# Reinforcement Learning: Then and Now

**WIP! Send comments: wcannon@rice.edu**

W. Cannon Lewis II

October 16, 2018

## Contents

# 1 Motivation

If you cruise around the Internet looking for Machine Learning (ML) resources, you'll find a pretty common breakdown of its sub-types: generally, we have Supervised Learning, Unsupervised Learning, and, somewhere in the middle, Reinforcement Learning (RL). Supervised and Unsupervised Learning are defined by the binary of human knowledge invested in the learning process, but RL occupies a gray area. RL can obviate the need for hand-labeling of training data, but properly applying RL often requires human domain expertise. Nonetheless, RL can make it much easier to design learning systems that don't require complex or costly data collection, or a sophisticated model of their operating environment. It is for these reasons that much effort has gone into theoretical and empirical investigation of RL. As we will see, however, much more needs to be done before RL can be safely and confidently applied to real-world systems.

In these notes, we will consider three motivating examples of systems that we might seek to control using RL. Obviously the potential applications of RL are much broader, but these examples will provide us with concrete demonstrations of the successes and failures of state-of-the-art RL theory and practice. In the following, when we refer to a *discrete* space, we typically mean a finite or countably infinite space isomorphic to a subset of the natural numbers $\mathbb{N}$. We refer to such a discrete space as a collection of *states*, such as $x \in \{1, 2, \ldots, n\}$, a grid $\{(x, y) \mid x, y = 1, \ldots, n\}$, or more esoteric arrangements of at most countably many elements. If $S$ is a discrete space, we denote the cardinality of the space by $|S|$ (e.g., $|\{1, \ldots, n\}| = n$). We may also be interested in *continuous* spaces, which we will generally take to mean spaces with uncountably many elements. Most typically, the continuous space under consideration will be $\mathbb{R}^n = \prod_{i=1}^n \mathbb{R}$, i.e. real vectors in $n$ dimensions.

## 1.1 GridWorld

Our first example system is perhaps the most famous system amenable to RL, GridWorld. A particular instance of a GridWorld problem is visualized in Figure **??FIXME:** Add GridWorld figure. You can also play around with simple RL algorithms in GridWorld using Andrej Karpathy's REINFORCEjs[1]. GridWorld consists of a finite grid (i.e. $\{(x, y) | x, y \in \{1, \ldots, n\}\}$), in which an *agent*, represented by a location $(x_a, y_a)$, can move North, South, East, and West. The agent receives real-valued *rewards* while moving around the space, and its goal is to maximize its reward over time. Intuitively, if the agent's movements are deterministic, this is an example of a classical search problem, amenable to algorithms such as A*. However,

[1] `https://cs.stanford.edu/people/karpathy/reinforcejs/`

in RL it is typical to consider actions that only succeed with a certain probability. We will develop more formal tools to discuss these issues in Section 2.

### 1.2 MountainCar with Bang-Bang Control

Our second example system models a car with limited control which is trying to reach the top of a mountain (really, any sufficiently smooth curve). This system, known as MountainCar, is visualized in Figure **??FIXME:** Add MountainCar figure. In general, the car may be navigating any curve, but the initial formulation by Moore [6] uses a quartic curve. There are many ways to formulate this problem, but we will use a formulation with a continuous *state space* and discrete *action space* (see Section 2 for definitions of these terms) to motivate our later discussion. In this formulation, the RL agent is represented by a pair $(s, \dot{s}) \in \mathbb{R}^2$, where $s$ represents the agent's distance from an origin along the $x$-axis and $\dot{s}$ represents the agent's velocity along the $x$-axis. The agent is able to take actions from the set $\{-1, 1\}$ representing *bang-bang* control of the car. This means that the agent is able to choose between accelerating forward at a fixed value and accelerating backward at the same value. In the most general case, the agent only receives reward for reaching the "top" of the mountain, i.e. when $s = x_g$ for some pre-specified $x_g$. However, other formulations include a penalty for switching between the two actions, for reasons that will be further explored in Section 5.

