8

Rewards for Robots

When the journalist Amy Sutherland was doing research for a book on exotic animal trainers, she learned that their primary method is preposterously simple: “reward behavior I like and ignore behavior I don’t.” And as she wrote in *The New York Times*’ Modern Love column, “Eventually it hit me that the same techniques might work on that stubborn but lovable species, the American husband.” Sutherland wrote about how, after years of futile nagging, sarcasm, and resentment, she used this simple method to covertly train her oblivious husband to pick up his socks, find his own car keys, show up to restaurants on time, and shave more regularly.[1]

This classic training technique, known in psychology as operant conditioning, has been used for centuries on animals and humans. Operant conditioning inspired an important machine-learning approach called reinforcement learning. Reinforcement learning contrasts with the supervised-learning method I’ve described in previous chapters: in its purest form, reinforcement learning requires no labeled training examples. Instead, an *agent*—the learning program—performs *actions* in an *environment* (usually a computer simulation) and occasionally receives *rewards* from the environment. These intermittent rewards are the only feedback the agent uses for learning. In the case of Amy Sutherland’s husband, the rewards were her smiles, kisses, and words of praise. While a computer program might not respond to a kiss or an enthusiastic “you’re the greatest,” it can be made to respond to a machine equivalent of such appreciation—such as positive numbers added to its memory.

FIGURE 22: A Sony Aibo robotic dog, about to kick a robot soccer ball

While reinforcement learning has been part of the AI toolbox for decades, it has long been overshadowed by neural networks and other supervised-learning methods. This changed in 2016 when reinforcement learning played a central role in a stunning and momentous achievement in AI: a program that learned to beat the best humans at the complex game of Go. In order to explain that program, as well as other recent achievements of reinforcement learning, I'll first take you through a simple example to illustrate how reinforcement learning works.

Training Your Robo-Dog

For our illustrative example, let's look to the fun game of robot soccer, in which humans (usually college students) program robots to play a simplified version of soccer on a room-sized "field." Sometimes the players are cute doglike Aibo robots like the one shown in figure 22. An Aibo robot (made by Sony) has a camera to capture visual inputs, an internal programmable computer, and a collection of sensors and motors that enable it to walk, kick, head-butt, and even wag its plastic tail.

Imagine that we want to teach our robo-dog the simplest soccer skill: when facing the ball, walk over to it, and kick it. A traditional AI approach would be to program the robot with the following rules: Take a step toward the ball. Repeat until one of your feet is touching the ball. Then kick the ball with that foot. Of course, shorthand descriptions such as "take a step toward the ball," "until one of your feet is touching the ball," and "kick the ball"

must be carefully translated into detailed sensor and motor operations built into the Aibo.

Such explicit rules might be sufficient for a task as simple as this one. However, the more "intelligent" you want your robot to be, the harder it is to manually specify rules for behavior. And of course, it's impossible to devise a set of rules that will work in every situation. What if there is a large puddle between the robot and the ball? What if a soccer cone is blocking the robot's vision? What if a rock is blocking the ball's movement? As always, the real world is awash with hard-to-predict edge cases. The promise of reinforcement learning is that the agent—here our robo-dog—can learn flexible strategies on its own simply by performing actions in the world and occasionally receiving rewards (that is, *reinforcement*) without humans having to manually write rules or directly teach the agent every possible circumstance.

Let's call our robo-dog Rosie, after my favorite television robot, the wry robotic housekeeper from the classic cartoon *The Jetsons*.[2] To make things easier for this example, let's assume that Rosie comes from the factory preprogrammed with the following ability: if a soccer ball is in Rosie's line of sight, she can estimate the number of steps she would need to take to get to the ball. This number is called the "state." In general, the state of an agent at a given time is the agent's perception of its current situation. Rosie is the simplest of possible agents, in that her state is a single number. When I say that Rosie is "in" a given state x, I mean that she is currently estimating that she is x steps away from the ball.

In addition to being able to identify her state, Rosie has three built-in *actions* she can perform: she can take a step *Forward*, take a step *Backward*, and she can *Kick*. (If Rosie happens to step out-of-bounds, she is programmed to immediately step back in.) In the spirit of operant conditioning, let's give Rosie a reward only when she succeeds in kicking the ball. Note that Rosie doesn't know ahead of time which, if any, states or actions will lead to rewards.

Given that Rosie is a robot, her "reward" is simply a number, say, 10, added to her "reward memory." We can consider the number 10 the robot equivalent of a dog treat. Or perhaps not. Unlike a real dog, Rosie has no intrinsic *desire* for treats, positive numbers, or anything else. As I'll detail below, in reinforcement learning, a human-created algorithm guides Rosie's

process of learning in response to rewards; that is, the algorithm tells Rosie *how* to learn from her experiences.

Reinforcement learning occurs by having Rosie take actions over a series of learning *episodes*, each of which consists of some number of *iterations*. At each iteration, Rosie determines her current state and chooses an action to take. If Rosie receives a reward, she then *learns* something, as I'll illustrate below. Here I'll let each episode last until Rosie manages to kick the ball, at which time she receives a reward. This might take a long time. As in training a real dog, we have to be patient.

Figure 23 illustrates a hypothetical learning episode. The episode begins with the trainer (me) placing Rosie and the ball in some initial locations on the field, with Rosie facing the ball (figure 23*A*). Rosie determines her current state: twelve steps away from the ball. Because Rosie hasn't learned anything yet, our dog, an innocent "tabula rasa," doesn't know which action should be preferred, so she chooses an action at random from her three possibilities: *Forward*, *Backward*, *Kick*. Let's say she chooses *Backward* and takes a step back. We humans can see that *Backward* is a bad action to take, but remember, we're letting Rosie figure out on her own how to perform this task.

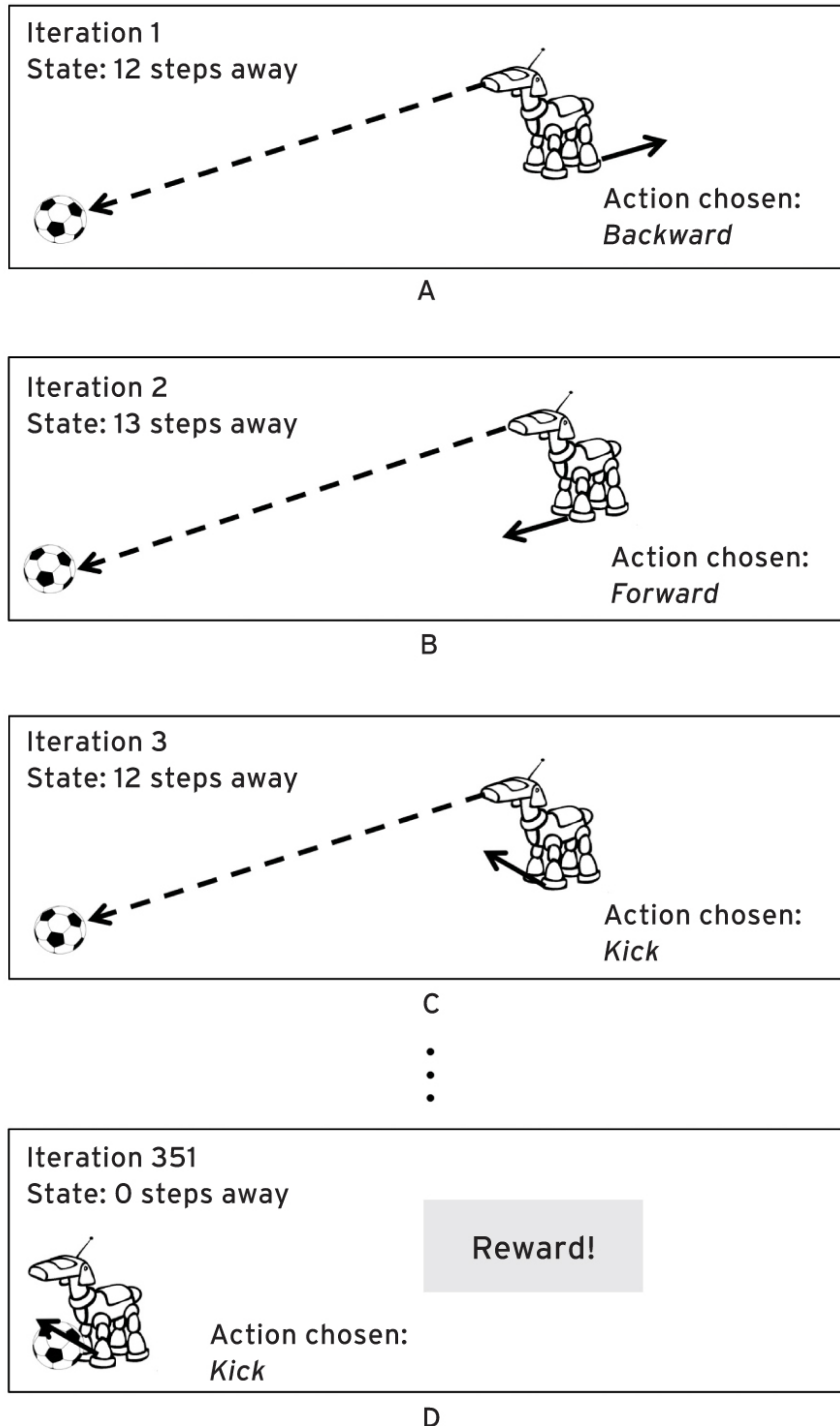

FIGURE 23: A hypothetical first episode of reinforcement learning

At iteration 2 (figure 23*B*), Rosie determines her new state: thirteen steps from the ball. She then chooses a new action to take, again at random: *Forward*. At iteration 3 (figure 23*C*), Rosie determines her "new" state:

