arg\max_{a'} Q_{w_i}(s', a')) - Q_{w_i}(s, a) \right)^2 \right\}. \quad (27)$$

The update to the target network stays unchanged from DQN, and remains a periodic copy of the online network.

DDQN gets most of the benefit of Double Q-learning [12], while keeping the rest of the DQN algorithm intact for a fair comparison, and with minimal computational overhead, since no additional parameters are required.

4.1.3 Prioritized Experience Replay

Experience replay lets online RL agents remember and reuse experiences from the past. It has been shown in DQN to greatly stabilize the training of a value function. However, the uniform experience replay in DQN simply replays transitions at the same frequency that they were originally experienced, regardless of their significance. It is therefore desirable to investigate how prioritizing which transitions are replayed can make experience replay more efficient and effective than if all transitions are replayed uniformly. It is well-known that planning algorithms such as value iteration can be made more efficient by prioritizing updates in an appropriate order. Prioritized sweeping [31] selects which state to update next based on some priority measures. Motivated by the idea of prioritized sweeping, Schaul *et al.* [39] propose the prioritized experience replay. The key idea is to more frequently replay transitions with high expected learning process, as measured by the magnitude of their TD errors.

Prioritizing With TD-Error The key component of prioritized replay is the criterion by which the importance of each transition is measured. A reasonable proxy is the magnitude of a transition's TD error δ, which indicates how far the value is from its next step bootstrap estimate. And this is particularly suitable for Q-learning methods that already compute the TD-error.

In practice, new transitions arrive without a known TD-error and they are put at maximum priority in order to guarantee that all experiences are seen at least once. When a transition is replayed from the memory, the TD-error of this transition sample is then updated. A naive sampling strategy is to use greedy TD-error prioritization, which always choose the transition with the maximum magnitude of TD-error. However, this approach suffers some problems, which are explained below.

Stochastic Prioritization Greedy TD-error prioritization has several issues. Transitions that have a low TD error on first visit may not be replayed for a long time. Further, it is sensitive to reward noises. Finally, greedy prioritization focuses on a small subset of the experience and this lack of diversity makes the system prone to overfitting.

To overcome these issues, a stochastic sampling method that interpolates between pure greedy prioritization and uniform random sampling is adopted. The probability of sampling transition i is calculated as

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha},$$

where α determines how much prioritization is used ($\alpha = 0$ corresponds to the uniform case) and $p_i > 0$ is the priority of transition i. Two prioritization variants are considered, i.e., proportional prioritization where $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero, and rank-based prioritization where $p_i = \frac{1}{\text{rank}(i)}$, where rank$(i)$ is the rank of transition i when the replay memory is sorted according to $|\delta_i|$.

Another issue is that prioritized replay introduces bias because it changes the data distribution in an uncontrolled fashion. Fortunately the bias can be corrected by using importance-sampling (IS) weights

$$m_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta,$$

which fully compensate for the non-uniform probabilities $P(i)$ if $\beta = 1$. These weights can be folded into the Q-learning update by using $m_i\delta_i$ instead of δ_i.

Prioritized replay is shown to both significantly speed up the learning process and to achieve better final policy quality.

4.1.4 Dueling Network Architectures

DQN and its several variants all use the same standard neural network architecture to represent the state-action value function. To learn the value function more efficiently, [50] proposes the *dueling architecture* to explicitly separate the representation of state value functions $V(s)$ and state-action advantages $A(s, a)$. The dueling architecture consists of two streams that represent the value and advantage functions, while sharing a common convolutional feature learning module. The two streams are then combined via a special aggregating layer to produce an estimate of the state-action value function $Q(s, a)$. This dueling network can be understood as a single Q-network with two streams that replaces the popular single-stream Q-network in existing algorithms such DQN. The dueling network automatically produces separate estimates of the state value function and advantage function, without any extra supervision.

