A Comprehensive Introduction to Reinforcement Learning

Likhit Vyas Yarramsetti
y.vyas.likhit@gmail.com

I. Introduction

Reinforcement learning (RL) has been a significant part of AI research for the past four decades. In a nutshell, RL is a type of machine learning (ML) where an AI agent learns to solve decision-making problems in environments through trial and error. Such an agent learns which decisions are the best to make in any given situation to maximize some notion of cumulative reward (Sutton & Barto, 2018).

Research in reinforcement learning has experienced explosive growth in the past two decades with the emergence of *deep reinforcement learning* in which RL is integrated with deep neural networks to handle more complex data and tasks. Deep RL has been the central technology behind many of the incredible AI breakthroughs in the later 2010s such as Google DeepMind's DQN algorithm which learned how to play Atari in 2015 from just looking at raw pixels (Mnih et al., 2015). Another notable example is the invention of AlphaGo in 2016, which defeated the Go game world champion Lee Sedol (Silver et al., 2016). Later, DeepMind researchers invented MuZero, an advanced RL model that can theoretically solve any game or task without requiring any prior knowledge of them as long as it can repeatedly interact with the environment and observe the outcomes of its actions (Schrittwieser et al., 2019). This was a significant achievement as it represents a step toward adaptable, rule-free learning in games and beyond. Today, RL is considered one of the "big three" branches of ML, alongside supervised and unsupervised learning and continues to be an area of intense research and development.

II. How Does Reinforcement Learning Work?

At the heart of RL is the idea of an agent interacting with an environment. Unlike supervised learning, where the model learns from a dataset of labeled examples, an RL agent learns by directly interacting with the environment in which it is in. Once this learning process is over, an RL agent has a better idea of which action to take at any given situation or *state*. For example, if we wanted to make an AI agent that autonomously plays a video game, the agent might be the virtual, playable character and the environment is the game world itself. In any game, the player must take actions to achieve a goal or collect reward. Through trial and error, the RL agent learns which are the best actions to take and most effective. It essentially maps situations to actions to maximize a reward signal.

To understand RL well, we must first lay the foundation by exploring **Markov Decision Processes (MDPs)**, a mathematical framework for modeling decision-making problems in uncertain situations. All reinforcement learning algorithms are essentially solving some variation of an MDP. Formally, an MDP is defined as a tuple $< S, A, T, R >$ where:

- S is the finite set of all possible states in the environment, also called the *state space*;
- A is the finite set of all actions an agent can take;
- $T: S \times A \times S \rightarrow [0, 1]$ is a transition function, which specifies the probability $T(s, a, s')$, i.e., the probability that if the agent is in state s and takes action a, it lands in state s'
- $R: S \times A \times S \rightarrow \mathbb{R}$ is a reward function that specifies the reward $R(s, a, s')$ that the agent will get if it is in state s, takes action a, and lands in state s'

The objective is the same as in RL: to find a mapping of states to actions $S \rightarrow A$ that maximizes the reward the agent can get. This mapping function is called the *policy*, often denoted as the function $\pi(s)$ which takes a state s and returns an action.

MDPs make some major assumptions about the problem being solved. The word "markov" comes from the assumption that the problem has the *Markov* property, the idea that the future of an agent only depends on the state the agent is currently in and zero consideration to its past history is required to be given. MDPs also assume that the agent has *full observability* meaning that it always perfectly knows exactly what state it is in. These are quite strong assumptions to make, as they do not hold in many real-world problems. Think of a robot navigating an environment with sensors: these sensors merely estimate the current state the robot is currently in, so they do not grant the agent full observability. Oftentimes, it is also necessary to keep track of an agent's history to make better decisions in the future. Still, with the many assumptions that MDPs make, they are incredibly useful for many problems and provide the basis for RL theory so it is vital to understand them.

