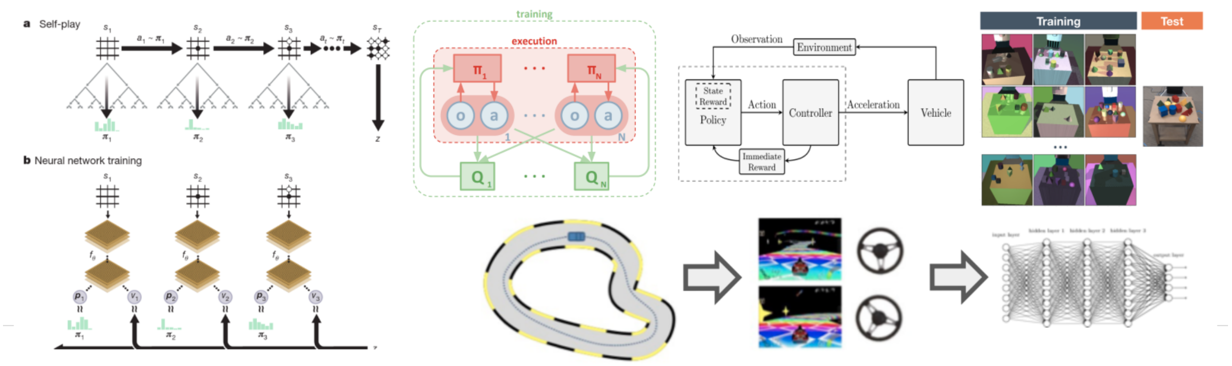
Reinforcement Learning: Looking Ahead and Getting Started



In previous posts, I **introduced reinforcement learning** and then got into **deep reinforcement learning methods**. These posts dealt with foundational theory and algorithms that broadly describe the field of reinforcement learning (RL).

This post is the stuff that was left over. There were lots of cool things in RL I wanted to talk about which didn't fit the mold of the last two posts. The topics I will cover are:

1. Multi-agent RL methods
2. The main problems in RL that keep us awake
3. Educational resources to get started with RL

I hope you find this interesting despite the lack of unifying theme. Feel free to pick your section(s) to read, as they can all be looked at independently.

