Vishal Maini Follow
Incoming @DeepMindAI. Previously @Upstart, @Yale, @TrueVenturesTEC.
Aug 19, 2017 · 12 min read

Machine Learning for Humans, Part 5: Reinforcement Learning

Exploration and exploitation. Markov decision processes. Q-learning, policy learning, and deep reinforcement learning.

[Update 9/1/17] This series is now available as a full-length e-book! Download here.

"I just ate some chocolate for finishing the last section."

In supervised learning, training data comes with an answer key from some godlike "supervisor". If only life worked that way!

In **reinforcement learning (RL)** there's no answer key, but your reinforcement learning **agent** still has to decide how to act to perform its task. In the absence of existing training data, the agent learns from experience. It collects the training examples ("this action was good, that action was bad") through **trial-and-error** as it attempts its task, with the goal of maximizing long-term **reward**.

In this final section of Machine Learning for Humans, we will explore:

- The **exploration/exploitation** tradeoff
- **Markov Decision Processes (MDPs)**, the classic setting for RL tasks
- **Q-learning, policy learning,** and **deep reinforcement learning**
- and lastly, the **value learning problem**

At the end, as always, we've compiled some favorite resources for further exploration.

Let's put a robot mouse in a maze

The easiest context in which to think about reinforcement learning is in games with a clear objective and a point system.

Say we're playing a game where our mouse 🐭is seeking the ultimate reward of cheese at the end of the maze (🧀+1000 points), or the lesser reward of water along the way (💧+10 points). Meanwhile, robo-mouse wants to avoid locations that deliver an electric shock (⚡ -100 points).

The reward is cheese.

After a bit of **exploration,** the mouse might find the mini-paradise of three water sources clustered near the entrance and spend all its time **exploiting** that discovery by continually racking up the small rewards of these water sources and never going further into the maze to pursue the larger prize.

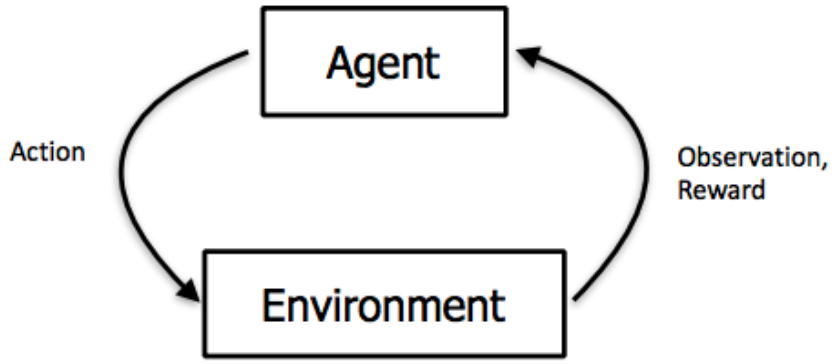
But as you can see, the mouse would then miss out on an even better oasis further in the maze, or the ultimate reward of cheese at the end!

This brings up the **exploration/exploitation** tradeoff. One simple strategy for exploration would be for the mouse to take the best known action most of the time (say, 80% of the time), but occasionally explore a new, randomly selected direction even though it might be walking away from known reward.

This strategy is called the **epsilon-greedy** strategy, where **epsilon** is the percent of the time that the agent takes a randomly selected action rather than taking the action that is most likely to maximize reward given what it knows so far (in this case, 20%). We usually start with a lot of exploration (i.e. a higher value for epsilon). Over time, as the mouse learns more about the maze and which actions yield the most

long-term reward, it would make sense to steadily reduce epsilon to 10% or even lower as it settles into exploiting what it knows.

It's important to keep in mind that the reward is not always immediate: in the robot-mouse example, there might be a long stretch of the maze you have to walk through and several decision points before you reach the cheese.