twelve steps away from the ball. She's back to square one, but Rosie doesn't even know that she has been in this state before! In the purest form of reinforcement learning, the learning agent doesn't remember its previous states. In general, remembering previous states might take a lot of memory and doesn't turn out to be necessary.

At iteration 3, Rosie—again at random—chooses the action *Kick*, but because she's kicking empty air, she doesn't get a reward. She has yet to learn that kicking gives a reward only if she's next to the ball.

Rosie continues to choose random actions, without any feedback, for many iterations. But at some point, let's say at iteration 351, just by dumb luck Rosie ends up next to the ball and chooses *Kick* (figure 23*D*). Finally, she gets a reward and uses it to learn something.

What does Rosie learn? Here we take the simplest approach to reinforcement learning: upon receiving a reward, Rosie learns only about the state and action that immediately preceded the reward. In particular, Rosie learns that if she is in that state (for example, zero steps from the ball), taking that action (for example, *Kick*) is a good idea. But that's all she learns. She doesn't learn, for example, that if she is zero steps from the ball, *Backward* would be a *bad* choice. After all, she hasn't tried that yet. For all she knows, taking a step backward in that state might lead to a much bigger reward! Rosie also doesn't learn at this point that if she is *one* step away, *Forward* would be a good choice. She has to wait for the next episode for that. Learning too much at one time can be detrimental; if Rosie happens to kick the air two steps away from the ball, we don't want her to learn that this ineffective kick was actually a necessary step toward getting the reward. In humans, this kind of behavior might be called superstition—namely, erroneously believing that a particular action can help cause a particular good or bad outcome. In reinforcement learning, superstition is something that you have to be careful to avoid.

A crucial notion in reinforcement learning is that of the *value of performing a particular action in a given state*. The *value* of action *A* in state *S* is a number reflecting the agent's current prediction of how much reward it will eventually obtain if, when in state *S*, it performs action *A*, and then continues performing high-value actions. Let me explain. If your current state is "holding a chocolate in your hand," an action with high value would be to bring your hand to your mouth. Subsequent actions with high value

would be to open your mouth, put the chocolate inside, and chew. Your reward is the delicious sensation of eating the chocolate. Bringing your hand to your mouth doesn't immediately produce this reward, but this action is on the right path, and if you've eaten chocolate before, you can predict the intensity of the upcoming reward. The goal of reinforcement learning is for the agent to learn values that are good predictions of upcoming rewards (assuming that the agent keeps doing the right thing after taking the action in question).[3] As we'll see, the process of learning the values of particular actions in a given state typically takes many steps of trial and error.

Q-table:

State	0 steps away		1 step away		...	10 steps away		...
Action	*Forward*	0	*Forward*	0	...	*Forward*	0	...
	Backward	0	*Backward*	0		*Backward*	0	
	Kick	**10**	*Kick*	0		*Kick*	0	

FIGURE 24: Rosie's Q-table after her first episode of reinforcement learning

Rosie keeps track of the values of actions in a big table in her computer memory. This table, illustrated in figure 24, lists all the possible states for Rosie (that is, all possible distances she could be from the ball, up to the length of the field), and for each state, her possible actions. Given a state, each action in that state has a numerical value; these values will change—becoming more accurate predictions of upcoming rewards—as Rosie continues to learn. This table of states, actions, and values is called the Q-table. This form of reinforcement learning is sometimes called Q-learning. The letter *Q* is used because the letter *V* (for *value*) was used for something else in the original paper on Q-learning.[4]

At the beginning of Rosie's training, I initialize the Q-table by setting all the values to 0—a "blank slate." When Rosie receives a reward for kicking the ball at the end of episode 1, the value of the action *Kick* when in state "zero steps away" is updated to 10, the value of the reward. In the future, when Rosie is in the "zero steps away" state, she can look at the Q-table, see that *Kick* has the highest value—that is, it predicts the highest reward—and

decide to choose *Kick* rather than choosing randomly. That's all that "learning" means here!

Episode 1 ended with Rosie finally kicking the ball. We now move on to episode 2 (figure 25), which starts with Rosie and the ball in new locations (figure 25*A*). Just as before, at each iteration Rosie determines her current state—initially, six steps away—and chooses an action, now by looking in her Q-table. But at this point, the values of actions in her current state are still all 0s; there's no information yet to help her choose among them. So Rosie again chooses an action at random: *Backward*. And she chooses *Backward* again at the next iteration (figure 25*B*). Our robo-dog's training has a long way to go.

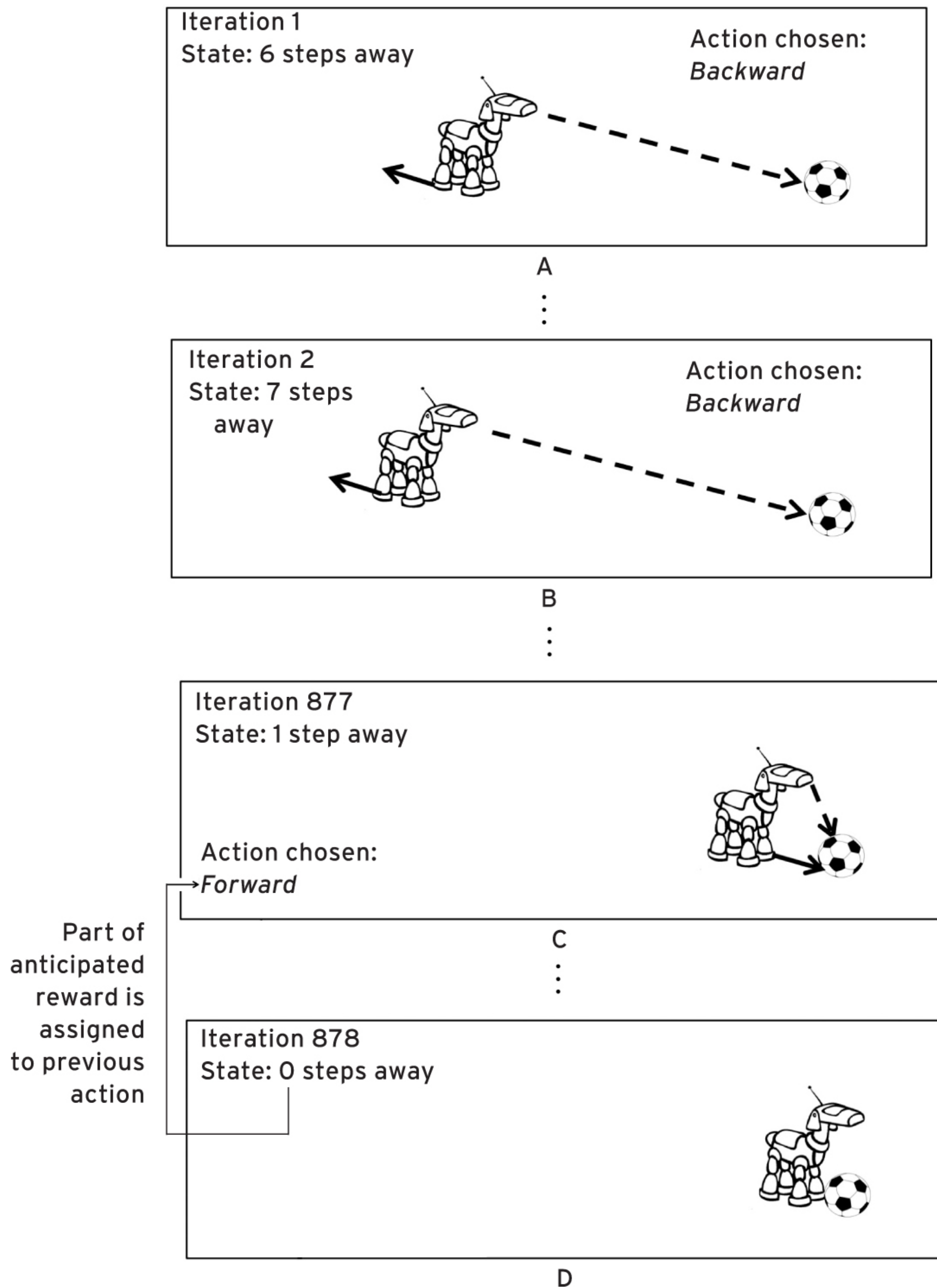

FIGURE 25: The second episode of reinforcement learning

Everything continues as before, until Rosie's floundering random trial-and-error actions happen to land her one step away from the ball (figure