### 1.3 Robotic Reaching

**FIXME:** Add robotic control example (continuous state, continuous action)

## 2 A Formalism in Need of a Problem

In order to say anything theoretical about RL, we need a formal model of the environment (or problem, or system, or task) that we are trying to learn a solution to. For this, RL papers tend to invoke the *Markov Decision Process*, or MDP. It is important to note that there exists a whole literature simply on finding optimal controls in known MDPs, and that the goal of RL is subtly different. In RL, we assume that nothing about the MDP is known, and try to maximize reward despite our lack of knowledge. The reader interested in methods for solving known MDPs should consult Puterman [8] for a thorough exposition.

## 2.1 The MDP Formalism

An MDP is defined as a tuple $(S, A, T, R)$. $S$ is a set known as the *state space*, which may be discrete or continuous. $A$ is another set known as the *action space*, which again may be discrete or continuous. Some authors choose to parameterize the action space by states in the state space and call it the “allowable” action space, but we will show that this is unnecessary**FIXME:** show this. $T : S \times A \times S \to \mathbb{R}$ is the MDP’s *transition function*, which is most typically specified as $\mathbb{P}(s' \mid s, a)$, the probability that an agent ends up in state $s'$ when taking action $a$ in state $s$. Note that $s'$ depends only on $s$ and $a$, rather than the whole history of states and actions taken; this is known as the *Markov property*. Finally, $R : S \times A \to \mathbb{R}$ is the MDP’s *reward function*, which provides the agent with a way of learning optimal behavior in the MDP.

There are many, many variants of this definition, but the above is the simplest and arguably the most amenable to a variety of interpretations. A few things are notable in the definition; chief among them is that the only source of nondeterminism in our formalism is the transition function $T$. Intuitively, this makes some sense; in an unknown environment, we typically do not know with surety what will happen next. Another thing worth noting is that there is a whole subclass of MDPs with reward functions that do not depend on the action taken in a state (i.e., $R(s, a) = R(s)\ \forall a \in A, s \in S$). These kinds of reward functions were studied by Andrew Ng [7], who found that optimal behavior in such MDPs satsifies some interesting properties.

The final piece of our formalism necessary for RL is the notion of a *policy*, which assigns to each action and state a probability of being in that state and taking that action. We typically denote an agent’s policy by $\pi : S \times A \to \mathbb{R}$, so that $\pi(a|s)$ for $s \in S, a \in A$ is the probability of taking action $a$, given that the agent finds itself in state $s$. In RL, our goal is to find a policy $\pi$ that is *optimal* in some sense derived from the rewards that the agent receives. We will examine some different notions of optimality in just a little bit.

All of this formalism sets the stage for RL, but it does not specify exactly how an agent interacts with its environment. In RL, we typically consider systems with discrete time, meaning that an agent’s experience in an environment is indexed by an ordered time set $T \subseteq \mathbb{N}$. While continuous-time MDPs can also be considered, this is largely the domain of classical control theory and we will not cover it here. Since our agent interacts with its environment in discrete time steps, we will discuss an agent’s experience in terms of *rollouts* of the form $(s_1, a_1, r_1, s_2, a_2, r_2, \ldots)$. Though in principle these rollouts can be infinite in extent, in practice they are finite (as even simulated time is finite).

## 2.2 POMDPs and SMDPs

**FIXME:** Add proofs that POMDPs and SMDPs are equivalent to MDPs

## 2.3 What Problem Are We Solving?

**FIXME:** Expand using Bertsekas's notions of optimality

Now that we've developed a formalism for RL, we still need to define what we mean by an optimal policy. Right off the bat, there are a few obvious ways that we could talk about optimality: we could want a policy that gets us a certain amount of reward in the fewest time steps, we could want a policy that gets us the highest total reward in a rollout, or we could be more concerned with getting the highest average reward over time. The notion of optimality that prioritizes getting high reward as quickly as possible is related to the theory of *optimal stopping***FIXME:** Cite, which is further developed elsewhere. Similarly, prioritizing total reward is studied in certain branches of control theory, and is not commonly explicitly optimized in RL (though it is worth noting that for environments with finite time horizons, the total and average reward criteria are equivalent). Rather, RL deals primarily with optimizing the discounted reward accumulated by an agent in a single rollout. This assumption is not often stated in RL papers, but it is central to notions of convergence in classical RL theory.

