

# Introduction to Reinforcement Learning

Joshua Achiam



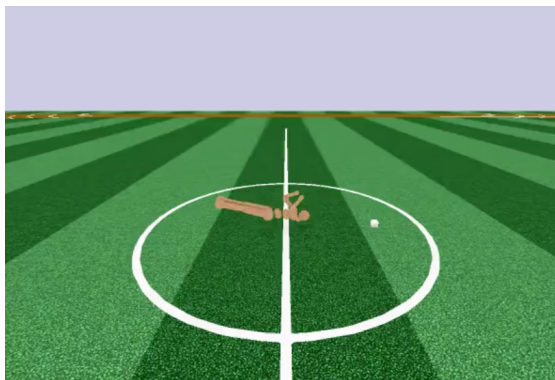
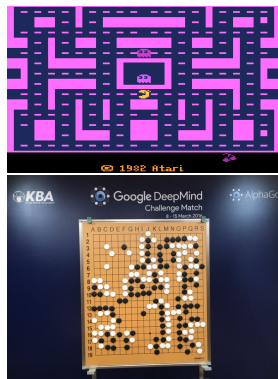
March 3, 2018

# What is RL?

# What can RL do?

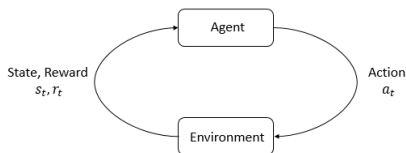
RL can...

- Play video games from raw pixels
- Control robots in simulation and in the real world
- Play Go and Dota 1v1 at superhuman levels



# What is RL?

- An *agent* interacts with an *environment*.



```
obs = env.reset()
done = False
while not (done):
    act = agent.get_action(obs)
    next_obs, reward, done, info = env.step(act)
    obs = next_obs
```

- The goal of the agent is to maximize cumulative reward (called *return*).
- Reinforcement learning (RL) is a field of study for algorithms that do that.

Before we can talk about algorithms, we have to talk about:

- Trajectories
- Return
- Policies
- The RL optimization problem
- Value and Action-Value Functions

**Note:** For this talk, we will talk about all of these things in the context of *deep* RL, where we use neural networks to represent them.

- A **trajectory**  $\tau$  is a sequence of states and actions in an environment:

$$\tau = (s_0, a_0, s_1, a_1, \dots).$$

- The initial state  $s_0$  is sampled from a *start state distribution*  $\mu$ :

$$s_0 \sim \mu(\cdot).$$

- State transitions depend only on the most recent state and action. They could be deterministic:

$$s_{t+1} = f(s_t, a_t),$$

or stochastic:

$$s_{t+1} \sim P(\cdot | s_t, a_t).$$

- A trajectory is sometimes also called an **episode** or **rollout**.

The **reward** function of an environment measures how good state-action pairs are:

$$r_t = R(s_t, a_t).$$

- Example: if you want a robot to run forwards but use minimal energy,  
 $R(s, a) = v - \alpha \|a\|_2^2$ .

The **reward** function of an environment measures how good state-action pairs are:

$$r_t = R(s_t, a_t).$$

- Example: if you want a robot to run forwards but use minimal energy,  
 $R(s, a) = v - \alpha \|a\|_2^2$ .

The **return** of a trajectory is a measure of cumulative reward along it. There are two main ways to compute return:

- Finite horizon undiscounted sum of rewards::

$$R(\tau) = \sum_{t=0}^T r_t$$

- Infinite horizon discounted sum of rewards:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

where  $\gamma \in (0, 1)$ . This makes rewards less valuable if they are further in the future. (Why would we ever want this? Think about cash: it's valuable to have it sooner rather than later!)



A **policy**  $\pi$  is a rule for selecting actions. It can be either

- **stochastic**, which means that it gives a probability distribution over actions, and actions are selected randomly based on that distribution ( $a_t \sim \pi(\cdot|s_t)$ ),
- or **deterministic**, which means that  $\pi$  directly maps to an action ( $a_t = \pi(s_t)$ ).

A **policy**  $\pi$  is a rule for selecting actions. It can be either

- **stochastic**, which means that it gives a probability distribution over actions, and actions are selected randomly based on that distribution ( $a_t \sim \pi(\cdot|s_t)$ ),
- or **deterministic**, which means that  $\pi$  directly maps to an action ( $a_t = \pi(s_t)$ ).