The agent **observes** the environment, takes an **action** to interact with the environment, and receives positive or negative **reward.** Diagram from Berkeley's CS 294: Deep Reinforcement Learning by John Schulman & Pieter Abbeel

Markov Decision Processes (MDPs)

The mouse's wandering through the maze can be formalized as a **Markov Decision Process**, which is a process that has specified transition probabilities from state to state. We will explain it by referring to our robot-mouse example. MDPs include:

1. **A finite set of states.** These are the possible positions of our mouse within the maze.
2. **A set of actions available in each state.** This is {forward, back} in a corridor and {forward, back, left, right} at a crossroads.
3. **Transitions between states.** For example, if you go left at a crossroads you end up in a new position. These can be a set of probabilities that link to more than one possible state (e.g. when you use an attack in a game of Pokémon you can either miss, inflict some damage, or inflict enough damage to knock out your opponent).
4. **Rewards associated with each transition.** In the robot-mouse example, most of the rewards are 0, but they're positive if you

reach a point that has water or cheese and negative if you reach a point that has an electric shock.

5. **A discount factor γ between 0 and 1.** This quantifies the difference in importance between immediate rewards and future rewards. For example, if γ is .9, and there's a reward of 5 after 3 steps, the present value of that reward is $.9^3*5$.

6. **Memorylessness.** Once the current state is known, the history of the mouse's travels through the maze can be erased because the current Markov state contains all useful information from the history. In other words, "the future is independent of the past given the present".

Now that we know what an MDP is, we can formalize the mouse's objective. We're trying to maximize the sum of rewards in the long term:

$$\sum_{t=0}^{t=\infty} \gamma^t r(x(t), a(t))$$

Let's look at this sum term by term. First of all, we're summing across all time steps t. Let's set γ at 1 for now and forget about it. r(x,a) is a reward function. For state *x* and action *a* (i.e., go left at a crossroads) it gives you the reward associated with taking that action *a* at state *x*. Going back to our equation, we're trying to maximize the sum of future rewards by taking the best action in each state.

Now that we've set up our reinforcement learning problem and formalized the goal, let's explore some possible solutions.

Q-learning: learning the action-value function

Q-learning is a technique that evaluates which action to take based on an **action-value function** that determines the value of being in a certain state and taking a certain action at that state.

We have a function Q that takes as an input one state and one action and returns the **expected reward** of that action (and all subsequent actions) at that state. Before we explore the environment, Q gives the same (arbitrary) fixed value. But then, as we explore the environment

more, Q gives us a better and better approximation of the value of an action *a* at a state *s*. We update our function Q as we go.

This equation from the Wikipedia page on Q-learning explains it all very nicely. It shows how we update the value of Q based on the reward we get from our environment:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Let's ignore the discount factor γ by setting it to 1 again. First, keep in mind that Q is supposed to show you the full sum of rewards from choosing action Q and all the optimal actions afterward.

Now let's go through the equation from left to right. When we take action at in state st, we update our value of Q(st,at) by adding a term to it. This term contains:

- Learning rate **alpha**: this is how aggressive we want to be when updating our value. When alpha is close to 0, we're not updating very aggressively. When alpha is close to 1, we're simply replacing the old value with the updated value.
- The **reward** is the reward we got by taking action at at state st. So we're adding this reward to our old estimate.
- We're also adding the **estimated future reward**, which is the maximum achievable reward Q for all available actions at xt+1.
- Finally, we subtract the old value of Q to make sure that we're only incrementing or decrementing by the difference in the estimate (multiplied by alpha of course).

Now that we have a value estimate for each state-action pair, we can select which action to take according to our **action-selection strategy** (we don't necessarily just choose the action that leads to the most expected reward every time, e.g. with an epsilon-greedy exploration strategy we'd take a random action some percentage of the time).

In the robot mouse example, we can use Q-learning to figure out the value of each position in the maze and the value of the actions {forward, backward, left, right} at each position. Then we can use our action-selection strategy to choose what the mouse actually does at each time step.

Policy learning: a map from state to action

In the Q-learning approach, we learned a **value function** that estimated the value of each state-action pair.