In order to ensure that the discounted reward optimality criterion is well-defined, many RL formulations include a *discount factor* $\gamma \in [0, 1]$. While many papers list this discount factor as an aspect of the underlying MDP, I prefer to view this as an aspect of the learning problem that RL imposes on top of a fundamental MDP model. The discount factor has no bearing on the dynamics of the system or reward function, and is only used by an RL agent to calculate its own success. Thus the discounted reward criterion is typically formulated in the following way for a rollout of $\{(s_t, a_t, r_t)\}$

$$U_T = \sum_{t=1}^{T} \gamma^t r_t$$

$U_T$ defined in this way is sometimes called a *$T$-step return*. For $\gamma < 1$, this sum is guaranteed to converge even as $T \to \infty$ (since $R$ does not depend on $t$), and so this is the most common criterion used to judge an RL agent's performance. Intuitively, $\gamma < 1$ prioritizes short-term rewards over long-term rewards, which some have argued imitates the way that human set priorities.

Though they are not commonly pursued in existing research, one could imagine many more exotic criteria for optimality of a policy. We highlight this here to emphasize that nothing about this formalism dictates how learning should proceed. In general, there are many goals which we might wish to optimize for when operating in an unknown MDP; we might want to optimize for the accuracy of our predictions of the transition function, or of the reward function, or we might simply want to explore as much of the space as possible while remaining above some reward threshold. Optimizing the discounted reward directly has been the most common criterion historically, but we have seen no indication that this is the only metric worth optimizing with RL. Further investigation is needed into topics such as curiosity-driven RL **FIXME:** Cite which subtly change the objective of learning.

## 3 Classic Algorithms and Convergence Results

We are now armed with a formalism and a metric to optimize, so let us proceed to learn! In all that follows, we will take as given that we are optimizing the discounted reward $U_T$ as defined above, and take a tour of the classical results which hold for this problem. Much of our exposition follows that of Sutton and Barto [12] and Bertsekas [1], though we have translated to the MDP notation established in Section 2.

### 3.1 Value Functions and Iteration

**FIXME:** Expand later

Though we can (and will) talk about optimizing our policy $\pi$ directly, most of the results in RL come instead from developing an intermediate *Value Function*, which can then be optimized and used as a tool to improve our policy. There are two types of value functions: $V$-functions and $Q$-functions, which are defined in the following ways:

$$V_\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right] \tag{1}$$

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right] \tag{2}$$

Intuitively, the $V$-function tells us the expected value of being in state $s$ and executing policy $\pi$ for the rest of time, and the $Q$-function tells us the expected value of being in state $s$, taking action $a$, and following $\pi$ from then on. As you might imagine, these value functions give us a pretty handy way of discussing the optimality of a given policy $\pi$, as well as an indication how we might optimize our policy.

It was shown by Bellman **FIXME:** Cite that the value functions also follow these recursive definitions, which are extremely useful to everything that follows.

$$V_\pi(s) = \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s' \mid s,a) \left[R(s,a) + \gamma V_\pi(s')\right] \tag{3}$$

$$Q_\pi(s,a) = \sum_{s' \in S} P(s' \mid s,a) \left[R(s,a) + \gamma \sum_{a' \in A} \pi(s',a') Q(s',a')\right] \tag{4}$$

$$Q_\pi(s,a) = \sum_{s' \in S} P(s' \mid s,a) \left[R(s,a) + \gamma V_\pi(s')\right] \tag{5}$$

The final equation above comes from simply combining our previous definitions, and hints at how one might derive the $Q$-function from the $V$-function, or vice versa. You might be wondering at this point why we care about these value functions. Well, intuitively the $V$- and $Q$-functions summarize in one number per state (or state-action pair) our knowledge about the future behavior of the MDP that we're trying to learn. As you might imagine, this dramatically enhances the quality of a greedy policy, i.e., the policy that chooses the action with highest $Q$-value (or that is most likely to result in a state with high $V$-value). There are two main ways that one might compute the fixed points of these equations (which can be shown to be unique): *policy iteration* and *value iteration*. Policy iteration consists of computing the fixed point $V_\pi$ for a given policy, then improving the policy by replacing it with the greedy policy according to $V_\pi$. Value iteration instead computes a series of truncated estimations of the $V$-function for the optimal policy, $V^*$, by updating the $V$ function at each iteration with the value resulting from the maximizing action instead of a weighted sum over all possible actions. At any point in this process, the "current" policy is simply the greedy policy with respect to the current estimate of $V^*$. Both of these algorithms can be shown to converge to the true optimal policy, as we might hope.