25*C*), and she happens to choose *Forward*. Suddenly Rosie finds her foot next to the ball (figure 25*D*), and the Q-table has something to say about this state. In particular, it says that her current state—zero steps from the ball—has an action—*Kick*—that is predicted to lead to a reward of 10. Now she can use this information, learned at the previous episode, to choose an action to perform, namely *Kick*. But here's the essence of Q-learning: Rosie can now learn something about the action (*Forward*) she took in the immediately *previous* state (one step away). That is what led her to be in the excellent position she is in now! Specifically, the value of action *Forward* in the state "one step away" is updated in the Q-table to have a higher value, some fraction of the value of the action "*Kick* when zero steps away," which directly leads to a reward. Here I've updated this value to 8 (figure 26).

Q-table:

State	0 steps away		1 step away		...	10 steps away		...
Action	*Forward*	0	***Forward***	**8**	...	*Forward*	0	...
	Backward	0	*Backward*	0		*Backward*	0	
	Kick	**10**	*Kick*	0		*Kick*	0	

FIGURE 26: Rosie's Q-table after her second episode of reinforcement learning

The Q-table now tells Rosie that it's really good to kick when in the "zero steps away" state and that it's almost as good to step forward when in the "one step away" state. The next time Rosie finds herself in the "one step away" state, she'll have some information about what action she should take, as well as the ability to learn an update for the immediately past action—the *Forward* action in the "two steps away" state. Note that it is important for these learned action values to be reduced ("discounted") as they go back in time from the actual reward; this allows the system to learn an efficient path to an actual reward.

Reinforcement learning—here, the gradual updating of values in the Q-table—continues, episode to episode, until Rosie has finally learned to perform her task from any initial starting point. The Q-learning algorithm is a way to assign values to actions in a given state, including those actions that

don't lead directly to rewards but that set the stage for the relatively rare states in which the agent does receive rewards.

I wrote a program that simulated Rosie's Q-learning process as described above. At the beginning of each episode, Rosie was placed, facing the ball, a random number of steps away (with a maximum of twenty-five and a minimum of zero steps away). As I mentioned earlier, if Rosie stepped out-of-bounds, my program simply has her step back in. Each episode ended when Rosie succeeded in reaching and kicking the ball. I found that it took about three hundred episodes for her to learn to perform this task perfectly, no matter where she started.

This "training Rosie" example captures much of the essence of reinforcement learning, but I left out many issues that reinforcement-learning researchers face for more complex tasks.[5] For example, in real-world tasks, the agent's perception of its state is often uncertain, unlike Rosie's perfect knowledge of how many steps she is from the ball. A real soccer-playing robot might have only a rough estimate of distance, or even some uncertainty about which light-colored, small object on the soccer field is actually the ball. The effects of performing an action can also be uncertain: for example, a robot's *Forward* action might move it different distances depending on the terrain, or even result in the robot falling down or colliding with an unseen obstacle. How can reinforcement learning deal with uncertainties like these?

Additionally, how should the learning agent choose an action at each time step? A naive strategy would be to always choose the action with the highest value for the current state in the Q-table. But this strategy has a problem: it's possible that other, as-yet-unexplored actions will lead to a higher reward. How often should you explore—taking actions that you haven't yet tried—and how often should you choose actions that you already expect to lead to some reward? When you go to a restaurant, do you always order the meal you've already tried and found to be good, or do you try something new, because the menu might contain an even better option? Deciding how much to *explore* new actions and how much to *exploit* (that is, stick with) tried-and-true actions is called the exploration versus exploitation balance. Achieving the right balance is a core issue for making reinforcement learning successful.

These are samples of ongoing research topics among the growing community of people working on reinforcement learning. Just as in the field

of deep learning, designing successful reinforcement-learning systems is still a difficult (and sometimes lucrative!) art, mastered by a relatively small group of experts who, like their deep-learning counterparts, spend a lot of time tuning hyperparameters. (How many learning episodes should be allowed? How many iterations per episode should be allowed? How much should a reward be "discounted" as it is spread back in time? And so on.)

Stumbling Blocks in the Real World

Setting these issues aside for now, let's look at two major stumbling blocks that might arise in extrapolating our "training Rosie" example to reinforcement learning in real-world tasks. First, there's the Q-table. In complex real-world tasks—think, for example, of a robot car learning to drive in a crowded city—it's impossible to define a small set of "states" that could be listed in a table. A single state for a car at a given time would be something like the entirety of the data from its cameras and other sensors. This means that a self-driving car effectively faces an infinite number of possible states. Learning via a Q-table like the one in the "Rosie" example is out of the question. For this reason, most modern approaches to reinforcement learning use a neural network instead of a Q-table. The neural network's job is to learn what values should be assigned to actions in a given state. In particular, the network is given the current state as input, and its outputs are its estimates of the values of all the possible actions the agent can take in that state. The hope is that the network can learn to group related states into general concepts (*It's safe to drive forward* or *Stop immediately to avoid hitting an obstacle*).

The second stumbling block is the difficulty, in the real world, of actually carrying out the learning process over many episodes, using a real robot. Even our "Rosie" example isn't feasible. Imagine yourself initializing a new episode—walking out on the field to set up the robot and the ball—hundreds of times, not to mention waiting around for the robot to perform its hundreds of actions per episode. You just wouldn't have enough time. Moreover, you might risk the robot damaging itself by choosing the wrong action, such as kicking a concrete wall or stepping forward over a cliff.

Just as I did for Rosie, reinforcement-learning practitioners almost always deal with this problem by building *simulations* of robots and environments and performing all the learning episodes in the simulation rather than in the real world. Sometimes this approach works well. Robots have been trained using simulations to walk, hop, grasp objects, and drive a remote-control car, among other tasks, and the robots were able, with various levels of success, to transfer the skills learned during simulation to the real world.[6] However, the more complex and unpredictable the environment, the less successful are the attempts to transfer what is learned in simulation to the real world. Because of these difficulties, it makes sense that to date the greatest successes of reinforcement learning have been not in robotics but in domains that can be perfectly simulated on a computer. In particular, the best-known reinforcement-learning successes have been in the domain of game playing. Applying reinforcement learning to games is the topic of the next chapter.

9

Game On

Since the earliest days of AI, enthusiasts have been obsessed with creating programs that can beat humans at games. In the late 1940s, both Alan Turing and Claude Shannon, two founders of the computer age, wrote programs to play chess before there were even computers that could run their code. In the decades that followed, many a young game fanatic has been driven to learn to program in order to get computers to play their favorite game, whether it be checkers, chess, backgammon, Go, poker, or, more recently, video games.