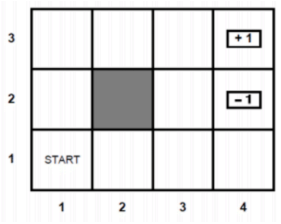

Figure 1: A simple MDP problem

Consider a gridworld as shown in Figure 1, in which an agent starts in the square $(1, 1)$. The state space S consists of all the squares the agent can be in except the square $(2, 2)$ which is a blocked cell and serves as a wall to all bordering cells. The set of actions the agent can take are $A = \{\uparrow, \rightarrow, \downarrow, \leftarrow\}$. Suppose the state transition function T is as follows: any action taken brings the agent to the intended square (i.e., the square in the direction of the action) with probability 0.8 and the squares to the left and right of the direction of the action with probabilities 0.1 each (see Figure 2). If the result of an

action hits a wall, the agent stays in place. Suppose the reward function is such that landing in any non-terminal square gives the agent a reward of -0.04 and the terminal squares $(4,3)$ and $(4,2)$ give a reward of $+1$ and -1 respectively. Landing in a terminal square effectively ends the episode or process.

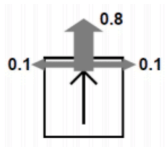

Figure 2: The state-transition dynamics of the simple MDP

What is the best policy to solve this MDP? One idea for a policy π is shown in Figure 3.

→	→	→	+1
↑		↑	-1
↑	→	↑	←

Figure 3: An idea for a policy π to solve the MDP

At first glance, this policy seems like a good idea: it tries to minimize the accumulated cost of living and get a large reward by trying to get to the reward state $(4,3)$ as fast as possible. However, if we consider the square $(3,2)$, we will see that there is a 0.1 probability that the agent lands in the negative terminal state $(4,2)$. We can evade that square altogether if choose action ← at state $(3,2)$. How can we know which policy is best? Is there a way of evaluating a policy's value?

This is where the idea of a **value function** comes in, a core idea in reinforcement learning. A value function $V: S \to \mathbb{R}$ associates a numerical value with each state. The value of a state is essentially the *expected cumulative reward* that the agent can get at that state (i.e., the reward that the agent can reasonably expect to get on average from that state onwards provided that the agent follows the policy). The function $V^{\pi}(s)$ denotes the value of the the policy $\pi(s)$ at a given state s. How do we compute this value function? We can use the following equation to iteratively compute the value of every state until convergence:

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

This is the famous Bellman-backup equation for policy evaluation, and is the core idea behind many MDP and RL solving algorithms.

The symbol γ is called the *discount factor*, a hyperparameter that is always set to a value between 0 and 1 and plays a crucial role in determining how future rewards are valued compared to immediate rewards. The closer it is set to 0, future rewards are not valued highly whereas if it is closer to 1 future rewards impact the expected reward greatly.

So how can we compute the best policy? A brute-force approach would be to enumerate through all possible policies, compute their value functions, and find the policy with the best value. However, if we think about time complexity this falls apart due to the fact that if there are a actions and s number of states in the state space, there are a^s possible policies. So this approach has exponential complexity in the number of states. Fortunately, there exists a method to compute the optimal policy by computing the optimal value of each state directly. This method is called **value iteration** and uses the Bellman equation:

$$V(s) = \max_{a} \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma V(s')]$$

Algorithm 1 Value Iteration.

Initialize $V^*(s) = 0$ forall $s \in S$
while until V^* **converges:**
  for $s \in S$:
    Update $V^*(s) = \max_{a} \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$
    Update $\pi(s) = argmax_a \sum_{s' \in S} T(s, a, s')V^*(s')$
  endfor
endwhile
return V^*, π

When the value iteration algorithm finishes, it will have computed V^*, the optimal value or maximum expected cumulative reward of every state. Using this, the agent adopts the greedy policy of selecting the action a that leads the agent to the most valuable states with the highest probability. In other words, we select $argmax_a \sum_{s' \in S} T(s, a, s')V^*(s')$.

Some improvements can be made to the value iteration algorithm. For instance, since we know how to evaluate a fixed policy, we can first try coming up with a random policy π, evaluate it, then for each state update the policy to the action a that gives the highest $\sum_{s' \in S} T(s, a, s')V^{\pi}(s')$. This strategy is known as the **policy iteration** algorithm.

Algorithm 2 Policy Iteration.