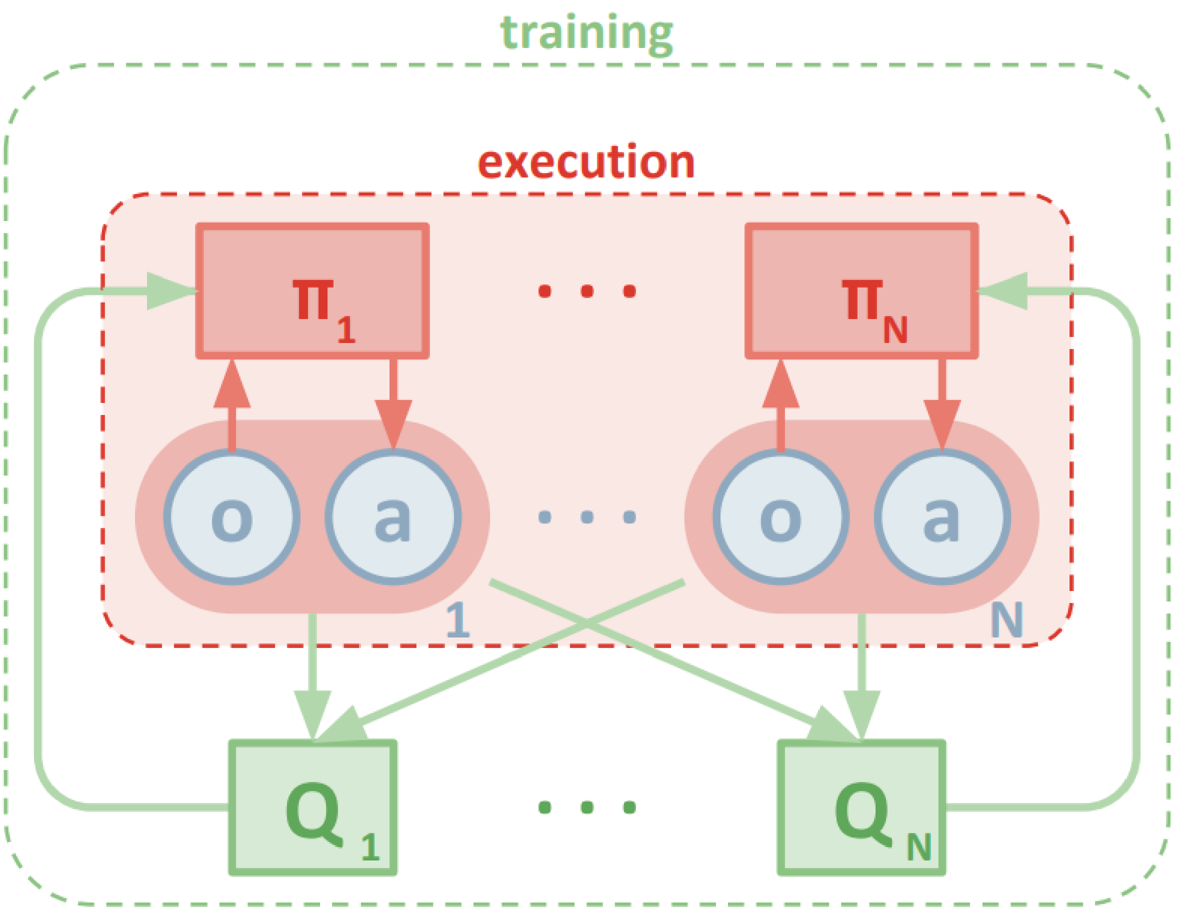
Multi-Agent Reinforcement Learning

There is an entire subset of RL dedicated to multi-agent systems, unsurprinsingly named **Multi-Agent RL (MARL)**. As outlined in **this paper**, MARL methods can be split up into a few categories.

Centralized vs. Decentralized

- **Fully centralized:** Basically, this is the same as a single-agent RL problem, but the observations and actions happen to be spread across multiple agents. No new techniques are needed.
- **Centralized learning, decentralized execution:** Think of having a critic that collects all agents' experiences and jointly trains the policies for each agent (or actors). However, once the policies are learned, each actor operates standalone. This is the key concept behind **Multi-Agent DDPG (MADDPG)**.
- **Fully Decentralized:** Here, each agent learns on its own. Again, this problem can be somewhat trivial on the surface, but things get interesting when agents are able to observe other agents. This is especially true if the agents are networked — that is, they have a way to communicate with each other. For example, they can broadcast their intent to one another as a learnable signal in the observation.

One cool example is **Social Influence as Intrinsic Motivation for Multi-Agent Deep Reinforcement Learning** (Jaques et al. 2018), in which agents are rewarded by how much they are able to influence other agents to change their behaviors. This paper also explores centralized and decentralized methods, where the latter requires agents to learn their own Model of Other Agents (MOA).



Multi-Agent DDPG diagram showing centralized critic (green) and decentralized actors (red).
Source: https://arxiv.org/abs/1706.02275

Cooperative vs. Competitive

- In *cooperative* systems, the agents work towards a common goal, which fits more closely with how we think of a single-agent system trying to maximize its own return. Now, all the agents aim to maximize a collective return. Here is **one modification of MADDPG** where agents consider joint team and individual reward.
- In *competitive* systems, the agents are usually placed into teams. Many demonstrations of competitive MARL deal with two teams so the problem can be expressed as a zero-sum game, like is done with the amazing **AlphaGo Zero** from Google DeepMind. A cool consequence of competitive multi-agent systems is that they can be trained through *self-play*.
- There are also *mixed* systems which are essentially represented as a collective of “selfish agents” that try to maximize their own return with little to no regard for other agents.

For more information on MARL, which is still a maturing area, take a look at **this presentation by Shimon Whiteson from Oxford**.