In 2010, a young British scientist and game enthusiast named Demis Hassabis, along with two close friends, launched a company in London called DeepMind Technologies. Hassabis is a colorful and storied figure in the modern AI world. A chess prodigy who was winning championships by the age of six, he started programming video games professionally at fifteen and founded his own video game company at twenty-two. In addition to his entrepreneurial activities, he obtained a PhD in cognitive neuroscience from University College London in order to further his goal of building brain-inspired AI. Hassabis and his colleagues founded DeepMind Technologies in order to "tackle [the] really fundamental questions" about artificial intelligence.[1] Perhaps not surprisingly, the DeepMind group saw video games as the proper venue for tackling those questions. Video games are, in Hassabis's view, "like microcosms of the real world, but … cleaner and more constrained."[2]

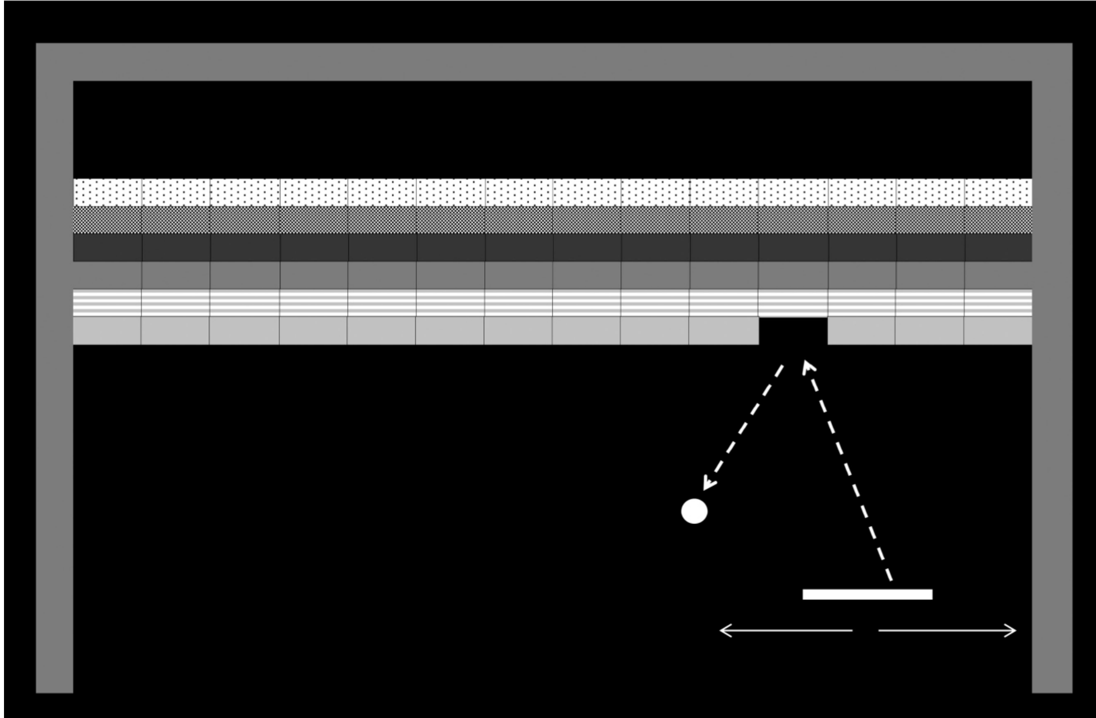

FIGURE 27: An illustration of Atari's *Breakout* game

Whatever your stance on video games, if you are going more for "clean and constrained" and less for "real world," you might consider creating AI programs to play Atari video games from the 1970s and '80s. This is exactly what the group at DeepMind decided to do. Depending on your age and interests, you might remember some of these classic games, such as *Asteroids*, *Space Invaders*, *Pong*, and *Ms. Pac-Man*. Are any of these ringing a bell? With their uncomplicated graphics and joystick controls, the games were easy enough for young children to learn but challenging enough to hold adults' interest.

Consider the single-player game called *Breakout*, illustrated in figure 27. The player uses the joystick to move a "paddle" (white rectangle at lower right) back and forth. A "ball" (white circle) can be bounced off the paddle to hit different-colored rectangular "bricks." The ball can also bounce off the gray "walls" at the sides. If the ball hits one of the bricks (patterned rectangles), the brick disappears, the player gains points, and the ball bounces back. Bricks in higher layers are worth more points than those in lower layers. If the ball hits the "ground" (bottom of the screen), the player

loses one of five "lives," and if any "lives" remain, a new ball shoots into play. The player's goal is to maximize the score over the five lives.

There's an interesting side note here. *Breakout* was the result of Atari's effort to create a single-player version of its successful game *Pong*. The design and implementation of *Breakout* were originally assigned in 1975 to a twenty-year-old employee named Steve Jobs. Yes, that Steve Jobs (later, cofounder of Apple). Jobs lacked sufficient engineering skills to do a good job on *Breakout*, so he enlisted his friend Steve Wozniak, aged twenty-five (later, the other cofounder of Apple), to help on the project. Wozniak and Jobs completed the hardware design of *Breakout* in four nights, starting work each night after Wozniak had completed his day job at Hewlett-Packard. Once released, *Breakout*, like *Pong*, was hugely popular among gamers.

If you're getting nostalgic but neglected to hang on to your old Atari 2600 game console, you can still find many websites offering *Breakout* and other games. In 2013, a group of Canadian AI researchers released a software platform called the Arcade Learning Environment that made it easy to test machine-learning systems on forty-nine of these games.[3] This was the platform used by the DeepMind group in their work on reinforcement learning.

Deep Q-Learning

The DeepMind group combined reinforcement learning—in particular Q-learning—with deep neural networks to create a system that could learn to play Atari video games. The group called their approach deep Q-learning. To explain how deep Q-learning works, I'll use *Breakout* as a running example, but DeepMind used the same method on all the Atari games they tackled. Things will get a bit technical here, so fasten your seat belt (or skip to the next section).

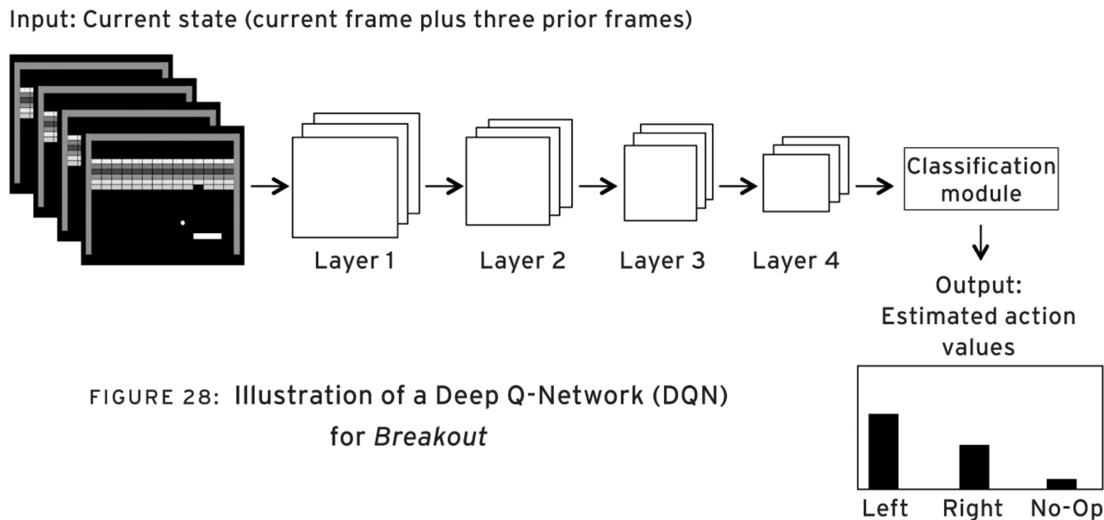

FIGURE 28: Illustration of a Deep Q-Network (DQN) for *Breakout*

FIGURE 28: Illustration of a Deep Q-Network (DQN) for *Breakout*

Recall how we used Q-learning to train Rosie the robo-dog. In an episode of Q-learning, at each iteration the learning agent (Rosie) does the following: it figures out its current state, looks up that state in the Q-table, uses the values in the table to choose an action, performs that action, possibly receives a reward, and—the learning step—updates the values in its Q-table.

DeepMind's deep Q-learning is exactly the same, except that a convolutional neural network takes the place of the Q-table. Following DeepMind, I'll call this network the Deep Q-Network (DQN). Figure 28 illustrates a DQN that is similar to (though simpler than) the one used by DeepMind for learning to play *Breakout*. The input to the DQN is the state of the system at a given time, which here is defined to be the current "frame"—the pixels of the current screen—plus three prior frames (screen pixels from three previous time steps). This definition of state provides the system with a small amount of memory, which turns out to be useful here. The outputs of the network are the estimated values for each possible action, given the input state. The possible actions are the following: move the paddle *Left*, move the paddle *Right*, and *No-Op* ("no operation," that is, don't move the paddle). The network itself is a ConvNet virtually identical to the one I described in chapter 4. Instead of the values in a Q-table, as we saw in the "Rosie" example, in deep Q-learning it is the *weights* in this network that are learned.

DeepMind's system learns to play *Breakout* over many episodes. Each episode corresponds to a play of the game, and each iteration during an episode corresponds to the system performing a single action. In particular,

at each iteration the system inputs its state to the DQN and chooses an action based on the DQN's output values. The system doesn't always choose the action with the highest estimated value; as I mentioned above, reinforcement learning requires a balance between exploration and exploitation.[4] The system performs its chosen action (for example, moving the paddle some amount to the left) and possibly receives a reward if the ball happens to hit one of the bricks. The system then performs a step of learning—that is, updating the weights in the DQN via back-propagation.