Unfortunately, while the above algorithms are nice, they require us to know the precise transition function $P$ and reward function $R$. It is worth noting that even when we do know the MDP exactly, it can still be expensive to compute the fixed point of the above equations. How, then, are we to solve the RL problem, where we don't have access to the true underlying MDP?

Two answers come to mind immediately: we could either try to directly estimate the $P$ and $R$ functions, or we could instead directly estimate the $Q-$ or $V-$functions from experience. This first approach is called *model-based* RL, and used to be the dominant form of RL. Of course, it has the usual drawbacks of model-based learning methods; insert your favorite statement of the bias-variance trade-off here. The alternate approach is known as *model-free* RL, and is currently dominant because it is amenable to the same kinds of optimization that are currently popular for deep learning. For the rest of these notes we will primarily focus on model-free RL, though there are obviously subtleties that arise in model-based RL as well. **FIXME:** Add whole other section on model-based RL?

## 3.2 Model-Free RL: SARSA, Q-Learning

So, we have decided to do model-free RL; how are we going to proceed? Note that it's sufficient to just work with the $V-$function, since we can always recover the $Q-$function using the equations above. Thus, in the following, we will work with either the $V-$function or $Q-$function interchangeably, with the understanding that the one can always be substituted for the other. Consider the following way of updating an estimation of $V_\pi$ from experience: given an experienced transition $(s_t, a_t, s_{t+1}, r_t)$, update $V_\pi$ with

$$V(s_t) \leftarrow V(s_t) + \lambda \left[r_t + \gamma V(s_{t+1}) - V(s_t)\right] \tag{6}$$

This kind update rule is known as *temporal difference*, or TD, learning. It can also be formulated with more than one step of experience in the obvious way. Here $\lambda$ is a learning rate parameter. It can be proved that under batch learning assumptions this TD update rule computes a maximum-likelihood estimate of the true value function [12]. One of the most basic RL algorithms, **SARSA**, essentially does policy iteration with the TD update rule. The other basic RL algorithm that you may have heard of is called **Q-Learning**, and applies this rule instead:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \lambda \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right] \tag{7}$$

You can think about the difference between SARSA and Q-Learning as being similar to the difference between policy iteration and value iteration. This distinction is often phrased as the difference between *on-policy* RL (SARSA) and *off-policy* RL (Q-Learning). These terms come from the fact that SARSA tries to estimate the value function for the current policy, whereas Q-Learning tries to directly estimate the value function of the optimal policy. There is also a third way, known as *actor-critic* methods, where a value function estimate and policy are optimized at the same time.

**FIXME:** Say something about $\epsilon$-greedy policy selection and convergence

## 3.3 The Problem with Infinity

**FIXME:** Say something about how convergence rates for classic RL are typically stated in terms of the sizes of the state and action spaces, and depend on visiting all state-action pairs infinitely many times. Neither of these concepts extend easily to countably infinite spaces, and it is generally unclear how they should be expressed in uncountably infinite spaces.

## 4 Connections to Control Theory

**FIXME:** Just write a short section for now, come back to this

## 5 New Frontiers: Continuity, Deep RL, IRL

### 5.1 Can We Always Discretize?

**FIXME:** Talk about curse of dimensionality, better bases for learning, etc.

### 5.2 Case Study: Motor Primitives

**FIXME:** Kober as example of intelligent discretization for robotics.

### 5.3 Deep RL

In response to the issues discussed in the previous sections, it has been suggested that complex function approximators such as deep neural networks may be sufficient to extend RL to large and/or continuous state and action spaces. I will not attempt to give an explanation of deep learning here; the interested reader should check one of the many resources now available such as LeCun et al. [3] or Goodfellow et al. [2]. There has been little theoretical exposition to support these algorithms, but for several tasks (such as video games and simple continuous control tasks) they have been shown empirically to work well. I will suspend any further proselytizing or criticism since this is still an active area of research, and simply present the four most popular methods in Deep RL: DQN, DDPG, TRPO, and PPO.