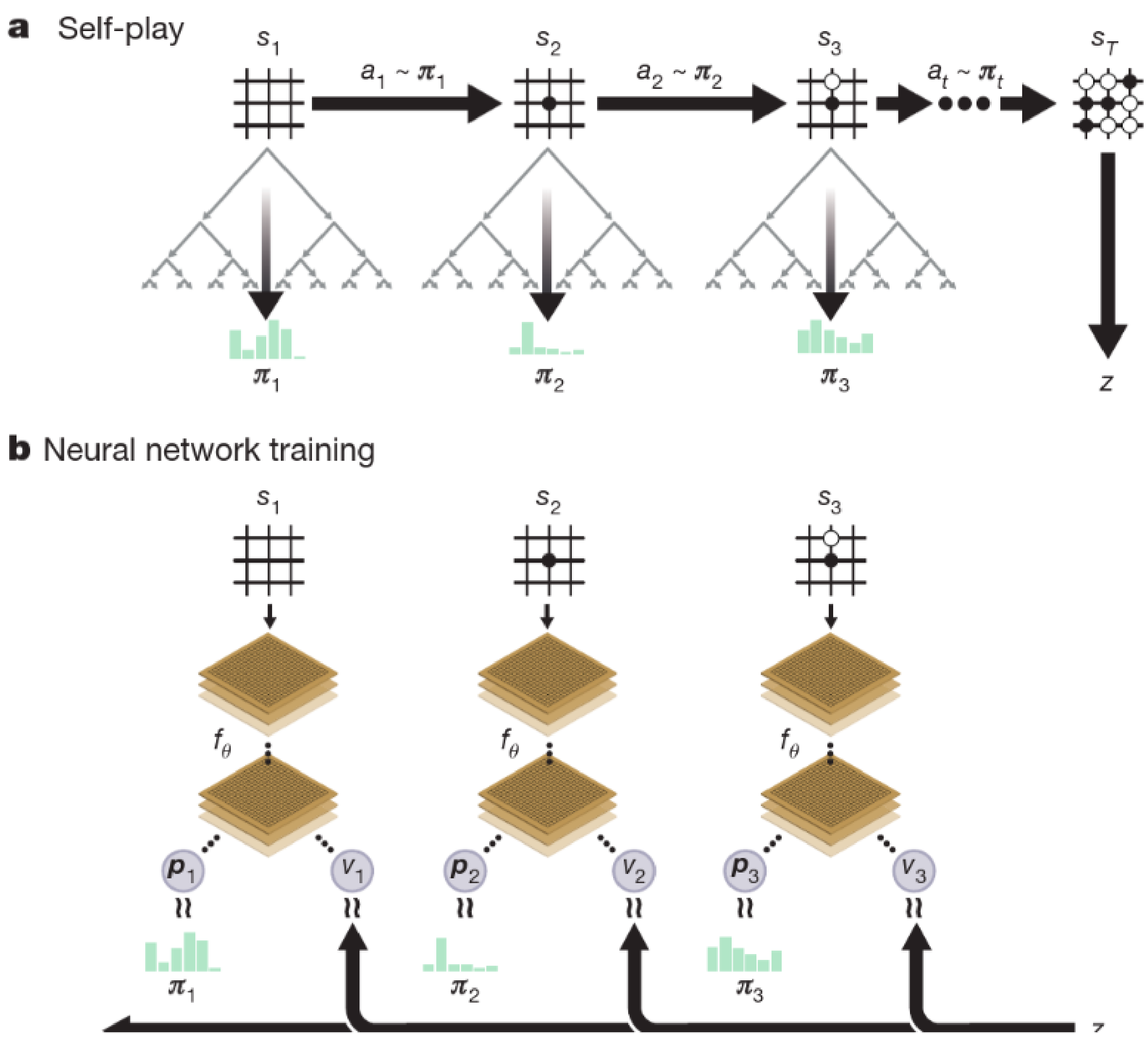


Diagram of AlphaGo Zero, which combines deep reinforcement learning for value estimation and action selection with Monte Carlo Tree Search (MCTS) for searching over long game sequences.
Source: https://deepmind.com/research/publications/mastering-game-go-without-human-knowledge

Expectation Setting: RL Ain’t Magic

RL is a big field and deep neural networks for function approximation has been key to solve more complex problems. This is especially the case when dealing with systems that have continuous states and/or actions, or with spatiotemporal aspects of observations that architectures like convolutional or recurrent neural networks can handle whereas tables are severely limited.