How are the weights updated? This is the crux of the difference between supervised learning and reinforcement learning. As you'll recall from earlier chapters, back-propagation works by changing a neural network's weights so as to reduce the *error* in the network's outputs. With supervised learning, measuring this error is straightforward. Remember our hypothetical ConvNet back in chapter 4 whose goal was to learn to classify photos as "dog" or "cat"? If an input training photo pictured a dog but the "dog" output confidence was only 20 percent, then the error for that output would be 100% − 20% = 80%; that is, ideally, the output should have been 80 points higher. The network could calculate the error because it had a *label* provided by a human.

However, in reinforcement learning we have no labels. A given frame from the game doesn't come labeled with the action that should be taken. How then do we assign an error to an output in this case?

Here's the answer. Recall that if you are the learning agent, the *value* of an action in the current state is your estimate of how much reward you will receive by the end of the episode, if you choose this action (and continue choosing high-value actions). This estimate should be better the closer you get to the end of the episode, when you can tally up the actual rewards you received! The trick is to assume that the network's outputs at the *current* iteration are closer to being correct than its outputs at the *previous iteration*. Then *learning* consists in adjusting the network weights (via back-propagation) so as to minimize the difference between the current and the previous iteration's outputs. Richard Sutton, one of the originators of this method, calls this "learning a guess from a guess."[5] I'll amend that to "learning a guess from a *better* guess."

In short, instead of learning to match its outputs to human-given labels, the network learns to make its outputs consistent from one iteration to the

next, assuming that later iterations give better estimates of value than earlier iterations. This learning method is called temporal difference learning.

To recap, here's how deep Q-learning works for the game of *Breakout* (and all the other Atari games). The system gives its current state as input to the Deep Q-Network. The Deep Q-Network outputs a value for each possible action. The system chooses and performs an action, resulting in a new state. Now the learning step takes place: the system inputs its new state to the network, which outputs a new set of values for each action. The difference between the new set of values and the previous set of values is considered the "error" of the network; this error is used by back-propagation to change the weights of the network. These steps are repeated over many episodes (plays of the game). Just to be clear, everything here—the Deep Q-Network, the virtual "joystick," and the game itself—is software running in a computer.

This is essentially the algorithm developed by DeepMind's researchers, although they used some tricks to improve it and speed it up.[6] At first, before much learning has happened, the network's outputs are quite random, and the system's game playing looks quite random as well. But gradually, as the network learns weights that improve its outputs, the system's playing ability improves, in many cases quite dramatically.

The $650 Million Agent

The DeepMind group applied their deep Q-learning method to the forty-nine different Atari games in the Arcade Learning Environment. While DeepMind's programmers used the same network architecture and hyperparameter settings for each game, their system learned each game from scratch; that is, the system's knowledge (the network weights) learned for one game was not transferred when the system started learning to play the next game. Each game required training for thousands of episodes, but this could be done relatively quickly on the company's advanced computer hardware.

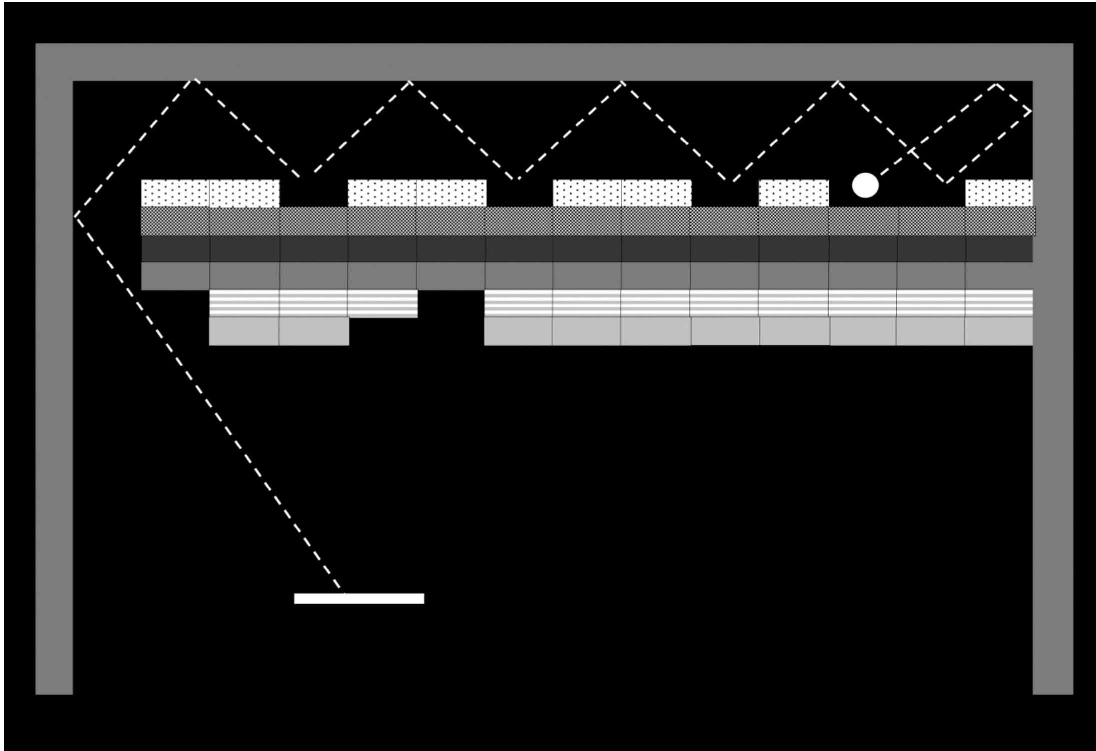

FIGURE 29: DeepMind's *Breakout* player discovered the strategy of tunneling through the bricks, which allowed it to quickly destroy high-value top bricks by bouncing off the "ceiling."

After a Deep Q-Network for each game was trained, DeepMind compared the machine's level of play with that of a human "professional games tester," who was allowed two hours of practice playing each game before being evaluated. Sound like a fun job? Only if you like being humiliated by a computer! DeepMind's deep Q-learning programs turned out to be better players than the human tester on more than half the games. And on half of *those* games, the programs were more than twice as good as the human. And on half of *those* games, the programs were more than five times better. One stunning example was on *Breakout*, where the DQN program scored on average more than *ten* times the human's average score.

What, exactly, did these superhuman programs learn to do? Upon investigation, DeepMind found that their programs had discovered some very clever strategies. For example, the trained *Breakout* program had discovered a devious trick, illustrated in figure 29. The program learned that if the ball was able to knock out bricks so as to build a narrow tunnel through the edge of the brick layer, then the ball would bounce back and forth between the

"ceiling" and the top of the brick layer, knocking out high-value top bricks very quickly without the player having to move the paddle at all.

DeepMind first presented this work in 2013 at an international machine-learning conference.[7] The audience was dazzled. Less than a year later, Google announced that it was acquiring DeepMind for £440 million (about $650 million at the time), presumably because of these results. Yes, reinforcement learning occasionally leads to big rewards.

With a lot of money in their pockets and the resources of Google behind them, DeepMind—now called Google DeepMind—took on a bigger challenge, one that had in fact long been considered one of AI's "grand challenges": creating a program that learns to play the game Go better than any human. DeepMind's program AlphaGo builds on a long history of AI in board games. Let's start with a brief survey of that history, which will help in explaining how AlphaGo works and why it is so significant.

Checkers and Chess

In 1949, the engineer Arthur Samuel joined IBM's laboratory in Poughkeepsie, New York, and immediately set about programming an early version of IBM's 701 computer to play checkers. If you yourself have any computer programming experience, you will appreciate the challenge he faced: as noted by one historian, "Samuel was the first person to do any serious programming on the 701 and as such had no system utilities [that is, essentially no operating system!] to call on. In particular he had no assembler and had to write everything using the op codes and addresses."[8] To translate for my nonprogrammer readers, this is something like building a house using only a handsaw and a hammer. Samuel's checkers-playing program was among the earliest machine-learning programs; indeed, it was Samuel who coined the term *machine learning*.