Choose a random policy π
while until π **does not change:**
  Evaluate V^{π}
  for $s \in S$**:**
    $\pi'(s) = argmax_a \sum_{s' \in S} T(s, a, s')V^{\pi}(s')$
  endfor
  Update $\pi = \pi'$
endwhile
return π

So when does reinforcement learning come into the mix? The only difference between MDP problems and RL problems lies in the fact that in RL, the transition function T and the reward function R are not given to the agent. The agent must come up with an optimal policy π without this knowledge. Through its interactions with the environment, an RL agent will learn the environment's state-transition dynamics and build an understanding of the transition function T. Whenever it experiences a reward, it will also update its understanding of the reward function R accordingly. Using this approximate understanding of T and R, it will try to come up with an optimal policy.

RL generally comes in two flavors: passive and active. In passive RL, we are mainly evaluating a fixed policy. The agent has no control over the decision-making, and it just executes a given policy and learns from the experience. The main objective here is to learn the state values (and maybe construct a learned model, e.g., a transition function T). In active RL, the agent learns and adjusts its policy based on the experienced rewards and it actively seeks out optimal actions. The agent sometimes may even choose to randomly do a different action than what its current policy suggests. Choosing between the options of passive and active algorithms is largely dependent on the problem being solved and your goals as an RL agent architect.

III. Model-based RL

There are two major types of reinforcement learning: **model-based** and **model-free**. What is a model? A model is anything that the agent uses to predict the outcome of its actions.

Model-based RL involves constructing or using a model of the environment to plan actions before they're taken. An example is an aspiring pilot using a flight simulator to learn how to safely operate an airplane. The pilot is essentially learning a mental model of how a flight's state-transition dynamics work, and uses this knowledge to take the right actions during actual flights. Model-based learning is especially useful when dealing with problems whose environments do not have the property of **ergodicity**. If an environment is ergodic, it means that the agent can reach all of the states in the environment if given enough time to explore no matter where the agent starts. If an environment is non-ergodic, some states in the environment may be difficult to reach or completely inaccessible from certain other states. An example of a problem with a non-ergodic environment is the problem of self-driving. In this problem, the agent is the autonomous, self-driving vehicle and the environment consists of the roads and street intersections. If the vehicle crashes, it will have effectively reached a state in which it cannot reach any other state in the environment. In such problems, using a model to plan actions is especially necessary because of the massive stakes involved.

For the MDP problem shown in Figure 1, suppose we did not know the transition function T or reward R, effectively making it an RL problem. Let's say that our goal was to construct a learned model of the environment, i.e., we are trying to build the T and R functions. To be able to do this, we must move around the environment, try to explore the entire state space, and observe the experienced transitions and rewards. Using this experience, we can build an approximate T and R. Let's fix a policy to this problem so we can do just that (see Figure 4).

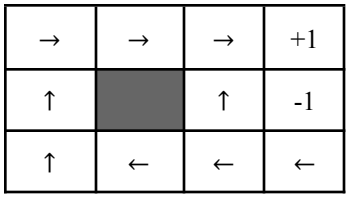

Figure 4: A fixed policy π to explore the environment

Let's say we start at the state $(1, 1)$. If we now act in the environment using this policy, we might observe a series of transitions like so:

Starting state s	Action $a = \pi(s)$	Resulting state s'	Experienced $R(s, a, s')$
$(1, 1)$	↑	$(1, 2)$	-0.04
$(1, 2)$	↑	$(1, 2)$	-0.04
$(1, 2)$	↑	$(1, 3)$	-0.04
$(1, 3)$	→	$(2, 3)$	-0.04
$(2, 3)$	→	$(3, 3)$	-0.04
$(3, 3)$	→	$(3, 2)$	-0.04
$(3, 2)$	↑	$(3, 3)$	-0.04
$(3, 3)$	→	$(4, 3)$	$+1$

Notice how the episode ended at the terminal state $(4, 3)$ with a reward of $+1$ experienced by the agent. To get a more accurate T, we must conduct more episodes so that our probability estimates more closely match the real T.