Specifically, consider a dueling network where one stream of fully-connected layers outputs a scalar $V(s; \beta, w_1)$, and the other stream outputs a vector $A(s, a; \beta, w_2)$. Here, β denotes the parameters of the convolutional layers, while w_1 and w_2 are the parameters of the two streams of fully-connected layers. Using the definition of advantage, a naive aggregating module can be constructed as follows:

$$Q(s, a; \beta, w_1, w_2) = V(s; \beta, w_1) + A(s, a; \beta, w_2).$$

However, the above equation is unidentifiable in the sense that given Q we cannot recover V and A uniquely. To address this issue of identifiability, the advantage function estimator

can be forced to have zero value at the chosen action. That is

$$Q(s,a;\beta,w_1,w_2) = V(s;\beta,w_1) + \left(A(s,a;\beta,w_2) - \max_{a'} A(s,a';\beta,w_2)\right). \quad (28)$$

Now, for $a^* = \arg\max_{a'} Q(s,a';\beta,w_1,w_2) = \arg\max_{a'} A(s,a';\beta,w_2)$, we have $Q(s,a^*;\beta,w_1,w_2) = V(s;\beta,w_1)$. Hence $V(s;\beta,w_1)$ provides an estimate of the value function while the other stream produces an estimate of the advantage function. An alternative module can also replace the max operator with an average:

$$Q(s,a;\beta,w_1,w_2) = V(s;\beta,w_1) + \left(A(s,a;\beta,w_2) - \frac{1}{|A|}\sum_{a'} A(s,a';\beta,w_2)\right). \quad (29)$$

Both the above equations can be viewed and implemented as part of the network and existing DQN algorithms can be recycled to train the dueling architecture. The advantage of the dueling architecture lies partly in its ability to learn the state value function efficiently. With every update of the Q-values, the value stream V is updated — this contrasts with the updates in a single-stream architecture where only the value for one of the actions is updated. In practice this phenomenon is more significantly observed especially when the number of actions is large.

4.1.5 Deep Recurrent Q-Network

DQNs are limited in the sense that they learn a mapping from a limited number of past states. Instead of a MDP, many real-world tasks often feature incomplete and noisy state information and become a Partially-Observable Markov Decision Process (POMDP). The Deep Recurrent Q-Network (DRQN) [13] is introduced to better deal with POMDPs by leveraging advances in RNNs.

Partial Observability In real world environments it's rare that the full state of the environment can be provided to the agent. A POMDP better captures the dynamics of many real world environments by explicitly assuming that the agent only gets partial glimpses of the underlying system state. Formally the agent now receives an observation $o \in \Omega$ instead of a state. This observation is generated from the underlying system state according to the probability distribution $o \sim \mathcal{O}(s)$. Vanilla DQN has no explicit mechanisms for deciphering the underlying state of the POMDP and simply uses a fixed number of stacked history observations as the state depictions.

DRQN Architecture The architecture of DRQN replaces DQN's first fully connected layer with a LSTM. For input, the recurrent network now takes a observation for a single time step instead of stacked history observations required by DQN. DRQN shares the same convolutional layers with DQN while the convolutional outputs are fed to the fully connected LSTM layer. Finally, a fully connected linear layer outputs the state-action value for each possible action.

The added recurrency in DRQN makes it successfully integrates information through time by seeing only a single frame at each time step and replicates DQN's performance on partially observed environments.

4.1.6 Asynchronous Q-learning Variations

Deep RL algorithms based on experience replay have achieved unprecedented success in a variety of domains. However, experience replay has several drawbacks: it uses more memory and computation per real interaction, and it requires off-policy learning algorithms. Motivated by parallel learning, a simple and lightweight framework for deep RL that uses asynchronous gradient descent is proposed in [29]. This parallelism enables a much larger spectrum of on-policy RL algorithms, such as SARSA, n-step methods, as well as off-policy RL algorithms such as Q-learning, to be applied robustly and effectively using deep neural networks.

Asynchronous Framework The aim in designing such a framework is to train deep neural network policies reliably. They use two main ideas to make the modified algorithms practical. First, they use asynchronous actor-learners, which are implemented by multiple CPU threads on a single machine. Keeping the learners on a single machine removes the communication costs of sending gradients and parameters. Second, multiple actor-learners running in parallel are likely to be exploring different parts of the environment. Moreover, different exploration strategies are used for different actor-learners to maximize this diversity. Therefore, the overall changes being made to the parameters by multiple actor-learners are likely to be less correlated. The parallel actors can perform the stabilizing role undertaken by experience replay in the DQN training algorithm.

Asynchronous One-Step Q-learning Similar to DQN, a target network is used. Each thread interacts with its own environment copy and at each step computes a gradient of the Q-learning loss. Gradients are accumulated over multiple time steps, which is similar to using minibatch. Additionally, each thread has a different exploration strategy (different ϵ in ϵ-greedy exploration).

Asynchronous One-Step SARSA The only difference with the Q-learning method is that it uses a different target value for $Q(s, a)$. The target value used by one-step SARSA is $r + \gamma Q(s', a')$, where a' is the action taken in state s'.