Examples of policies:

- Stochastic policy over discrete actions:

```
obs = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
net = mlp(obs, hidden_dims=(64,64), activation=tf.tanh)
logits = tf.layers.dense(net, units=num_actions, activation=None)
actions = tf.squeeze(tf.multinomial(logits=logits, num_samples=1), axis=1)
```

- Deterministic policy for a vector-valued continuous action:

```
obs = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
net = mlp(obs, hidden_dims=(64,64), activation=tf.tanh)
actions = tf.layers.dense(net, units=act_dim, activation=None)
```

# The Reinforcement Learning Problem

The goal in RL is to learn a policy which maximizes expected return. The optimal policy  $\pi^*$  is:

$$\pi^* = \arg \max_{\pi} \mathbb{E}_{\tau \sim \pi} [R(\tau)],$$

where by  $\tau \sim \pi$ , we mean

$$s_0 \sim \mu(\cdot), \quad a_t \sim \pi(\cdot|s_t), \quad s_{t+1} \sim P(\cdot|s_t, a_t).$$

There are two main approaches for solving this problem:

- policy optimization
- and Q-learning.

# Value Functions and Action-Value Functions

Value functions tell you the expected return after a state or state-action pair.

$$V^\pi(s) = \mathbb{E}_{\tau \sim \pi} [R(\tau) | s_0 = s] \quad \text{Start in } s \text{ and then sample from } \pi$$

$$Q^\pi(s, a) = \mathbb{E}_{\tau \sim \pi} [R(\tau) | s_0 = s, a_0 = a] \quad \text{Start in } s, \text{ take action } a, \text{ then sample from } \pi$$

$$V^*(s) = \max_{\pi} \mathbb{E}_{\tau \sim \pi} [R(\tau) | s_0 = s] \quad \text{Start in } s \text{ and then act optimally}$$

$$Q^*(s, a) = \max_{\pi} \mathbb{E}_{\tau \sim \pi} [R(\tau) | s_0 = s, a_0 = a] \quad \text{Start in } s, \text{ take action } a, \text{ then act optimally}$$

The value functions satisfy recursive **Bellman equations**:

$$V^\pi(s) = \mathbb{E}_{\substack{a \sim \pi \\ s' \sim P}} [r(s, a) + \gamma V^\pi(s')] \quad Q^\pi(s, a) = \mathbb{E}_{s' \sim P} \left[ r(s, a) + \gamma \mathbb{E}_{a' \sim \pi} [Q^\pi(s', a')] \right]$$

$$V^*(s) = \max_a \mathbb{E}_{s' \sim P} [r(s, a) + \gamma V^*(s')] \quad Q^*(s, a) = \mathbb{E}_{s' \sim P} \left[ r(s, a) + \gamma \max_{a'} Q^*(s', a') \right]$$

The optimal  $Q$  function,  $Q^*$ , is especially important because it gives us a policy. In any state  $s$ , the optimal action is

$$a^* = \arg \max_a Q^*(s, a).$$

We can measure how good a  $Q^*$ -approximator,  $Q_\theta$ , is by measuring its **mean-squared Bellman error**:

$$\ell(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(s, a, s', r) \in \mathcal{D}} \left( Q_\theta(s, a) - \left( r + \gamma \max_{a'} Q_\theta(s', a') \right) \right)^2.$$

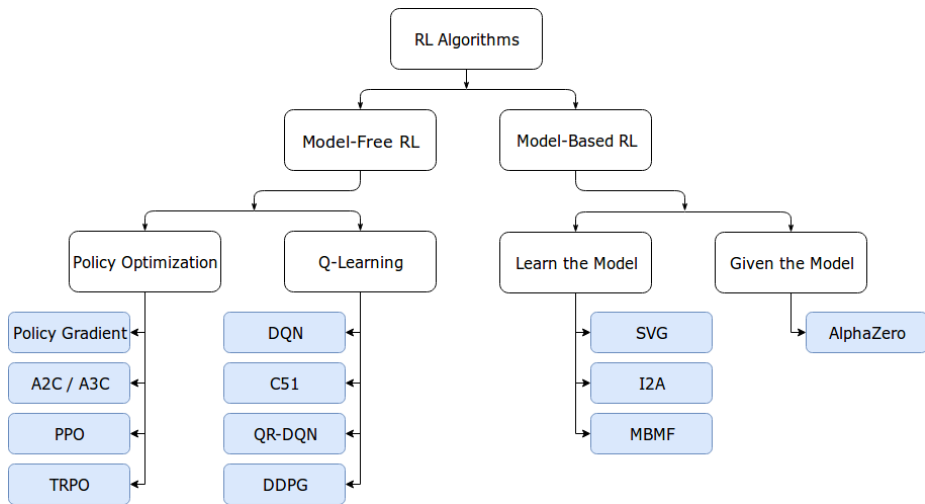
This (roughly) says how well it satisfies the Bellman equation