I want to share some important thoughts on the hype behind “end-to-end” reinforcement learning. While impressive, many RL methods today (especially the ones that get media coverage) are notoriously narrow and brittle systems — meaning that many models are trained to do a specific thing well and no more than that. This is not how humans or animals learn, keeping in mind that a big purpose of AI is to understand these phenomena and “put them to paper”, so to speak.

So when you see RL being used — for example — to train a simulated robot to walk, I want you to think about a few things.

1. How do you come up with a good reward function?
2. If the simulator’s physics engine and/or the lengths of the robot links changed slightly would the policy still work?
3. If you tried to deploy this on a real robot, would the policy still work? (Spoiler alert: No)
4. If we already have a physical model of the robot and a “solved” field of controls that could make the robot move stably, why are we trying to learn weird† motions from scratch?
5. If you asked the robot to perform a different task (say, walking sideways instead of forward), could it do this without having to relearn from scratch?

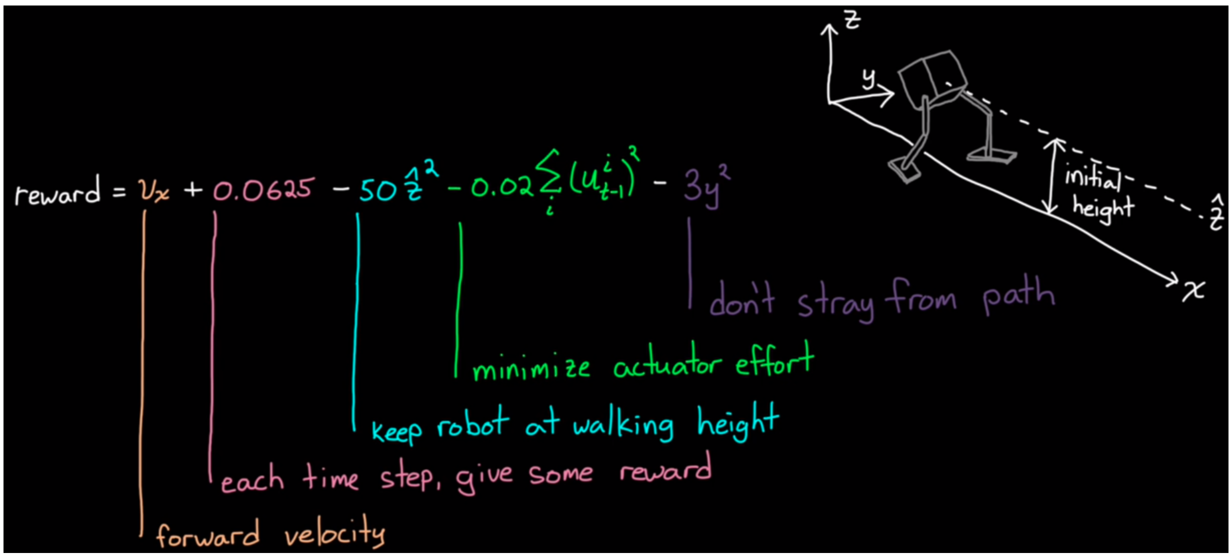
I don’t want to discredit the community’s incredible work, because tackling academic problems is the natural order of how research eventually breaks out into the mainstream. However, I want to make it clear that end-to-end RL can be a great show of force, but today is still fairly impractical for engineering design — especially when dealing with physical systems in the real world. That said, there is *so* much current work that tries to address these issues, with the hopes that someday RL does make the shift from academic playground to practical toolkit.