Asynchronous N-Step Q-learning In n-step Q-learning, $Q(s, a)$ is updated toward the n-step return defined as $r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \max_a \gamma^n Q(s_{t+n}, a)$. This results in a single reward r directly affecting the values of n preceding state-action pairs. This makes the process of propagating rewards to relevant state-action pairs potentially much more efficient.

4.1.7 Discussion

DQN is the first deep reinforcement learning method to bring the success of deep learning to the reinforcement learning fields. The innovations of experience replay and target network are crucial to the stability of training large, nonlinear function approximators (such as deep convolutional neural networks) to represent the state-action value function $Q(s, a)$. By

the combination of deep learning and Q-learning, DQN and its variants allow the agent to learn complex strategies in an end-to-end manner and bridge the divide between high-dimensional sensory inputs and actions. However, these methods are restricted to discrete action domains due to the limitation of Q-learning.

4.2 Policy-based Deep Reinforcement Learning

Policy-based approaches explicitly maintain a parameterized policy, which is known as the actor, since it is used to select actions. They require minimal computation in order to select actions, especially for continuous action domains. As shown in Section 3.2.2, actor-only methods suffer from the problem that the estimated policy gradient has a large variance. Therefore training deep neural networks to represent the policy is even more challenging for these actor-only methods. In the following text, we discuss some recent advances on combining deep learning with actor-critic approaches.

4.2.1 Asynchronous Advantage Actor-Critic

Consider a stochastic policy $\pi_\theta(s, a)$ parameterized by θ, standard REINFORCE algorithms estimate the policy gradient by $\nabla_\theta \log \pi_\theta(s_t, a_t) Q^\pi(s_t, a_t)$, as shown in equation (15). We can use $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ to get an unbiased estimate of $Q^\pi(s_t, a_t)$ and hence $\nabla_\theta \log \pi_\theta(s_t, a_t) R_t$ is an unbiased estimate of the policy gradient. It is possible to reduce the variance of this estimate while keeping unbiased by subtracting a baseline $b_t(s_t)$, which is a learned function of the state s_t. The resulting gradient is $\nabla_\theta \log \pi_\theta(s_t, a_t)(R_t - b_t(s_t))$. A learned estimate of the value function is commonly used as the baseline $b_t(s_t) \approx V^\pi(s_t)$. When an approximated value function is used as the baseline, the quantity $R_t - b_t$ used to scale the policy gradient is actually the estimate of the advantage function $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$. And this approach can be seen as the advantage actor-critic method.

Following the asynchronous framework described in Section 4.1.6, the asynchronous advantage actor-critic (A3C) algorithm is also proposed in [29]. A3C maintains a policy $\pi_\theta(s, a)$ and an estimate of the value function $V_w(s)$. Like the n-step Q-learning variant, A3C uses the n-step returns to update both the policy and the value function. The update performed by A3C can be seen as $\nabla_{\theta'} \log \pi(s_t, a_t; \theta') A(s_t, a_t; \theta, w)$ where $A(s_t, a_t; \theta, w)$ (θ and θ' are the global shared parameter and thread-specific parameter respectively) is an estimate of the advantage function given by $\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; w) - V(s_t; w)$, where k can vary from state to state and is upper-bounder by t_{max}.

By introducing the parallel actor learners and accumulated gradient updates, A3C improves the training stability a lot, especially if deep neural network function approximators are used.

4.2.2 Trust Region Policy Optimization

Policy-based RL methods typically work by getting a good estimator of the policy gradient, which is essentially the gradient of the expected return with respect to the policy parameters. However most policy gradient methods do not give us any guidance on how to

choose the step size, making these methods oscillate in practice. A recent work [40] proposes and analyzes trust region methods for optimizing stochastic policies, with guaranteed monotonic improvement.

Consider the expected return of two different policies $J(\pi)$ and $J(\tilde{\pi})$, we have the following identity equation [19]:

$$J(\tilde{\pi}) = J(\pi) + \int_{\mathcal{S}} d^{\tilde{\pi}}(s) \int_{\mathcal{A}} \tilde{\pi}(s,a) A^{\pi}(s,a). \tag{30}$$

Equation (30) expresses the expected cost of another policy $\tilde{\pi}$ in terms of the advantage over π, accumulated over time steps. The complex dependency of $d^{\tilde{\pi}}(s)$ on $\tilde{\pi}$ makes (30) difficult to optimize. Instead, the following local approximation to $J(\tilde{\pi})$ is considered:

$$L_{\pi}(\tilde{\pi}) = J(\pi) + \int_{\mathcal{S}} d^{\pi}(s) \int_{\mathcal{A}} \tilde{\pi}(s,a) A^{\pi}(s,a). \tag{31}$$