Suppose another episode went like this:

Starting state s	Action $a = \pi(s)$	Resulting state s'	Experienced $R(s, a, s')$
$(1, 1)$	↑	$(1, 2)$	-0.04
$(1, 2)$	↑	$(1, 3)$	-0.04
$(1, 3)$	→	$(2, 3)$	-0.04
$(2, 3)$	→	$(3, 3)$	-0.04
$(3, 3)$	→	$(3, 2)$	-0.04
$(3, 2)$	↑	$(3, 3)$	-1

Two episodes are not nearly enough to build an accurate model of an environment but we can start building T by

counting the outcomes s' of each state s and action a and normalizing to get an estimate of $T(s, a, s')$. For example, out of the 3 times we were in state $(3, 3)$ and executed the policy-given action of $\rightarrow$, only once did we land in the state of $(4, 3)$. So we can estimate this transition's probability as $T((3, 3), \rightarrow, (4, 3)) = \frac{1}{3}$. We can continue in a similar fashion to get the probability estimates of all of the state-action transitions. Now that we have constructed a learned model, we can use it to evaluate our policy. This is simple, as we just use the policy evaluation equation $V^{\pi}(s)$. What we just did is passive model-based learning: we learned a model of the environment by following a fixed policy and using our model to evaluate our policy.

An interesting observation here is that we can use our learned model of T and R to compute the greedy optimal policy using the Bellman equations! However, we run into a problem. Notice how if the real transition function T is how it is according to the dynamics shown in Figure 2, and we follow the policy shown in Figure 4, reaching the states $(3, 1)$ and $(4, 1)$ would be impossible. As a result, the agent may never know that the states $(3, 1)$ and $(4, 1)$ exist. If our goal was to construct an accurate model of the environment and use it to improve our current policy, not having visited all of the state space might be to our detriment because there could be states that yield higher rewards.

This leads us nicely to active model-based learning. Instead of starting with a fixed policy, we start by doing random actions in every state and using the traces we get from this exploration to construct the T and R functions. We can then use this information to compute and follow the greedy policy with respect to V^*. One problem with this approach is that while random exploration is guaranteed to get us to V^* asymptotically, it's too slow in a lot of cases. How can we resolve this issue?

One idea is to select a value ε between 0 and 1, which will be the probability with which we execute a random action so that we can explore the environment and possibly discover unseen states. With $1 - \varepsilon$ probability, we follow the greedy policy obtained from the learned T, R and computed V^*. How do we select a good value for ε? This is the critical **exploration/exploitation dilemma**, an important concept in RL and has a lot of interesting connections to what intelligent beings do in their lives: if you explore too little and not take too much risk, you might not discover better states (e.g., better food, prey, land, weather, mates) and thus follow a suboptimal policy whereas if you explore too much, you will learn too slowly which makes the learning process inefficient. Too much exploration also has little utility once the optimal policy is learned. In many RL algorithms, ε is often *annealed*, or gradually decreased over time. This helps the agent explore thoroughly in the early stages but focus on maximizing rewards in the later stages. This strategy of selecting actions is termed the **ε-greedy strategy**.

Algorithm 3 Active Model-Based Learning.

Initialize empty T, R, V^*, Initialize π to random policy
Get initial state s
while until V^* **converges:**
  Sample action a using ε-greedy strategy and execute
  Observe new state s'
  Update $T(s, a, s')$, $R(s, a, s')$
  if s' **is** terminal:
    $V^*(s) = R(s, a, s')$
    Sample new initial state s'
  else:
    Update $V^*(s) = \max_{\delta} \sum_{\theta \in S} T(s, \delta, \theta)[R(s, \delta, \theta) + \gamma V^*(\theta)]$
    Update $\pi(s) = argmax_a \sum_{s' \in S} T(s, a, s')V^*(s')$
  $s = s'$
endwhile

IV. Model-free RL

Model-free learning involves directly interacting with the environment through trial and error. Consider the following task: to compute the average of a set of numbers. However, the set is not given to us all at once but rather as a stream. Every time a new number or *sample* comes in, we are to compute a running average. How would one do this? The formula to compute the mean of a set of numbers in such an online fashion is as follows:

$$\hat{X}_{n+1} = \hat{X}_n + \frac{1}{n+1}(x_{n+1} - \hat{X}_n)$$

where:

- $\hat{X}_{n+1}$ is the new mean we are to compute
- $\hat{X}_n$ is the old mean estimate of the past n samples
- x_{n+1} is the new sample

Essentially, whenever we are given a new sample x_{n+1}, the new mean is the old estimate plus the weighted difference of the new sample and the old estimate. This is the core idea behind how model-free methods work. For a lot of problems, it is not possible to give an agent the entire transition function T for every possible state s or even specify a good-enough model. This is oftentimes due to the extremely large and complex state-space of the problem. Think back to when you learned how to ride a bicycle: did you maintain a mental model of a bicycle's state-transition dynamics to get a priori knowledge of how much torque to exert on one handle versus the other, how much force to use on each of the pedals, or how much your upper and lower body muscles had to contract to balance correctly? No, that would be silly. Rather, you learned by directly interacting with the bicycle and your muscle memory figured out which actions are the best to take in any given situation. You now reflexively take the correct actions in any given state because you learned that those actions have the

best value through experience. These are the kinds of problems where model-free approaches work well, where the agent needs to directly take actions in the complex environment and through its learned experiences updates its understanding of the values of its actions in an online fashion.

In fact, this is the very definition of **temporal-difference (TD) learning**. "If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning", said Richard Sutton and Andrew Barto, widely regarded as the founders of RL, in their book *Reinforcement Learning: An Introduction*, a comprehensive introduction to RL covering much of the field's development since the 1980s.

TD methods allow the agent to learn directly from raw experience without having to maintain a model of the environment's dynamics. This means that we can directly learn the values of each state without maintaining T or R. Whenever we act in the environment by executing an action a and observe the new state s' as well as the experienced reward $R(s, a, s')$, we obtain a new $sample = R(s, a, s') + \gamma V^{\pi}(s')$. Using this, we make a **TD-Update** using the equation:

$$V^{\pi}(s) = V^{\pi}(s) + \alpha[sample - V^{\pi}(s)]$$

where:

- $\alpha \in [0, 1]$ is called the *learning rate*, which is the weight we want to give to the difference of the new sample and old estimate

Notice how this equation bears a striking resemblance to the original online mean estimation equation. This is no coincidence. Every time we obtain a new sample, we make a new value estimation by adding to the old estimate the weighted difference of the new sample and old estimate. In TD-learning, the learning rate α starts with a value closer to 1 and is annealed over time to ensure convergence, prevent too much fluctuations, and reduce sensitivity to noise especially in the later stages of training. We are still doing passive RL, as we only are evaluating a fixed policy here, but in a model-free manner. How can we make this active? How can we learn an optimal policy in a model-free manner?

The ideas of exploration/exploitation come back into play. We can use the ε-greedy strategy to execute random actions and explore the state space. However, we run into a problem when selecting an optimal strategy based on the learned experience from exploration. The policy at any given state s is given by $argmax_a \sum_{s' \in S} T(s, a, s')V^{*}(s')$. We cannot make this calculation as we are not maintaining T, so we cannot get the value $T(s, a, s')$. Basically, we have an issue where we are trying to learn in a model-free way, but we need the model. How do we resolve this situation?

In active model-free RL, we are trying to transition away from thinking about *values of states* and towards thinking about *values of actions*. If we can somehow get a version of the value equation that allows us to obtain the values of actions, we can learn an action-value function using the ideas of TD-learning and use that learned function to easily obtain the policy at any given state: the action of highest action-value.

Notice this term in the original value iteration equation:

$$\sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma V(s')]$$

This is essentially the expected cumulative reward of executing a certain action a at a given state s. The value iteration equation *is* the maximization of this term over all actions $a \in A$. If we define this term as a function $Q(s, a)$, the value iteration equation can be rewritten as:

$$V(s) = \max_a Q(s, a)$$

Since we have now redefined $V(s)$, we can use it to replace $V(s')$ in the original term:

$$\sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma \max_{a'} Q(s', a')]$$

This essentially redefines $Q(s, a)$ as well:

$$Q(s, a) = \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma \max_{a'} Q(s', a')]$$

Now, the policy at any state s becomes:

$$\pi(s) = argmax_a Q(s, a)$$

The crutch of needing the model T to get Q-values still exists. However, the $Q(s, a)$ formulation allows us to estimate Q-values directly using the ideas found in temporal-difference learning. Whenever we execute an action a at any state s and experience a reward $R(s, a, s')$, we can make a Q update using the equation:

$$Q(s, a) = Q(s, a) + \alpha[sample - Q(s, a)]$$

where:

- $sample = R(s, a, s') + \gamma \max_{a'} Q(s', a')$

Algorithm 4 Q-Learning.