†Google DeepMind uses the word “idiosyncratic” in their **Emergence of Locomotion Behaviour in Rich Environments** work from 2017.

Reward Shaping

It is hard to learn from sparse rewards — for example, waiting until the end of a multi-turn game to get a measly binary win/lose signal, or giving the robot a “thumbs up” after it successfully traverses an entire obstacle course. While RL is sometimes advertised as a “magical” framework that learns from nothing, the truth is that there can be a very manual and time-consuming task of *reward shaping* (or *reward engineering*) to help guide the agent towards solving its task.

In other words, it can be extremely challenging to get an algorithm to learn what you want it to learn — while algorithmic choices are important, a lot falls on the reward function. For games, you may have a good enough proxy reward like the score, but how do you quantify how well a robot walks or picks up an object, for example? **This blog post** explains the reward engineering problem beautifully. I have also struggled with **my own reward shaping task** as you can see in the following picture.



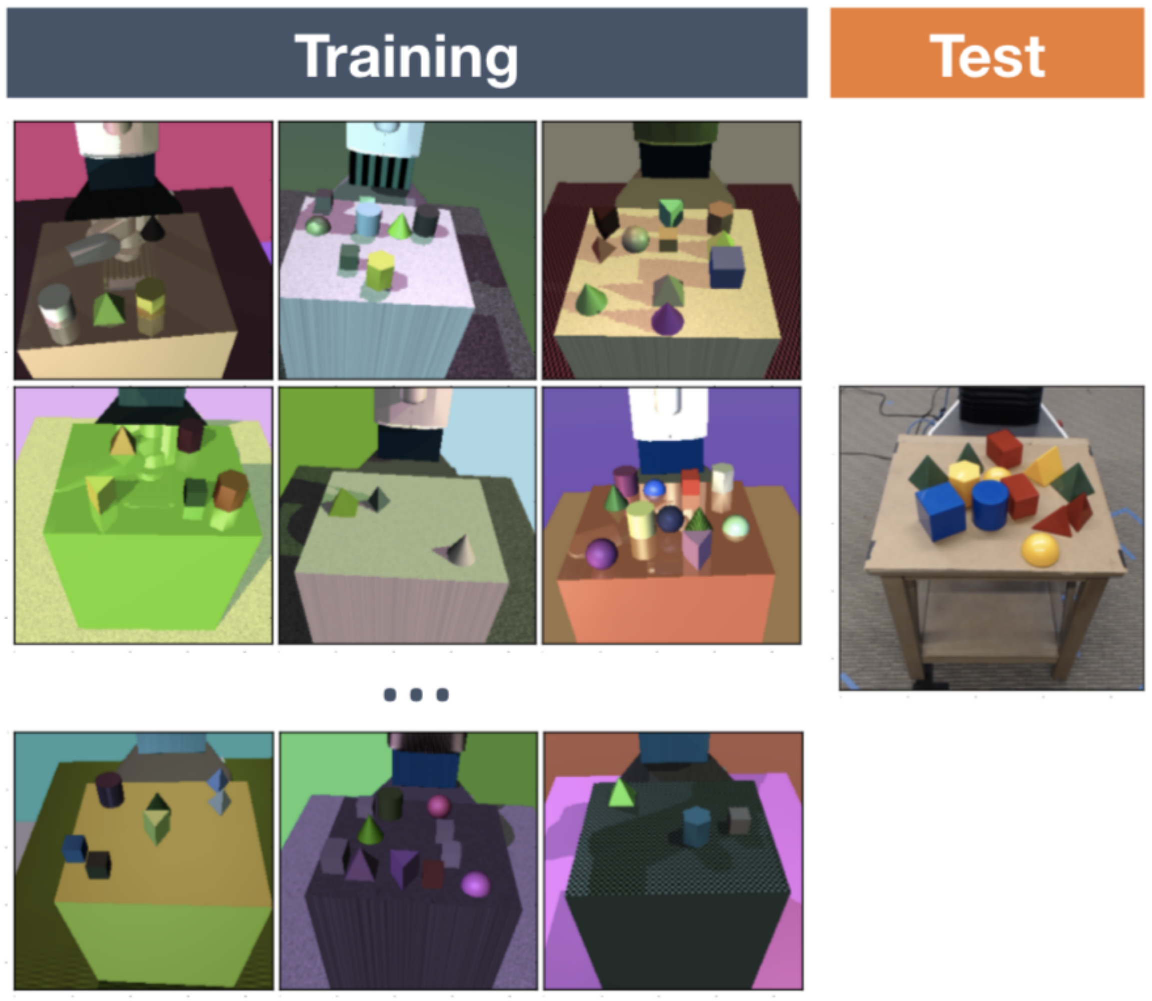
$\text{reward} = v_x + 0.0625 - 50\hat{z}^2 - 0.02\sum_i (u_{t-1}^i)^2 - 3y^2$
forward velocity
each time step, give some reward
keep robot at walking height
minimize actuator effort
don't stray from path
z
y
x
initial height
$\hat{z}$