If the policy parameterization is differentiable, then it is shown in [19] that $L_{\pi}(\tilde{\pi})$ matches $J(\tilde{\pi})$ to first order. One principal theoretical result in [40] is that by optimizing the above approximation function plus a distance measure between two different policies, we are guaranteed to get a sequence of monotonically improving policies. Formally, by solving the following optimization problem:

$$\begin{aligned}
\pi_{i+1} &= \arg\max_{\pi} \left[L_{\pi_i}(\pi) + \left(\frac{2\epsilon\gamma}{(1-\gamma)^2} \right) D_{KL}^{\max}(\pi_i || \pi) \right] \\
\text{where } & \epsilon = \max_s \max_a |A^{\pi}(s,a)|, D_{KL}^{\max}(\pi_i||\pi) = \max_s D_{KL}(\pi_i(s,\cdot)||\pi(s,\cdot)) \\
\text{and } & L_{\pi_i}(\pi) = J(\pi_i) + \int_{\mathcal{S}} d^{\pi_i}(s) \int_{\mathcal{A}} \pi(s,a) A^{\pi_i}(s,a),
\end{aligned}$$

we guarantee that $J(\pi_{i+1}) \geq J(\pi_i)$.

Based on the above theoretical result, the trust region policy optimization (TRPO) is proposed in [40], which is an approximation to the above optimization procedure. Specifically, the following optimization problem is solved to generate a policy update:

$$\begin{aligned}
&\max \ L_{\pi_{\theta_{\text{old}}}}(\pi_{\theta}) \\
&\text{subject to } \bar{D}_{KL}(\pi_{\theta_{\text{old}}} || \pi_{\theta}) \leq \delta,
\end{aligned} \tag{32}$$

where $\pi_{\theta_{\text{old}}}$ denotes the previous policy that we wish to improve upon. The penalty is replaced by a constraint on the KL divergence and a heuristic approximation is used which considers the average KL divergence. Equation (32) can be further formulated as follows by applying Monte Carlo simulation to the objective and constraint:

$$\begin{aligned}
&\max_{\theta} \ \mathbb{E}\left\{ \frac{\pi_{\theta}(s,a)}{\pi_{\theta_{\text{old}}}(s,a)} A_{\theta_{\text{old}}}(s,a) \right\} \\
&\text{subject to } \mathbb{E}\left\{ D_{KL}(\pi_{\theta_{\text{old}}}(s,\cdot) || \pi_{\theta}(s,\cdot)) \right\} \leq \delta.
\end{aligned} \tag{33}$$

The optimization problem defined in equation (32) actually provides a unifying perspective on a number of policy update schemes. The natural policy gradient [18] can be

obtained as a special case of the update in (32) by using a linear approximation to L and a quadratic approximation to the $\bar{D}_{KL}$ constraint, resulting in the following problem:

$$\begin{aligned} &\max_{\theta} \ \left[\nabla_\theta L_{\pi_{\theta_{\text{old}}}}(\pi_\theta)|_{\theta=\theta_{\text{old}}}(\theta-\theta_{\text{old}})\right] \\ &\text{subject to } \frac{1}{2}(\theta_{\text{old}}-\theta)^T\mathbf{F}(\theta_{\text{old}})(\theta_{\text{old}}-\theta)\leq\delta, \\ &\text{where } \mathbf{F}(\theta_{\text{old}})_{ij}=\frac{\partial}{\partial\theta_i}\frac{\partial}{\partial\theta_j}\mathbb{E}\left\{D_{KL}(\pi_{\theta_{\text{old}}}(s,\cdot)||\pi_\theta(s,\cdot))\right\}|_{\theta=\theta_{\text{old}}}. \end{aligned} \tag{34}$$

The update for the standard natural policy gradient is $\theta_{\text{new}}=\theta_{\text{old}}-\lambda\mathbf{F}(\theta_{\text{old}})^{-1}\nabla_\theta L(\theta)|_{\theta=\theta_{\text{old}}}$, where λ is typically treated as an algorithm parameter. The standard policy gradient update can also be obtained by using an ℓ_2 constraint:

$$\begin{aligned} &\max_{\theta} \ \left[\nabla_\theta L_{\pi_{\theta_{\text{old}}}}(\pi_\theta)|_{\theta=\theta_{\text{old}}}(\theta-\theta_{\text{old}})\right] \\ &\text{subject to } \frac{1}{2}||\theta-\theta_{\text{old}}||^2\leq\delta. \end{aligned} \tag{35}$$