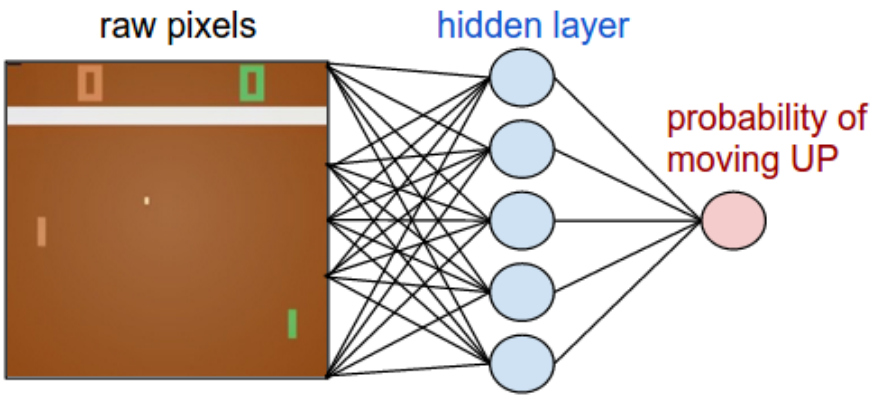
Policy learning is a more straightforward alternative in which we learn a **policy function,** π, which is a direct map from each state to the best corresponding action at that state. Think of it as a behavioral policy: "when I observe state *s*, the best thing to do is take action *a*". For example, an autonomous vehicle's policy might effectively include something like: "if I see a yellow light and I am more than 100 feet from the intersection, I should brake. Otherwise, keep moving forward."

$$a = \pi(s)$$

A **policy** is a map from **state** to **action.**

So we're learning a function that will maximize expected reward. What do we know that's really good at learning complex functions? Deep neural networks!

Andrej Karpathy's Pong from Pixels provides an excellent walkthrough on using **deep reinforcement learning** to learn a policy for the Atari game Pong that takes raw pixels from the game as the input (state) and outputs a probability of moving the paddle up or down (action).



In a policy gradient network, the agent learns the optimal policy by adjusting its weights through gradient descent based on reward signals from the environment. Image via http://karpathy.github.io/2016/05/31/rl/

If you want to get your hands dirty with deep RL, work through Andrej's post. You will implement a 2-layer policy network in 130 lines of code, and will also learn how to plug into OpenAI's Gym, which allows you to quickly get up and running with your first reinforcement

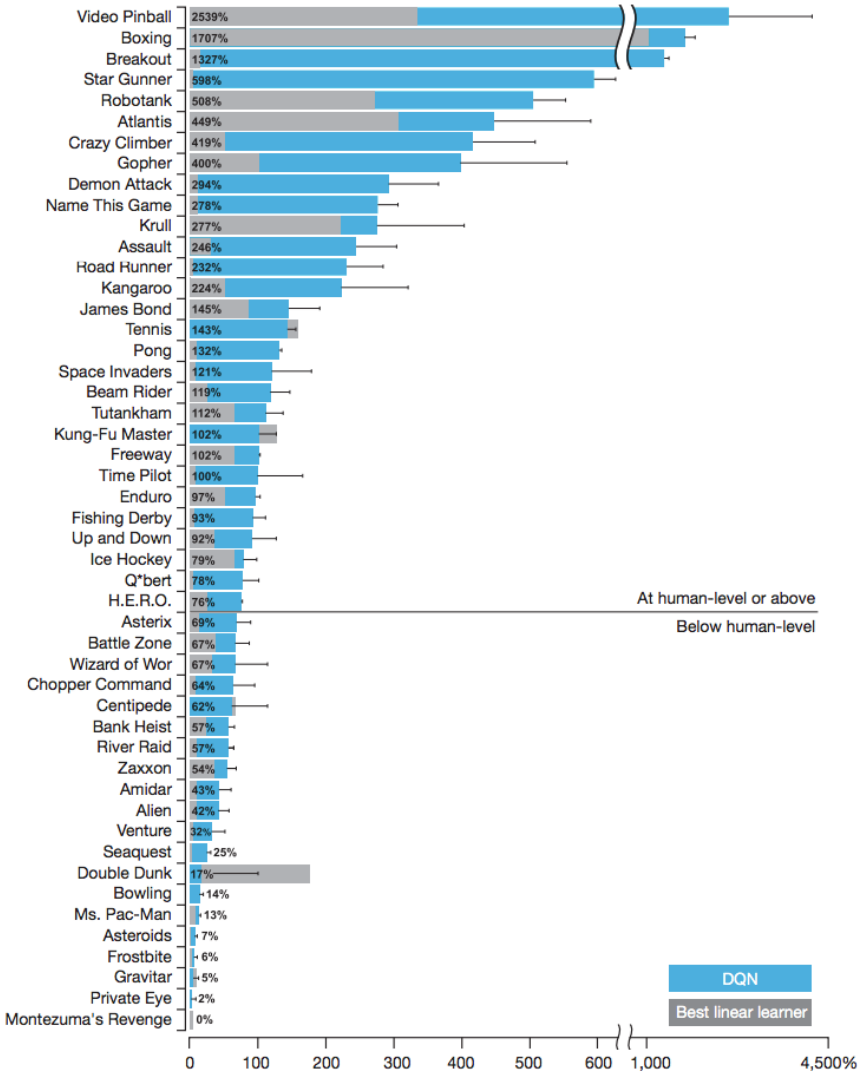
learning algorithm, test it on a variety of games, and see how its performance compares to other submissions.