The reward function converged on for the MATLAB walking robot example. It took a lot of manual engineering on my part.
Source: **Video by Brian Douglas**

Sim-to-Real

This is a challenge often found in robotics applications of RL, so obviously I care about it. Many robotics tasks require visuomotor skills — that is, there is a perceptive element (typically vision) that guides an embodies agent (the robot) to physically interact with the environment. At the same time, simulation is crucial in robotics because running untrained policies with exploration on a real robot can be risky. How do you ensure that a learned policy in simulation can transfer over to the real-world without having to relearn? This collection of problems is formally known as *sim-to-real*.

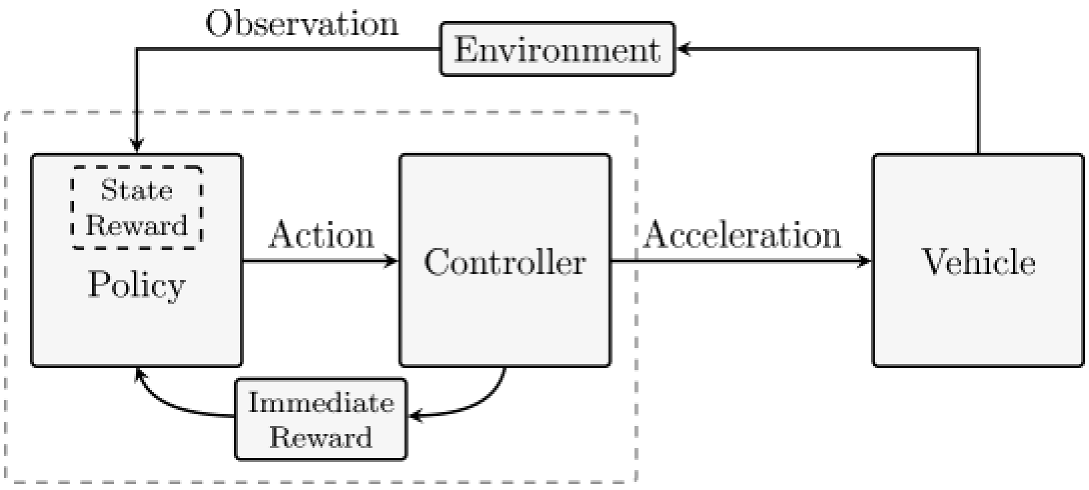
One approach, of course, is to have better simulators. Another is to leverage the fact that simulation is easy to configure. With techniques like ***domain randomization*** (learning on randomized simulator physics, perception, or both), we can prevent a learned policy from overfitting to artifacts unique to one simulation environment. The goal is to learn on a variety of simulation conditions to get a regularized policy with a higher likelihood to transfer to the real world — whether this means reducing or entirely eliminating the need to retrain on real hardware.