In practice, to efficiently solve the trust region policy optimization problem, the same approximation schemes are used as that in the natural policy gradient. One major difference between the natural policy gradient and TRPO is that TRPO enforces the KL divergence constraint at each update, which is implemented by a line search. Without this line search, the algorithm occasionally computes large steps that cause a catastrophic degradation of performance. For deep neural network policies with tens of thousands of parameters, forming and inverting the Fisher information matrix incurs prohibitive computation cost. Therefore a conjugate gradient algorithm is proposed in TRPO to compute the natural gradient direction without explicitly forming the matrix inverse. This makes TRPO practical to deep neural network policies.

4.2.3 Deep Deterministic Policy Gradient

Most existing actor-critic algorithms are on-policy RL methods, in which the critic must learn about and critique whatever policy is currently being followed by the actor. Therefore, on-policy actor-critic methods cannot adopt techniques such as experience replay [30] and learning neural function approximators is difficult and unstable. To bring the success of DQN to continuous action domains, [27] proposes a model-free, off-policy actor-critic algorithm using deep function approximators. The resulting deep deterministic policy gradient (DDPG) algorithm combines deep learning with the deterministic policy gradient (DPG) [42].

Deterministic Policy Gradient We now formally consider a deterministic policy $\pi_\theta(s):\mathcal{S}\mapsto\mathcal{A}$ with parameter vector θ. With a deterministic policy, the expected return $J(\pi)$ is then

$$J(\pi_\theta)=\int_{\mathcal{S}}d^\pi(s)\int_{\mathcal{S}}f(s,\pi(s),s')R(s,\pi(s),s')ds'ds, \tag{36}$$

where the notations follow the definitions in Section 3.1. A deterministic analogue to the policy gradient theorem is provided in [42] as follows:

Theorem 3 *(Deterministic Policy Gradient Theorem) Suppose that the MDP defined in Section 3.1 and the deterministic policy satisfy the following condition:* $f(s,a,s'), \nabla_a f(s,a,s'), \nabla_\theta \pi_\theta(s), R(s,a,s'), \nabla_a R(s,a,s'), d_0(s)$ *are all continuous in all parameters and variables. These imply that* $\nabla_\theta \pi_\theta(s)$ *and* $\nabla_a Q^\pi(s,a)$ *exist and that the deterministic policy gradient exists. Then*

$$\nabla_\theta J(\pi_\theta) = \int_{\mathcal{S}} d^\pi(s) \nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)|_{a=\pi_\theta(s)} ds. \tag{37}$$

It is shown in [42] that the deterministic policy gradient is the limiting case, as the policy variance tends to zero, of the stochastic policy gradient. Therefore the familiar machinery of policy gradients, such as compatible function approximation, natural gradients, actor-critic, is also applicable to deterministic policy gradients.

From a practical view point, there is a crucial difference between the stochastic and deterministic policy gradients. In the stochastic case, the policy gradient integrates over both state and action spaces, whereas in the deterministic case it only integrates over the state space. As a result, the deterministic policy gradient can be estimated much more efficiently than the usual stochastic policy gradient, especially if the action space has many dimensions. Based on the above deterministic policy gradient theorem, a series of off-policy actor-critic algorithms are proposed in [42].

Deep Deterministic Policy Gradient DDPG combines the actor-critic approach developed in DPG with insights from the recent success of DQN. DQN is able to learn value functions using large, nonlinear function approximators in s stable and robust way due to two innovations: 1. the network is trained off-policy with samples from a replay buffer to minimize correlations between samples; 2. the network is trained with a target Q-network to give consistent targets during temporal difference learning. DDPG makes use of the same ideas in the off-policy actor-critic algorithms.