Initialize with $Q_0(s, a) = 0$ for all s, a
Get initial state s
while until Q **converges:**
  Sample action a using ε-greedy strategy and execute
  Observe new state s' and $R(s, a, s')$
  if s' **is** terminal:
    $sample = R(s, a, s')$
    Sample new initial state s'
  else:
    $sample = R(s, a, s') + \gamma \max_{a'} Q(s', a')$
  Update $Q_{t+1}(s, a) = Q_t(s, a) + \alpha[sample - Q_t(s, a)]$
  $s = s'$
endwhile

This is **Q-Learning**, of the most well-known RL algorithms. What makes this algorithm incredible is its *off-policy* nature, meaning that it always converges to an optimal policy even if the agent acts suboptimally. Some caveats are that you still have to explore enough and anneal α

enough over time but not decrease it too quickly. Specifically, α must be annealed over time t such that two properties hold:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \qquad \sum_{t=0}^{\infty} \alpha_t^2 < \infty$$

These two relations are basically saying that the learning rate should be large enough initially to allow the agent to learn well from the high amount of exploration at the beginning, and that it must decrease quickly enough for convergence.

VI. Deep Reinforcement Learning

Perhaps one of the most incredible breakthroughs in RL and AI in general is **deep reinforcement learning**. In fact, RL research experienced a bit of a "lull" up until the early 2010s primarily due to the significant limitation of not having enough compute resources to deal with the complex state-spaces of many problems, until deep RL came along and revived the field by introducing new techniques for dealing with such large state-spaces.

So what is deep RL? Essentially, it's the integration of deep neural networks with RL, which enables us to handle high-dimensional data such as images and sounds. Up until now, we've done RL using **atomic** states–states that are treated as unique, indivisible units–and stored the entire transition function T or Q-values for every state $s \in S$ in memory. This is not feasible for a lot of problems. For instance, in a board game like Go, if we treat every possible board arrangement as a single state, the state-space would be larger than the number of particles in the universe. This gets worse when RL is applied to play video games like Atari. We can resolve this issue by adopting a **feature-based** representation of states, where each state is described in terms of its features or attributes. Instead of storing the functions π, V, or Q explicitly as a table in memory, we can use supervised-learning methods to train a neural network that takes as input the features of a state and returns a value. The challenge now shifts to achieving **generalization**: if our agent only visits a miniscule subset of the vast state-space, how can it *generalize* that experience so that it knows what to do for all states in the state-space?

DeepMind's development of **Deep Q-Networks (DQN)** is one of the first significant breakthroughs in deep RL. The algorithm approximates Q-values by training a neural network $Q(s, a, \theta)$ where θ represents the parameters of the network that approximates the $Q(s, a)$, and contains all the weights and biases of each of the layers. All the parameters in θ are randomly initialized. Using this initial Q-network, we first act in the environment using an ε-greedy strategy, collect the samples of our experiences (s, a, r, s'), and store them in an **experience replay buffer**. We then sample a random minibatch of transitions (s, a, r, s') from this buffer and use classic supervised learning methods such as stochastic gradient descent and backpropagation to train the network according to the loss function:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim Buffer}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]$$

During the training process, two networks are maintained: the **target network** and main policy network that we are training. Everytime we calculate the error $L(\theta)$, we are actually calculating the expectation of the square of the error between the target network's output which uses weights θ^- and the main network's output which uses weights θ. We periodically update the target network to just be our main network during the training process. This strategy of updating the target network less frequently than the main network reduces oscillations in the learning updates. To play the game Atari, DeepMind researchers fed as input to the Q-network the frames of the game, which are 210×160 images downsampled and resized to 84×84 images and used a convolution neural network (CNN) architecture to learn spatial features from the image without having to use hand-crafted features. Q-Networks that use CNNs and trained with this approach are referred to as Deep Q-Networks (Mnih et. al, 2015). It worked incredibly well, showing better than human-level performance in the game. For instance, the network learned that one optimal strategy in Atari is to break a hole on one side of the blocks-barrier and let the ball bounce inside between the ceiling and blocks-barrier so that it can hit many blocks and collect as many points as possible.