Domain randomization for sim-to-real in a robotic pick-and-place task.
Source: [Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World](), Tobin et al. 2017

Combining Controls and RL

Going back to one of my deflating questions: If there exists a working model-based control strategy for your agent, why throw it away? There is an argument to learning everything for academic reasons, but many cool and practical robotics applications of RL combine controls with learning what controls cannot solve — for example, going from pixels to some high-level command that a controller can then execute. This not only makes learning much more efficient, but allows us to take advantage of many of the safety and stability guarantees that a fully interpretable controller provides.



A system architecture that combines reinforcement learning for high-level decision making and a low-level controller.
Source: [Learning When to Drive in Intersections by Combining Reinforcement Learning and Model Predictive Control]

There is also the concept of *imitation learning*. Instead of learning from scratch with a random policy, we can bootstrap our policy parameters to "copy" an expert — either a human operator or a non-learned policy such as a controller. This method is called *behavioral cloning*, which is a supervised learning problem where the data is the observation and the labels are the action that the expert took given that observation. Now we have model parameters that will presumably be much better than random, which ideally means less iterations for RL to find a solution.

Of course, there might be problems with starting RL from a local minimum enforced by an expert. If there *is* a much better solution that we did not think of, it will be less likely to reach it from this new starting point. One way to address this is using another imitation learning technique called *inverse reinforcement learning (IRL)*. In IRL, a demonstration is used to learn a reward function. Then, that learned reward function can be used for training a policy without the bias induced by bootstrapping the policy through something like behavior cloning.

For more information on imitation learning, this tutorial is great.
Source: https://sites.google.com/view/icml2018-imitation-learning/

Learning to Generalize

There has been much work that reaches into the cognitive science bucket and tries to use RL in a way that can generalize across tasks, akin to how we humans would learn. For example, if we had never seen an Atari and learned to play one game from scratch, we expect that learning a new game would be much quicker since we now know generic things like the joystick probably moves your player.

Implementing an RL system like this is still very much a research area, and typically involves a hierarchical approach that at the lower level learns common features to make it understand and operate in the world. Then, the higher levels can use what has already been learned and adapt it to specific tasks. In these types of problems you also see applications of *unsupervised learning* to learn latent representations of the world that help generalize across tasks in a lower dimensional space compared to raw observations (such as images).

Here is an incomplete smattering of papers I have encountered which deal with this general idea. If you have other suggestions, let me know!

- **Model-Agnostic Meta-Learning (MAML)**: A method to learn how to perform "well enough" on a variety of tasks, thus reducing the time needed to learn new tasks which are similar.
- **Hierarchical Reinforcement Learning with Advantage-Based Auxiliary Rewards (HAAR)**: Concurrent learning of high-level and low-level tasks that occur at different time scales. [**Videos**]
- **Deep Skill Chaining:** A hierarchical RL method that learns a high-level policy which enables an agent to choose from a set of discrete skills, which can be given by experts or learned using — for example — unsupervised methods. [**Video**]
- **Playing Hard Exploration Games by Watching YouTube**: This method uses self-supervised learning to get a low-dimensional latent representation based on video and audio that spans across tasks (in this case, videogames). Then, this latent representation can be used to derive a reward function that imitates an expert demonstration. [**Videos**]

- **Data-efficient Hindsight Off-policy Option Learning**: This is a brand-new paper from Google DeepMind that just came out a few days before posting this — so I had to add it! The authors divide their policy into a "high-level policy" that chooses high-level options based on state (or observation), and a "low-level policy" that chooses lower-level actions based on state and selected option. If you read my **previous post on deep reinforcement learning**, you'll see that this is a hierarchical twist on an off-policy actor-critic algorithm (like DDPG) that is stabilized by trust-region constraints (like PPO). In summary, it's an RL *tour de force*.

Animation of various trajectories in the Atari game **"Montezuma's Revenge"** and a low-dimensional visualization of learned embeddings from joint video and audio, which are used for reinforcement learning.
Source: **Playing hard exploration games by watching YouTube**

Getting Started with Reinforcement Learning

Since you've obviously read through my 3 posts by now, it's time to go out into the world, learn more, and do more! This section contains resources to get started with reinforcement learning, both in terms of theory and hands-on programming.