In the DDPG algorithm, both the deterministic actor $\pi_\theta(s)$ and the critic $Q_w(s,a)$ are parameterized by deep neural networks. The critic is learned in the same way as in DQN. The actor is updated by the following deterministic policy gradient

$$\mathbb{E}_{\pi'} \left\{ \nabla_a Q_w(s,a)|_{s=s_t, a=\pi_\theta(s_t)} \nabla_\theta \pi_\theta(s)|_{s=s_t} \right\}, \tag{38}$$

where the expectation is taken with respect to a behavior policy. As in DQN, DDPG also uses a replay buffer. Transitions are sampled from the environment according to the exploration policy and the sample is stored in the replay buffer. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer. DDPG also adopts the idea of target network from DQN but modify it for actor-critic by using "soft" target updates, instead of directly copying weights. More specifically, the weights of the target network are updated as follows:

$$\theta' = \tau\theta + (1-\tau)\theta'$$
$$w' = \tau w + (1-\tau)w'.$$

The exploration policy π' is constructed by adding noise sampled from a noise process $\mathcal{N}$ to the actor policy

$$\pi'(s) = \pi_\theta(s) + \mathcal{N},$$

where $\mathcal{N}$ is chosen as an Ornstein-Uhlenbeck process [46] to generate temporally correlated exploration noise.

4.3 Combining Deep Learning with Model-based Methods

While model-free RL methods provide a simple, direct approach to train policies for complex tasks with minimal feature and policy engineering, the sample complexity of model-free algorithms tends to be high, particularly when using function approximators with large capacity such as deep neural networks. This limits their applicability to real-world physical problems. In this section we discuss the recent attempts to combine deep learning with model-based RL methods.

4.3.1 Q-learning Variations

Model-free RL methods enjoy the benefit of reduced manual engineering and greater generality at the price of high sample complexity. Model-based methods typically improves sample efficiency but limit the policy to only be as good as the learned model. It is therefore desirable to bring the generality of model-free deep RL into real-world domains by reducing their sample complexity. Gu *et al.* [10] proposes two complementary techniques for improving the efficiency of deep RL in continuous action domains, which are presented in detail below.

Continuous Q-learning DQN and its variants are only applicable to discrete action domains since finding the greedy policy is non-trivial. For this reason, continuous action domains are often tackled using actor-critic methods, where a separate parameterized actor policy π is learned in addition to the value function, such as DDPG. To enable Q-learning in continuous action domains, the normalized advantage functions (NAF) is proposed in [10]. The idea behind the normalized advantage functions is to represent the state-action value function $Q(s, a)$ in such a way that its maximum, $\arg\max_a Q(s, a)$, can de determined analytically. Specifically, the advantage function is parameterized as a quadratic function of nonlinear features of the state:

$$\begin{aligned} A(s, a; w^A) &= -\frac{1}{2}(a - \mu(s; w^\mu))^T \mathbf{P}(s; w^P)(a - \mu(s; w^\mu)), \\ Q(s, a; w^A, w^V) &= A(s, a; w^A) + V(s; w^V). \end{aligned}$$

where $\mathbf{P}(s; w^P)$ is a state-dependent, positive-definite matrix, which is parameterized by $\mathbf{P}(s; w^P) = \mathbf{L}(s; w^P)\mathbf{L}(s; w^P)^T$, where $\mathbf{L}(s; w^P)$ is a lower-triangular matrix whose entries come from a linear output layer of a neural network, with the diagonal terms exponentiated. While this representation is more restrictive than a general neural network function, the action that maximizes the Q-function is always given by $\mu(s; w^\mu)$. This representation can be used with deep Q-learning algorithms analogous to DQN, using target networks and a replay buffers as described in DDPG. The parameters $\{w^V, w^\mu, w^P\}$ are parameterized by separate neural networks.

Model-based Imagination Rollouts The data efficiency of NAF can be improved by exploiting learned models. However, a naive idea to use the learned model to generate good behaviors by planning or trajectory optimization often brings very small or even no improvement in practice. The intuition behind this result is that off-policy model-based exploration is too different from the learned policy and Q-learning must consider alternatives in order to ascertain the optimality of a given action.

One way to avoid these problems while still allowing a large amount of on-policy exploration is to generate on-policy trajectories under a learned model. Adding synthetic samples to the replay buffer effectively augments the amount of experience available for Q-learning. The so called imagination rollouts can be viewed as a variant of the Dyna-Q algorithm [43]. It's also related to Monte Carlo tree search methods in the sense that the simulation data are used to train the policy. In practice it is observed that the benefit of model-based learning is derived in the early stages of the learning process, when the policy induced by the Q-function is poor. As learning proceeds, on-policy behavior tends to outperform model-based controllers.

4.3.2 Guided Policy Search

Successful applications of deep RL rely on large amounts of data interactions with the environment. However, real-world robot interaction data is often scarce, especially in safety-critic unstable systems. To address these challenges while still being able to train end-to-end deep neural network policies on robots, Levine *et al.* [26] proposes the guided policy search (GPS) algorithm for sensorimotor control.