DQNs, A3C, and advancements in deep RL

In 2015, DeepMind used a method called **deep Q-networks (DQN)**, an approach that approximates Q-functions using deep neural networks, to beat human benchmarks across many Atari games:

We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was 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, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks. (Silver et al., 2015)

Here is a snapshot of where DQN agents stand relative to linear learners and humans in various domains:



These are normalized with respect to professional human games testers: 0% = random play, 100% = human performance. Source: DeepMind's DQN paper, Human-level control through deep reinforcement learning

To help you build some intuition for how advancements are made in RL research, here are some examples of improvements on attempts at non-linear Q-function approximators that can improve performance and stability:

- **Experience replay**, which learns by randomizing over a longer sequence of previous observations and corresponding reward to avoid overfitting to recent experiences. This idea is inspired by biological brains: rats traversing mazes, for example, "replay" patterns of neural activity during sleep in order to optimize future behavior in the maze.

- **Recurrent neural networks (RNNs)** augmenting DQNs. When an agent can only see its immediate surroundings (e.g. robot-mouse only seeing a certain segment of the maze vs. a birds-eye view of the whole maze), the agent needs to remember the bigger picture

so it remembers where things are. This is similar to how humans babies develop object permanence to know things exist even if they leave the baby's visual field. RNNs are "recurrent", i.e. they allow information to persist on a longer-term basis. Here's an impressive video of a **deep recurrent Q-network (DQRN)** playing Doom.

Paper: https://arxiv.org/abs/1609.05521. Source: Arthur Juliani's Simple Reinforcement Learning with Tensorflow series

In 2016, just one year after the DQN paper, DeepMind revealed another algorithm called **Asynchronous Advantage Actor-Critic (A3C)** that surpassed state-of-the-art performance on Atari games after training for half as long (Mnih et al., 2016). A3C is an **actor-critic** algorithm that combines the best of both approaches we explored earlier: it uses an **actor** (a policy network that decides how to act) AND a **critic** (a Q-network that decides how valuable things are). Arthur Juliani has a nice writeup on how A3C works specifically. A3C is now OpenAI's Universe Starter Agent.

Since then, there have been countless fascinating breakthroughs—from AIs inventing their own language to teaching themselves to walk in a variety of terrains. This series only scratches the surface on the cutting edge of RL, but hopefully it will serve as a starting point for further exploration!

As a parting note, we'd like to share this incredible video of DeepMind's agents that learned to walk… with added sound. Grab some popcorn, turn up the volume, and witness the full glory of artificial intelligence.

Practice materials & further reading

Code

- *Andrej Karpathy's Pong from Pixels will get you up-and-running quickly with your first reinforcement learning agent. As the article describes, "we'll learn to play an ATARI game (Pong!) with PG, from scratch, from pixels, with a deep neural network, and the whole thing is 130 lines of Python only using numpy as a dependency (Gist link)."*
- *Next, we'd highly recommend Arthur Juliani's Simple Reinforcement Learning with Tensorflow tutorial. It walks through DQNs, policy learning, actor-critic methods, and strategies for exploration with implementations using TensorFlow. Try understanding and then re-implementing the methods covered.*

Reading + lectures

- *Richard Sutton's book, Reinforcement Learning: An Introduction—a fantastic book, very readable*
- *John Schulman's CS 294: Deep Reinforcement Learning (Berkeley)*
- *David Silver's Reinforcement Learning course (UCL)*

YOU DID IT!

> *If you've made it this far, that is all the reward we could hope for.*