My personal path to RL was non-traditional: After possibly solving a grid world or discretized pendulum-cart system in university classes, I returned to RL through the **MathWorks Reinforcement Learning Toolbox** in 2019 while I worked there, with the hopes to recreate something akin to the **original DDPG paper** in time for the toolbox release. What resulted was a **walking robot video** (which demonstrates how little I knew about RL at the time) and a **great companion video by Brian Douglas**. I also helped develop **this example** in which a simulated robot learns to avoid obstacles using lidar measurements. The University of Chile RoboCup team did something similar with a SoftBank Pepper robot using both lidar and depth images. Here is **their paper** and a **video of the robot**.

Since then, I've taken a step back to make sure I know the foundations well enough to try reading papers and (sort of) make sense of what's going on. Below I will share some links that helped me learn enough to attempt these blog posts, and hopefully will also help you on your RL journey.

Educational Resources

For a get a deeper theoretical foundation of RL, I would recommend the following.

1. **Reinforcement Learning: An Introduction** book by Andrew Barto and Rich Sutton
2. **Reinforcement Learning course** from University of Central London (UCL) by David Silver
3. **Reinforcement Learning Lecture Series** from Google DeepMind and UCL, 2018
4. **Deep Reinforcement Learning course** from UC Berkeley by Sergey Levine
5. **Deep Reinforcement Learning Nanodegree** from Udacity [**GitHub**]
6. **Spinning Up in Deep RL** from OpenAI
7. **Blog posts** by Lilian Weng from OpenAI

I took the **Udacity Deep Reinforcement Learning Nanodegree** while Udacity offered a free month at the start of the 2020 COVID-19 pandemic, and it helped enormously. While the course is not free, even if you do not enroll **their course examples are available on GitHub**. This course was great because it "forced" you to implement several algorithms between the in-class exercises and course projects. As for those projects, you can find my solution repositories here which contain Jupyter notebooks:

1. DQN for banana collection game
2. PPO (continuous action-space) for simulated robot arms
3. Multi-Agent DDPG for Tennis game

Software Tools You Should Know

While theory and courses can be important, there is simply no replacement for hands-on experience — so get coding! There are a few tools out there you might want to look into, but note that this area is still very much in flux so there is nothing standard that will solve all your problems. Exciting, right?

- First, you should be comfortable with deep learning frameworks like PyTorch and TensorFlow to implement your own neural networks and/or algorithms.
- OpenAI Gym for some premade environments and the ability to make new environments as Python classes. They also have a separate repository with baselines for popular RL algorithms.
- RLLib, an open-source library built on top of Ray, which focuses on scalable distributed computing. It can integrate with PyTorch, TensorFlow, and OpenAI Gym.
- Unity ML-Agents helps create RL environments around the Unity game engine. This is what the Udacity Deep RL Nanodegree uses.
- Reinforcement Learning Toolbox by MathWorks lets you use RL to train agents in environments built with MATLAB and Simulink. I recommend this if it makes sense to model your environment in MATLAB/Simulink and you are OK with a standard set of neural network components and learning algorithms that you think will need little to no customization. MathWorks' AI sweet spot is applied machine learning whereas the cutting-edge research has almost exclusively standardized on Python.

Hope you have enjoyed this RL survey blog series. If you didn't read the previous posts, they may help provide more background on the **foundations of RL** and the **role of neural networks in deep RL**; else, you have some great resources a few paragraphs above.

If there is anything in RL (or other robotics/AI/software topics) you would like to see in a future blog post, or have other suggestions for learning materials, please leave a comment or reach out. Cheers!

Sebastian Castro **August 22, 2020**
Artificial Intelligence, **Machine Learning**, **Reinforcement Learning**

alphago zero, hierarchical reinforcement learning, imitation learning, inverse reinforcement learning, meta-learning, multi-agent, reward shaping, sim-to-real