$$Q^*(s, a) = \mathbb{E}_{s' \sim P} \left[ r(s, a) + \gamma \max_{a'} Q^\pi(s', a') \right]$$

# Deep RL Algorithms

# Deep RL Algorithms

There are many different kinds of RL algorithms! This is a non-exhaustive taxonomy (with specific algorithms in blue):



We will talk about two of them: Policy Gradient and DQN.

Using Model-Free RL Algorithms:

Algorithm	$a$ Discrete	$a$ Continuous
Policy optimization	Yes	Yes
DQN / C51 / QR-DQN	Yes	No
DDPG	No	Yes

Using Model-Based RL Algorithms:

- Learning the model means learning to generate next state and/or reward:

$$\hat{s}_{t+1}, \hat{r}_t = \hat{f}_\phi(s_t, a_t)$$

- Some algorithms may only work with an *exact* model of the environment
  - AlphaZero uses the rules of the game to build its search tree



- An algorithm for training stochastic policies:
  - Run current policy in the environment to collect rollouts
  - Take stochastic gradient ascent on policy performance using the **policy gradient**:

$$\begin{aligned} g &= \nabla_{\theta} \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^T r_t \right] \\ &= \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \left( \sum_{t'=t}^T r_{t'} \right) \right] \\ &\approx \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \left( \sum_{t'=t}^T r_{t'} \right) \end{aligned}$$

- Core idea: push up the probabilities of good actions and push down the probabilities of bad actions
- Definition: sum of rewards after time  $t$  is the *reward-to-go* at time  $t$ :

$$\hat{R}_t = \sum_{t'=t}^T r_{t'}$$

## Make the model, loss function, and optimizer:

```
# make model
with tf.variable_scope('model'):
    obs_ph = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
    net = mlp(obs_ph, hidden_sizes=[hidden_dim]*n_layers)
    logits = tf.layers.dense(net, units=nActs, activation=None)
    actions = tf.squeeze(tf.multinomial(logits=logits,num_samples=1), axis=1)

# make loss
adv_ph = tf.placeholder(shape=(None,), dtype=tf.float32)
act_ph = tf.placeholder(shape=(None,), dtype=tf.int32)
action_one_hots = tf.one_hot(act_ph, nActs)
log_probs = tf.reduce_sum(action_one_hots * tf.nn.log_softmax(logits), axis=1)
loss = -tf.reduce_mean(adv_ph * log_probs)

# make train op
train_op = tf.train.AdamOptimizer(learning_rate=lr).minimize(loss)

sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
```

## Example Implementation (Continued)

### One iteration of training:

```
# train model for one iteration
batch_obs, batch_acts, batch_rtgs, batch_rets, batch_lens = [], [], [], [], []
obs, rew, done, ep_rews = env.reset(), 0, False, []
while True:
    batch_obs.append(obs.copy())
    act = sess.run(actions, {obs_ph: obs.reshape(1,-1)})[0]
    obs, rew, done, _ = env.step(act)
    batch_acts.append(act)
    ep_rews.append(rew)
    if done:
        batch_rets.append(sum(ep_rews))
        batch_lens.append(len(ep_rews))
        batch_rtgs += list(discount_cumsum(ep_rews, gamma))
        obs, rew, done, ep_rews = env.reset(), 0, False, []
        if len(batch_obs) > batch_size:
            break

# normalize advs trick:
batch_advs = np.array(batch_rtgs)
batch_advs = (batch_advs - np.mean(batch_advs)) / (np.std(batch_advs) + 1e-8)
batch_loss, _ = sess.run([loss, train_op], feed_dict={obs_ph: np.array(batch_obs),
                                                       act_ph: np.array(batch_acts),
                                                       adv_ph: batch_advs})
```

- Core idea: learn  $Q^*$  and use it to get the optimal actions
- Way to do it:
  - Collect experience in the environment using a policy which trades off between acting randomly and acting according to current  $Q_\theta$
  - Interleave data collection with updates to  $Q_\theta$  to minimize Bellman error:

$$\min_{\theta} \sum_{(s,a,s',r) \in \mathcal{D}} \left( Q_\theta(s,a) - \left( r + \gamma \max_{a'} Q_\theta(s',a') \right) \right)^2$$

...sort of! This actually won't work!