Guided policy search converts policy search into supervised learning, by iteratively constructing the training data using an efficient model-free trajectory optimization procedure. GPS consists of two main components. The first is a supervised learning algorithm that trains policy parameterizations π_θ. This is implemented as a stochastic Gaussian policy, with the mean and the covariance both parameterized by deep neural networks. The second component is a trajectory-centric RL algorithm that generates guiding distributions which provide the supervision used to train the policy. These two components form a policy search algorithm that can be used to train complex robotic tasks using only a high-level reward function, as in RL. Supervised learning will not, in general, produce a policy with good long-horizon performance, since a small mistake on the policy will place the system into states that are outside the distribution in the training data, causing compounding errors. To address this issue, GPS alternates between model-based trajectory-centric RL and supervised learning. By explicitly modeling the KL divergence between two kinds of trajectories, the problem can be formalized as a variant of BADMM [49] algorithm for constrained optimization which converges to a locally optimal solution. Readers are referred to [26] for algorithm derivation details.

GPS allows us to train deep neural network policies efficiently in an end-to-end manner. However, the quality of the final policy is restricted by the performance of the model-based method.

5 Applications of Deep Reinforcement Learning

Deep reinforcement learning has been successfully applied to a variety of domains, ranging from simple classic control problem to very complex games such as Go. In this section we review the recent applications of deep reinforcement learning, which can be roughly divided into the following two categories: game playing and robotics application.

5.1 Deep Reinforcement Learning for Playing Games

Deep reinforcement learning is naturally suitable to learn to play various games since the agent can in principle get unlimited training data by consistently interacting with the game environments. Indeed, the success of the recent deep reinforcement learning trend starts with the video game domain.

Bellemare *et al.* [2] introduce the Arcade Learning Environment (ALE): a challenging platform evaluating the development of general, domain-independent AI technology. ALE provides an interface to hundreds of Atari 2600 game environments, each one different, interesting, and designed to be a challenge for human players. Following the development of ALE, Mnih *et al.* [30] apply the DQN algorithm to the challenging domain of Atari 2600 games [2] and are able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games. With the exciting success of DQN, various deep reinforcement learning methods have been proposed and the ALE has become the most popular RL benchmark for evaluating these methods.

In addition to the Atari 2600 games, some other popular games are also explored. For example, Oh *et al.* [32] introduce a new set of RL tasks in *Minecraft*[1] (a flexible 3D world). These tasks not only have the usual RL challenges of partial observability (due to first-person visual observations), delayed rewards, high-dimensional perception, but also require an agent to use active perception. They added memory components to DQN to tackle these tasks. Deep reinforcement learning has also been applied in 3D first-person shooter games [22] and car racing games [27].

Another huge success of deep reinforcement learning is mastering the game of Go [41]. The game of Go has long been viewed as the most challenging of classic games for artificial intelligence due to its enormous search space and the difficulty of evaluating board positions and moves. Silver *et al.* [41] introduce a new approach to computer Go that uses value networks to evaluate board positions and policy networks to select moves. These deep neural networks are trained by a novel combination of supervised training from human expert games, and reinforcement learning from games of self-play. The RL approach is further combined with Monte Carlo search and the resulting AlphaGo defeated the human world champion for the first time.

[1] `https://minecraft.net/`

5.2 Deep Reinforcement Learning for Robotics

RL also offers to robotics a general framework for the design of sophisticated behaviors. Traditional RL methods have been used in simple classic control tasks such as cart-pole balancing, cart-pole swing up, and double inverted pendulum balancing. These relatively low-dimensional tasks provide quick evaluations and comparisons of simple RL algorithms. Recently Duan *et al.* [5] propose a unified benchmark suit of various continuous control tasks, including classic tasks, tasks with very high state and action dimensionality, tasks with partial observations, and tasks with hierarchical structure. The author also compare the difference of various deep reinforcement learning methods,which lays the foundation for applying deep reinforcement learning to more complex robotics applications.

Due to the sample efficiency of model-based methods, combinations of deep learning and model-based RL methods are often preferred in real-world robotics. In [26] GPS is applied to various robotic manipulation tasks. Zhang *et al.* [54] further apply GPS to autonomous aerial vehicles by combination of model predictive control. The main advantage of these methods is that the whole system including perception and control can be trained in an end-to-end manner.