#### 5.3.1 DQN

The method that launched the Deep RL gold rush that we still find ourselves in was DQN (Deep Q Networks), developed by the savants at Deepmind in Mnih et al. [5]. Their major innovations behind DQN were deciding to store an agent's experience in an *experience replay buffer*, from which transitions could be sampled uniformly in order to train a deep network representing a $Q-$function by stochastic gradient descent, and maintaining a *target network* to improve stability. For a loss function, DQN simply uses the Q-learning update rule described above. The policy represented by this network is represented by simply taking the argmax of its output heads for each state. It cannot be overstated how simple the innovations behind this method are, and yet it has significantly outperformed prior RL methods on video game test environments. Note, however that DQN can only apply to environments with discrete action spaces, since we must be able to take the argmax from the available actions at each step.

### 5.3.2 DDPG

DDPG (Deep Deterministic Policy Gradients) was, once again, developed by Deepmind in Lillicrap et al. [4]. It was designed to extend the innovations behind DQN to problems with continuous action spaces, primarily by transitioning to an actor-critic method with an explicitly parameterized policy. Thus DDPG maintains two deep networks: one for estimating the $Q-$function, and one to represent the current policy. As in DQN, DDPG uses an experience replay buffer and target networks in order to make learning feasible. However, DDPG must use the SARSA update rule, since it is infeasible to find the maximizing action in the Q-learning update rule when the agent's action space is continuous. During training, gradients are propagated through the critic network and through the action network, so that the two are trained simultaneously. DDPG was demonstrated to perform well on simple continuous control tasks, but is also susceptible to a phenomenon known as *catastrophic forgetting* in which the policy's fitness suddenly decreases rapidly. Furthermore, Rajeswaran et al. [9] showed that simple linear or radial basis function policies can perform better and more stably than deep policies on the same tasks. Early results from my own research also suggest that DDPG may not be capable of solving even simple robotic control tasks. Nonetheless, DDPG is a simple and compelling demonstration that continuous RL is possible and worthy of further investigation.

### 5.3.3 TRPO and PPO

For a change, TRPO (Trust Region Policy Optimization) [10] and PPO (Proximal Policy Optimization) [11] were developed at the University of California at Berkeley. These methods are significantly different from DQN and DDPG because they seek to directly optimize an agent's policy, and so do away with estimation of a value function altogether. TRPO and PPO both represent an RL agent's policy directly by a feedforward deep network, then collect some number of rollouts in the agent's environment. They then use these rollouts to compute an improved policy which is constrained by an estimation of the new policy's divergence from the old policy. Intuitively, this constraint prevents chattering or oscillation in the policy optimization process by taking more informed, smaller optimization steps. TRPO uses an estimate of Kullback-Leibler Divergence to constrain optimization, and PPO uses either a "clipping" constraint, an adaptive Kullbak-Leibler Divergence constraint, or some combination thereof to prevent poor policy updates. These can all be viewed as terms added to the loss function which penalize policy updates through some function of $r_t(\theta)$:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{8}$$

In empirical demonstrations, TRPO and PPO appear to be the most stable of state-of-the-art Deep RL methods, and have achieved impressive results on continuous control tasks. Nonetheless, as with DDPG, Rajeswaran et al. [9] showed that TRPO and PPO are beat by simple linear or radial basis function policies. More work is clearly needed to better exploit deep networks for RL.

## 5.4 Inverse RL

**FIXME:** Write something about imitation learning and inverse RL

## 5.5 Meta-learning

**FIXME:** Say something about meta-learning and its equivalence to normal RL

## References

[1] D. P. Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena Scientific Belmont, Massachusetts, 1996.

[2] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[3] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

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

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[6] A. W. Moore. Efficient memory-based learning for robot control. 1990.

[7] A. Y. Ng. *Shaping and policy search in reinforcement learning.* PhD thesis, University of California, Berkeley, 2003.

[8] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2014.

[9] A. Rajeswaran, K. Lowrey, E. V. Todorov, and S. M. Kakade. Towards generalization and simplicity in continuous control. In *Advances in Neural Information Processing Systems*, pages 6550–6561, 2017.

[10] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[12] R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.