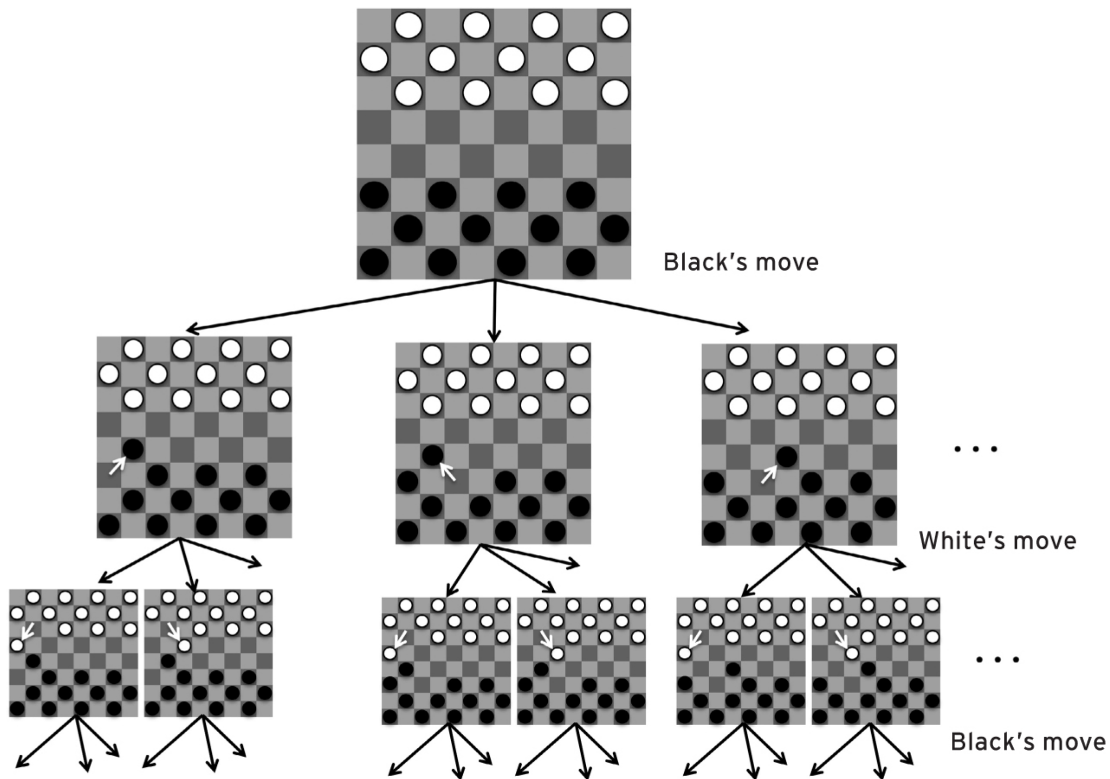

FIGURE 30: Part of a game tree for checkers. For simplicity, this figure shows only three possible moves from each board position. The white arrows point from a moved piece's previous square to its current square.

Samuel's checkers player was based on the method of searching a game tree, which is the basis of all programs for playing board games to this day (including AlphaGo, which I'll describe below). Figure 30 illustrates part of a game tree for checkers. The "root" of the tree (by convention drawn at the top, unlike the root of a natural tree) shows the initial checkerboard, before either player has moved. The "branches" from the root lead to all possible moves for the first player (here, Black). There are seven possible moves (for simplicity, the figure shows only three of these). For each of those seven moves for Black, there are seven possible response moves for White (not all shown in the figure), and so on. Each of the boards in figure 30, showing a possible arrangement of pieces, is called a board position.

Imagine yourself playing a game of checkers. At each turn, you might construct a small part of this tree in your mind. You might say to yourself, "If I make *this* move, then my opponent could make that move, in which case I could make that move, which will set me up for a jump." Most people, including the best players, consider only a few possible moves, looking

ahead only a few steps before choosing which move to make. A fast computer, on the other hand, has the potential to perform this kind of look-ahead on a much larger scale. What's stopping the computer from looking at every possible move and seeing which sequence of moves most quickly leads to a win? The problem is the same kind of exponential increase we saw back in chapter 3 (remember the king, the sage, and the grains of rice?). The average game of checkers has about fifty moves, which means that the game tree in figure 30 might extend down for fifty levels. At each level, there are on average six or seven branches from each possible board position. This means that the total number of board positions in the tree could be more than six raised to the fiftieth power—a ridiculously huge number. A hypothetical computer that could look at a trillion board positions per second would take more than 10^{19} years to consider all the board positions in a single game tree. (As is often done, we can compare this number with the age of the universe, which is merely on the order of 10^{10} years.) Clearly a complete search of the game tree is not feasible.

Fortunately, it's possible for computers to play well without doing this kind of exhaustive search. On each of its turns, Samuel's checkers-playing program created (in the computer's memory) a small part of a game tree like the one in figure 30. The root of the tree was the player's current board position, and the program, using its built-in knowledge of the rules of checkers, generated all the legal moves it could make from this current board position. It then generated all the legal moves that the opponent could make from each of the resulting positions, and so on, up to four or five turns (or "plies") of look-ahead.[9]

The program then evaluated board positions that appeared at the end of the look-ahead process; in figure 30, these would be the board positions in the bottom row in the partial tree. *Evaluating* a board position means assigning it a numerical value that estimates how likely it is to lead to a win for the program. Samuel's program used an *evaluation function* that gave points, thirty-eight in total, for various features of the board, such as Black's advantage in total number of pieces, Black's number of kings, and how many of Black's pieces were close to being kinged. These specific features were chosen by Samuel using his knowledge of checkers. Once each of the bottom-row board positions was thus evaluated, the program employed a classic algorithm, called minimax, which used these values—from the end of the

look-ahead process—in order to rate the program's immediate possible moves from its current board position. The program then chose the highest-rated move.

The intuition here is that the evaluation function will be more accurate when applied to board positions further along in the game; thus the program's strategy is to first look at all possible move sequences a few steps into the future and then apply the evaluation function to the resulting board positions. The evaluations are then propagated back up the tree by minimax, which produces a rating of all the possible immediate moves from the current board position.[10]

What the program learned was *which* features of the board should be included in the evaluation function at a given turn, as well as how to weight these different features when summing their points. Samuel experimented with several methods for learning in his system. In the most interesting version, the system learned while playing itself! The method for learning was somewhat complicated, and I won't detail it here, but it had some aspects that foreshadowed modern reinforcement learning.[11]

In the end, Samuel's checkers player impressively rose to the level of a "better-than-average player," though by no means a champion. It was characterized by some amateur players as "tricky but beatable."[12] But notably, the program was a publicity windfall for IBM: the day after Samuel demonstrated it on national television in 1956, IBM's stock price rose by fifteen points. This was the first of several times IBM saw its stock price increase after a demonstration of a game-playing program beating humans; as a more recent example, IBM's stock price similarly rose after the widely viewed TV broadcasts in which its Watson program won in the game show *Jeopardy!*

While Samuel's checkers player was an important milestone in AI history, I made this historical digression primarily to introduce three all-important concepts that it illustrates: the game tree, the evaluation function, and learning by self-play.

Deep Blue

Although Samuel's "tricky but beatable" checkers program was remarkable, especially for its time, it hardly challenged people's idea of themselves as uniquely intelligent. Even if a machine could win against human checkers champions (as one finally did in 1994[13]), mastering the game of checkers was never seen as a proxy for general intelligence. Chess is a different story. In the words of DeepMind's Demis Hassabis, "For decades, leading computer scientists believed that, given the traditional status of chess as an exemplary demonstration of human intellect, a competent computer chess player would soon also surpass all other human abilities."[14] Many people, including the early pioneers of AI Allen Newell and Herbert Simon, professed this exalted view of chess; in 1958 Newell and Simon wrote, "If one could devise a successful chess machine, one would seem to have penetrated to the core of human intellectual endeavor."[15]

Chess is significantly more complex than checkers. For example, I said above that in checkers there are, on average, six or seven possible moves from any given board position. In contrast, chess has on average thirty-five moves from any given board position. This makes the chess game tree enormously larger than that of checkers. Over the decades, chess-playing programs kept improving, in lockstep with improvements in the speed of computer hardware. In 1997, IBM had its second big game-playing triumph with Deep Blue, a chess-playing program that beat the world champion Garry Kasparov in a widely broadcast multigame match.

Deep Blue used much the same method as Samuel's checkers player: at a given turn, it created a partial game tree using the current board position as the root; it applied its evaluation function to the furthest layer in the tree and then used the minimax algorithm to propagate the values up the tree in order to determine which move it should make. The major differences between Samuel's program and Deep Blue were Deep Blue's deeper look-ahead in its game tree, its more complex (chess-specific) evaluation function, hand-programmed chess knowledge, and specialized parallel hardware to make it run very fast. Furthermore, unlike Samuel's checkers-playing program, Deep Blue did not use machine learning in any central way.