6 Conclusions and Future Research

In this survey, we identified a current trend of combining deep learning and RL and reviewed recent work on deep reinforcement learning. By introducing deep learning into the RL domains, deep reinforcement learning allows us to train policies directly from high-dimensional sensory inputs. To stabilize the training process when deep neural networks are used as the function approximators for value function or the policy, several algorithms have been proposed, ranging from deep Q-network, asynchronous Q-learning to deep deterministic policy gradient. New neural network architectures designed specially for RL are also explored to further boost the performance. With the great representation power of deep neural networks, we can now achieve human level performance on many complex control domains. Many model-free deep RL methods are capable to solve various decision making problem with minimal manual effort.

Current applications of deep RL, especially for those model-free methods, focus on playing games and controlling in a simulator, where an interacting environment is available to generate infinite training data. Directly applying model-free deep RL methods to real-world applications is difficult since real-world interaction is scarce and has many limitations. In the future, we may expect to transfer policies trained in simulated environments to real-world tasks. Besides transferring learned policies from simulated environments to real-world tasks, transfer learning between and multi-task learning of similar tasks are also promising directions to reduce the sample complexity of model-free RL methods. While a large majority of recent deep RL methods are model-based, their sample efficiency tends to be less satisfying. Combing model-free deep RL methods with traditional model-based methods in a smart way is also an interesting direction.

References

[1] J Andrew Bagnell and Jeff Schneider. Covariant policy search. IJCAI, 2003.

[2] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

[3] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[4] Shalabh Bhatnagar, Richard S Sutton, Mohammad Ghavamzadeh, and Mark Lee. Natural actor–critic algorithms. *Automatica*, 45(11):2471–2482, 2009.

[5] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *ICML*, 2016.

[6] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[7] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

[8] Peter W Glynn. Likelilood ratio gradient estimation: an overview. In *Proceedings of the 19th conference on Winter simulation*, pages 366–375. ACM, 1987.

[9] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307, 2012.

[10] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *ICML*, 2016.

[11] Vijaykumar Gullapalli. A stochastic reinforcement learning algorithm for learning real-valued functions. *Neural networks*, 3(6):671–692, 1990.

[12] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[13] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

[14] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243, 1968.

[17] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[18] Sham Kakade. A natural policy gradient. In *NIPS*, volume 14, pages 1531–1538, 2001.

[19] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *ICML*, volume 2, pages 267–274, 2002.

[20] Vijay R Konda and John N Tsitsiklis. On actor-critic algorithms. *SIAM journal on Control and Optimization*, 42(4):1143–1166, 2003.

[21] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[22] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. *arXiv preprint arXiv:1609.05521*, 2016.

[23] Sascha Lange and Martin Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2010.

[24] John Langford and Bianca Zadrozny. Relating reinforcement learning performance to classification performance. In *ICML*, pages 473–480. ACM, 2005.

[25] Michael KK Leung, Hui Yuan Xiong, Leo J Lee, and Brendan J Frey. Deep learning of the tissue-regulated splicing code. *Bioinformatics*, 30(12):i121–i129, 2014.

[26] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[27] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *ICLR*, 2016.

[28] Junshui Ma, Robert P Sheridan, Andy Liaw, George E Dahl, and Vladimir Svetnik. Deep neural nets as a method for quantitative structure–activity relationships. *Journal of Chemical Information and Modeling*, 55(2):263–274, 2015.

[29] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[31] Andrew W Moore and Christopher G Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, 1993.

[32] Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*, 2016.

[33] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[34] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural actor-critic. In *European Conference on Machine Learning*, pages 280–291. Springer, 2005.

[35] Jan Reinhard Peters. *Machine learning of motor skills for robotics*. PhD thesis, University of Southern California, 2007.

[36] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[37] Martin Riedmiller, Jan Peters, and Stefan Schaal. Evaluation of policy gradient methods and variants on the cart-pole benchmark. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 254–261. IEEE, 2007.

[38] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[39] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *ICLR*, 2016.

[40] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, pages 1889–1897, 2015.

[41] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[42] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[43] Richard S Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ICML*, pages 216–224, 1990.

[44] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[45] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[46] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[47] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, 2016.

[48] ALEKSAND. VM, VI Sysoyev, and SHEMENEV. VV. Stochastic optimization. *Engineering Cybernetics*, (5):11, 1968.

[49] Huahua Wang, Arindam Banerjee, and Zhi-Quan Luo. Parallel direction method of multipliers. In *NIPS*, pages 181–189, 2014.

[50] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. In *ICML*, 2016.

[51] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[52] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356, 1988.

[53] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[54] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning deep control policies for autonomous aerial vehicles with MPC-guided policy search. In *ICRA*, 2016.