Figure 5: DQN playing Atari

After the success of DQN, DeepMind developed AlphaGo, which achieved superhuman performance in the game of Go and defeated world champion Lee Sedol. AlphaGo used a combination of model-based and model-free approaches. Namely, it uses a policy network to select moves and a value network to evaluate board positions and combines this with an algorithm called **Monte Carlo Tree Search (MCTS)** which essentially allows AlphaGo to plan for possible futures and select the best strategy (Silver et. al, 2016). Training the policy and value networks involved both supervised learning–using data from real games–and reinforcement learning through self-play.

AlphaGo relies heavily on knowledge of the game's rules and objectives. DeepMind researchers wanted to make an algorithm that can learn any environment or game on its own where the only knowledge given to the agent is all legal actions at the current state and if the game is over (someone won or it's a draw) but not the overall rules of the game. This was the challenge given to MuZero, which is an algorithm that combines the strengths of model-free and model-based RL by learning the environment's dynamics from scratch (Schrittwieser et. al, 2019). MuZero not only maintains a

policy network and value network, but it also maintains a dynamics model that predicts state transitions and rewards, even without knowing the exact rules of the environment. MuZero, just like AlphaGo, also simulates future states and plans using MCTS but it also predicts what these states actually look like according to the rules of the game it is playing. MuZero has successfully learned how to play several games including Go, Chess, and Shogi all without any prior knowledge of what game it is actually playing! This represents a significant step in RL, as it paves the way for applications in unknown or partially known environments.

VII. The Future

While RL has made significant strides, exciting recent developments in ML such as transformer-based NN architectures and the new era of large language models (LLMs) have captured much of the attention in today's AI research. Nevertheless, RL remains central to progress in fields that need adaptive, autonomous decision-making in uncertain situations. It continues to be refined for real-world applications across robotics, healthcare, and so on. RL is actually being used in LLMs as well, with algorithms like Proximal Policy Optimization (PPO) being used to fine tune the models. RL is also the central technique behind OpenAI's large reasoning model, o1, which was "trained with large-scale reinforcement learning to reason using chain-of-thought" (OpenAI, 2024).

Today, achieving generalization quickly is still seen as the biggest challenge in RL. It is especially important in non-ergodic problems: "How do you make sure that an agent gets enough experience to learn a high-performing policy, all the while not harming its environment, other agents, or itself?" (Sutton & Barto, 2018) In such scenarios, it's necessary to give the agent a well-defined reward function as well as come up with an effective exploration/exploitation strategy. Researchers have increasingly turned to supervised and unsupervised learning methods to solve problems that need generalization as they seem to capture the underlying distributions in the training data better than RL algorithms. Continued research in generalization methods will likely unlock RL's potential for broader applications and allow the field to move beyond specialized tasks like playing video games.

There is still a lot of work to be done in this subfield of ML. Its future lies in making agents more efficient, safe, adaptive, and able to generalize. Once strong generalization methods are achieved, RL will likely mature greatly, blend with other AI fields, expand into new domains, and play an incredibly pivotal role in AI's broader advancements.

References

Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). The MIT Press.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2013). *Playing Atari with deep reinforcement learning* (arXiv:1312.5602). arXiv. https://doi.org/10.48550/arXiv.1312.5602

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneerselvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature, 529*(7587), 484–489. https://doi.org/10.1038/nature16961

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., & Silver, D. (2019). *Mastering Atari, Go, Chess and Shogi by planning with a learned model* (arXiv:1911.08265). arXiv. https://doi.org/10.48550/arXiv.1911.08265

OpenAI. o1-system-card-20240917.pdf. Technical report, OpenAI, September 2024. URL https://cdn.openai.com/o1-system-card-20240917.pdf.