Like Samuel's checkers player before it, Deep Blue's defeat of Kasparov spurred a significant increase in IBM's stock price.[16] This defeat also generated considerable consternation in the media about the implications for superhuman intelligence as well as doubts about whether

humans would still be motivated to play chess. But in the decades since Deep Blue, humanity has adapted. As Claude Shannon wrote presciently in 1950, a machine that can surpass humans at chess "will force us either to admit the possibility of mechanized thinking or to further restrict our concept of thinking."[17] The latter happened. Superhuman chess playing is now seen as something that doesn't require general intelligence. Deep Blue isn't intelligent in any sense we mean today. It can't do anything but play chess, and it doesn't have any conception of what "playing a game" or "winning" means to humans. (I once heard a speaker say, "Deep Blue may have beat Kasparov, but it didn't get any joy out of it.") Moreover, chess has survived —even prospered—as a challenging human activity. Nowadays, computer-chess programs are used by human players as a kind of training aid, in the way a baseball player might practice using a pitching machine. Is this a result of our evolving notion of intelligence, which advances in AI help to clarify? Or is it another example of John McCarthy's maxim: "As soon as it works, no one calls it AI anymore"?[18]

The Grand Challenge of Go

The game of Go has been around for more than two thousand years and is considered among the most difficult of all board games. If you're not a Go player, don't worry; none of my discussion here will require any prior knowledge of the game. But it's useful to know that the game has serious status, especially in East Asia, where it is extremely popular. "Go is a pastime beloved by emperors and generals, intellectuals and child prodigies," writes the scholar and journalist Alan Levinovitz, who goes on to quote the South Korean Go champion Lee Sedol: "There is chess in the western world, but Go is incomparably more subtle and intellectual."[19]

Go is a game that has fairly simple rules but produces what you might call emergent complexity. At each turn, a player places a piece of his or her color (black or white) on a nineteen-by-nineteen-square board, following rules for where pieces may be placed and how to capture one's opponent's pieces. Unlike chess, with its hierarchy of pawns, bishops, queens, and so on, pieces in Go ("stones") are all equal. It's the configuration of stones on the board that a player must quickly analyze to decide on a move.

Creating a program to play Go well has been a focus of AI since the field's early days. However, Go's complexity made this task remarkably hard. In 1997, the same year Deep Blue beat Kasparov, the best Go programs could still be easily defeated by average players. Deep Blue, you'll recall, was able to do a significant amount of look-ahead from any board position and then use its evaluation function to assign values to future board positions, where each value predicted whether a particular board position would lead to a win. Go programs are not able to use this strategy for two reasons. First, the size of a look-ahead tree in Go is dramatically larger than that in chess. Whereas a chess player must choose from on average 35 possible moves from a given board position, a Go player has on average 250 such possibilities. Even with special-purpose hardware, a Deep Blue–style brute-force search of the Go game tree is just not feasible. Second, no one has succeeded in creating a good evaluation function for Go board positions. That is, no one has been able to construct a successful formula that examines a board position in Go and predicts who is going to win. The best (human) Go players rely on their pattern-recognition skills and their ineffable "intuition."

AI researchers haven't yet figured out how to encode intuition into an evaluation function. This is why, in 1997, the same year that Deep Blue beat Kasparov, the journalist George Johnson wrote in *The New York Times*, "When or if a computer defeats a human Go champion, it will be a sign that artificial intelligence is truly beginning to become as good as the real thing."[20] This may sound familiar—just like what people used to say about chess! Johnson quoted one Go enthusiast's prediction: "It may be a hundred years before a computer beats humans at Go—maybe even longer." A mere twenty years later, AlphaGo, which learned to play Go via deep Q-learning, beat Lee Sedol in a five-game match.

AlphaGo Versus Lee Sedol

Before I explain how AlphaGo works, let's first commemorate its spectacular wins against Lee Sedol, one of the world's best Go players. Even after watching AlphaGo defeat the then European Go champion Fan Hui half a year earlier, Lee remained confident that he would prevail: "I think

[AlphaGo's] level doesn't match mine.… Of course, there would have been many updates in the last four or five months, but that isn't enough time to challenge me."[21]

Perhaps you were one of the more than two hundred million people who watched some part of the AlphaGo-Lee match online in March 2016. I'm certain that this ranks as the largest audience by far for any Go match in the game's twenty-five-hundred-year history. After the first game, you might have shared Lee's reaction at his loss to the program: "I am in shock, I admit that. … I didn't think AlphaGo would play the game in such a perfect manner."[22]

AlphaGo's "perfect" play included many moves that evoked surprise and admiration among the match's human commentators. But partway through game 2, AlphaGo made a single move that gobsmacked even the most advanced Go experts. As *Wired* reported,

> At first, Fan Hui [the aforementioned European Go champion] thought the move was rather odd. But then he saw its beauty. "It's not a human move. I've never seen a human play this move," he says. "So beautiful." It's a word he keeps repeating. Beautiful. Beautiful. Beautiful. … "That's a very surprising move," said one of the match's English language commentators, who is himself a very talented Go player. Then the other chuckled and said: "I thought it was a mistake." But perhaps no one was more surprised than Lee Sedol, who stood up and left the match room. "He had to go wash his face or something, just to recover," said the first commentator.[23]

Of this same move, *The Economist* noted, "Intriguingly, moves like these are sometimes made by human Go masters. They are known in Japanese as *kami no itte* ('the hand of God,' or 'divine moves')."[24]

AlphaGo won that game, and the next. But in game 4, Lee had his own *kami no itte* moment, one that captures the intricacy of the game and the intuitive power of the top players. Lee's move took the commentators by surprise, but they immediately recognized it as potentially lethal for Lee's opponent. One writer noted, "AlphaGo, however, didn't seem to realize what was happening. This wasn't something it had encountered … in the millions and millions of games it had played with itself. At the post-match press conference Sedol was asked what he had been thinking when he played it. It was, he said, the only move he had been able to see."[25]

AlphaGo lost game 4 but came back to win game 5 and thus the match. In the popular media, it was Deep Blue versus Kasparov all over again, with an

endless supply of think pieces on what AlphaGo's triumph meant for the future of humanity. But this was even more significant than Deep Blue's win: AI had surmounted an even greater challenge than chess and had done so in a much more impressive fashion. Unlike Deep Blue, AlphaGo acquired its abilities by reinforcement learning via self-play.

Demis Hassabis noted that "the thing that separates out top Go players [is] their intuition" and that "what we've done with AlphaGo is to introduce with neural networks this aspect of intuition, if you want to call it that."[26]

How AlphaGo Works

There have been several different versions of AlphaGo, so to keep them straight, DeepMind started naming them after the human Go champions the programs had defeated—AlphaGo Fan and AlphaGo Lee—which to me evoked the image of the skulls of vanquished enemies in the collection of a digital Viking. Not what DeepMind intended, I'm sure. In any case, AlphaGo Fan and AlphaGo Lee both used an intricate mix of deep Q-learning, "Monte Carlo tree search," supervised learning, and specialized Go knowledge. But a year after the Lee Sedol match, DeepMind developed a version of the program that was both simpler than and superior to the previous versions. This newer version is called AlphaGo Zero because, unlike its predecessor, it started off with "zero" knowledge of Go besides the rules.[27] In a hundred games of AlphaGo Lee versus AlphaGo Zero, the latter won every single game. Moreover, DeepMind applied the same methods (though with different networks and different built-in game rules) to learn to play both chess and *shogi* (also known as Japanese chess).[28] The authors called the collection of these methods AlphaZero. In this section, I'll describe how AlphaGo Zero worked, but for conciseness I'll simply refer to this version as AlphaGo.