# Getting Q-Learning to Work (DQN)

## Experience replay:

- Data distribution changes over time: as your  $Q$  function gets better and you *exploit* this, you visit different  $(s, a, s', r)$  transitions than you did earlier
- Stabilize learning by keeping old transitions in a replay buffer, and taking minibatch gradient descent on mix of old and new transitions

# Getting Q-Learning to Work (DQN)

## Experience replay:

- Data distribution changes over time: as your  $Q$  function gets better and you *exploit* this, you visit different  $(s, a, s', r)$  transitions than you did earlier
- Stabilize learning by keeping old transitions in a replay buffer, and taking minibatch gradient descent on mix of old and new transitions

## Target networks:

- Minimizing Bellman error directly is unstable!
- It's *like* regression, but it's not:

$$\min_{\theta} \sum_{(s,a,s',r) \in \mathcal{D}} (Q_{\theta}(s,a) - y(s',r))^2,$$

where the target  $y(s',r)$  is

$$y(s',r) = r + \gamma \max_{a'} Q_{\theta}(s',a').$$

Targets depend on parameters  $\theta$ —so an update to  $Q$  changes the target!

- Stabilize it by *holding the target fixed* for a while: keep a separate target network,  $Q_{\theta_{\text{target}}}$ , and every  $k$  steps update  $\theta_{\text{target}} \leftarrow \theta$

## Algorithm 1 Deep Q-Learning

Randomly generate  $Q$ -function parameters  $\theta$   
 Set target  $Q$ -network parameters  $\theta_{\text{targ}} \leftarrow \theta$   
 Make empty replay buffer  $\mathcal{D}$   
 Receive observation  $s_0$  from environment  
**for**  $t = 0, 1, 2, \dots$  **do**  
     With probability  $\epsilon$ , select random action  $a_t$ ; otherwise select  $a_t = \arg \max_a Q_\theta(s_t, a)$   
     Step environment to get  $s_{t+1}, r_t$  and end-of-episode signal  $d_t$   
     Linearly decay  $\epsilon$  until it reaches final value  $\epsilon_f$   
     Store  $(s_t, a_t, r_t, s_{t+1}, d_t) \rightarrow \mathcal{D}$   
     Sample mini-batch of transitions  $B = \{(s, a, r, s', d)_i\}$  from  $\mathcal{D}$   
     For each transition in  $B$ , compute

$$y = \begin{cases} r & \text{transition is terminal } (d = \text{True}) \\ r + \gamma \max_{a'} Q_{\theta_{\text{targ}}}(s', a') & \text{otherwise} \end{cases}$$

Update  $Q$  by gradient descent on regression loss:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_{(s,a,y) \in B} (Q_\theta(s, a) - y)^2$$

**if**  $t \% t_{\text{update}} = 0$  **then**

    Set  $\theta_{\text{targ}} \leftarrow \theta$

**end if**

**end for**

- A2C / A3C: Mnih et al, 2016 (<https://arxiv.org/abs/1602.01783>)
- PPO: Schulman et al, 2017 (<https://arxiv.org/abs/1707.06347>)
- TRPO: Schulman et al, 2015 (<https://arxiv.org/abs/1502.05477>)
- DQN: Mnih et al, 2013  
(<https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>)
- C51: Bellemare et al, 2017 (<https://arxiv.org/abs/1707.06887>)
- QR-DQN: Dabney et al, 2017 (<https://arxiv.org/abs/1710.10044>)
- DDPG: Lillicrap et al, 2015 (<https://arxiv.org/abs/1509.02971>)
- SVG: Heess et al, 2015 (<https://arxiv.org/abs/1510.09142>)
- I2A: Weber et al, 2017 (<https://arxiv.org/abs/1707.06203>)
- MBMF: Nagabandi et al, 2017 (<https://sites.google.com/view/mbmf>)
- AlphaZero: Silver et al, 2017 (<https://arxiv.org/abs/1712.01815>)