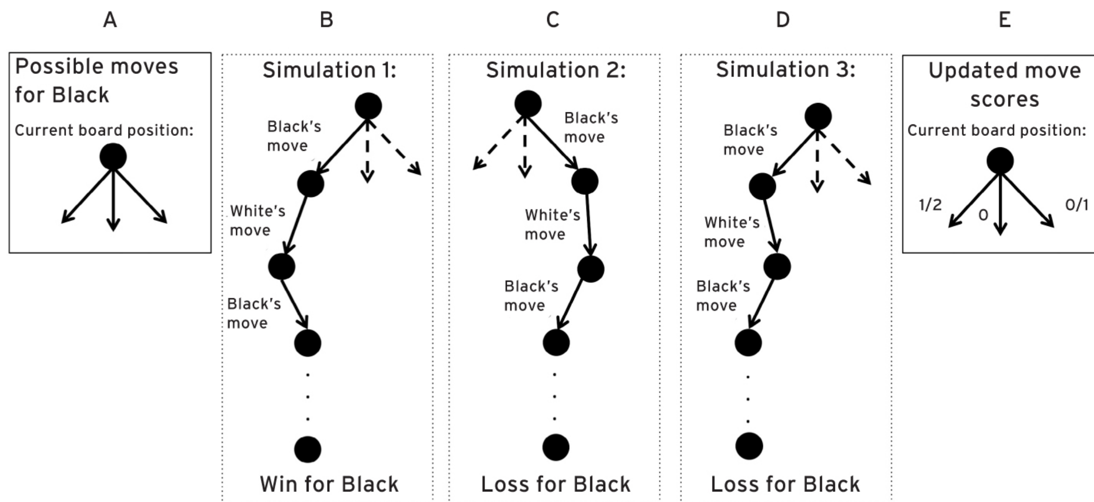

FIGURE 31: An illustration of Monte Carlo tree search

The word *intuition* has an aura of mystery, but AlphaGo's intuition (if you want to call it that) arises from its combination of deep Q-learning with a clever method called "Monte Carlo tree search." Let's take a moment to unpack that cumbersome name. First, the "Monte Carlo" part. Monte Carlo is, of course, the most glamorous part of the tiny Principality of Monaco, on the French Riviera, famous for its jet-setter casinos, car racing, and frequent appearance in James Bond movies. But in science and mathematics, "Monte Carlo" refers to a family of computer algorithms, the so-called Monte Carlo method, which was first used during the Manhattan Project to help design the atomic bomb. The name comes from the idea that a degree of *randomness*—like that of the iconic spinning roulette wheel in the Monte Carlo Casino—can be used by a computer to solve difficult mathematical problems.

Monte Carlo tree search is a version of the Monte Carlo method specifically devised for computer game-playing programs. Similar to the way Deep Blue's evaluation function worked, Monte Carlo tree search is used to assign a score to each possible move from a given board position. However, as I explained above, using extensive look-ahead in the game tree is not feasible for Go, and no one has been able to come up with a good evaluation function for board positions in Go. Monte Carlo tree search works differently.

Figure 31 illustrates Monte Carlo tree search. First, look at figure 31*A*. The black circle represents the current board position—that is, the configuration of pieces on the board at the current turn. Assume our Go-

playing program is playing Black, and it is Black's move. Let's assume for simplicity that there are three possible moves for Black, represented by the three arrows. Which move should Black choose?

If Black had enough time, it could do a "full search" of the game tree: look ahead at *all* the possible sequences of moves that could be played and choose a move that gives the best chance of leading to a win for Black. But doing this exhaustive look-ahead isn't possible; as I mentioned earlier, even all the time since the beginning of the universe isn't nearly enough to do a full tree search in Go. With Monte Carlo tree search, Black looks ahead at only a minuscule fraction of the possible sequences that could arise from each move, counts how many wins and losses those hypothetical sequences lead to, and uses those counts to give a score to each of its possible moves. The roulette-wheel-inspired randomness is used in deciding how to do the look-ahead.

More specifically, in order to choose a move from its current position, Black "imagines" (that is, simulates) several possible ways the game could play out, as illustrated in figure 31*B–D*. In each of these simulations, Black starts at its current position, randomly chooses one of its possible moves, then (from the new board position) randomly chooses a move for its opponent (White), and so on, continuing until the simulated game ends up in a win or loss for Black. Such a simulation, starting from a given board position, is called a roll-out from that position.

In the figure, you can see that in the three roll-outs, Black won once and lost twice. Black can now assign a score to each possible move from its current board position (figure 31*E*). Move 1 (leftmost arrow) participated in two roll-outs, one of which ended in a win, so that move's score is 1 out of 2. Move 3 (rightmost arrow) participated in one roll-out, which ended in a loss, so its score is 0 out of 1. The center move was not tried at all, so its score is set to 0. Moreover, the program keeps similar statistics on all the intermediate moves that participated in the roll-outs. Once this round of Monte Carlo tree search has finished, the program can use its updated scores to decide which of its possible moves seems the most promising—here, move 1. The program can then make that move in the actual game.

When I said before that during a roll-out the program chooses moves for itself and its opponents at random, what actually happens is that the program chooses moves *probabilistically* based on any scores that those moves might

have from previous rounds of Monte Carlo tree search. When each roll-out finishes with a win or loss, the algorithm updates all the scores of moves it made during that game to reflect the win or loss.

At first, the program's choice of moves from a given board position is quite random (the program is doing the equivalent of spinning a roulette wheel to choose a move), but as the program performs additional roll-outs, generating additional statistics, it is increasingly biased to choose those moves that in past roll-outs led to the most wins.

In this way, Monte Carlo tree search doesn't have to guess from just looking at a board position which move is most likely to lead to a win; it uses its roll-outs to collect statistics on how many times a given move *actually* leads to a win or loss. The more roll-outs the algorithm runs, the better its statistics. As before, the program needs to balance exploitation (choosing the highest-scoring moves during a roll-out) with exploration (sometimes choosing lower-scoring moves for which the program doesn't yet have much statistics). In figure 31, I showed three roll-outs; AlphaGo's Monte Carlo tree search performed close to two thousand roll-outs per turn.

The computer scientists at DeepMind didn't invent Monte Carlo tree search. It was first proposed in the context of game trees in 2006, and it resulted in a very big improvement in the ability of computer Go programs. But these programs still couldn't beat the best humans. One problem was that generating sufficient statistics from roll-outs can take a lot of time, especially in Go, with its vast number of possible moves. The DeepMind group realized that they could improve their system by complementing Monte Carlo tree search with a deep convolutional neural network. Given the current board position as input, AlphaGo uses a trained deep convolutional neural network to assign a rough value to all possible moves from the current position. Then Monte Carlo tree search uses those values to kick-start its search: rather than initially choosing moves at random, Monte Carlo tree search uses values output by the ConvNet as an indicator of which initial moves should be preferred. Imagine that you are AlphaGo staring at a board position: before you start the Monte Carlo process of performing roll-outs from that position, the ConvNet is whispering in your ear which of the possible moves from your current position are probably the best ones.

Conversely, the results of Monte Carlo tree search feed back to train the ConvNet. Imagine yourself as AlphaGo after a Monte Carlo tree search. The

results of your search are new probabilities assigned to all your possible moves, based on how many times those moves led to wins or losses during the roll-outs you performed. These new probabilities are now used to correct your ConvNet's output, via back-propagation. You and your opponent then choose moves, as a result of which you have a new board position, and the process continues. In principle, the convolutional neural network will learn to recognize patterns, just as Go masters do. Eventually, the ConvNet will play the role of the program's "intuition," which is further improved by Monte Carlo tree search.

Like its ancestor, Samuel's checkers player, AlphaGo learns by playing against *itself* over many games (about five million). During its training, the convolutional neural network's weights are updated after each move based on the difference between the network's output values and the improved values after Monte Carlo tree search is run. Then, when it's time for AlphaGo to play, say, a human like Lee Sedol, the trained ConvNet is used at each turn to generate values to help Monte Carlo tree search get started.

With its AlphaGo project, DeepMind demonstrated that one of AI's longtime grand challenges could be conquered by an inventive combination of reinforcement learning, convolutional neural networks, and Monte Carlo tree search (and adding powerful modern computing hardware to the mix). As a result, AlphaGo has attained a well-deserved place in the AI pantheon. But what's next? Will this potent combination of methods generalize beyond the world of game playing? This is the question I discuss in the next chapter.

8: Rewards for Robots

1. A. Sutherland, “What Shamu Taught Me About a Happy Marriage,” *New York Times*, June 25, 2006, www.nytimes.com/2006/06/25/fashion/what-shamu-taught-me-about-a-happy-marriage.html.
2. thejetsons.wikia.com/wiki/Rosey.
3. To be more precise, this approach to reinforcement learning, called value learning, is not the only possible approach. A second approach, called policy learning, has the goal of learning directly what action to perform in a given state, rather than first learning the numerical *values* of actions.
4. C. J. Watkins and P. Dayan, “Q-Learning,” *Machine Learning* 8, nos. 3–4 (1992): 279–92.
5. For a detailed, technical introduction to reinforcement learning, see R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. (Cambridge, Mass.: MIT Press, 2017), incompleteideas.net/book/the-book-2nd.html.
6. For example, see the following papers: P. Christiano et al., “Transfer from Simulation to Real World Through Learning Deep Inverse Dynamics Model,” arXiv:1610.03518 (2016); J. P. Hanna and P. Stone, “Grounded Action Transformation for Robot Learning in Simulation,” in *Proceedings of the Conference of the American Association for Artificial Intelligence* (2017), 3834–40; A. A. Rusu et al., “Sim-to-Real Robot Learning from Pixels with Progressive Nets,” in *Proceedings of the First Annual Conference on Robot Learning, CoRL* (2017); S. James, A. J. Davison, and E. Johns, “Transferring End-to-End Visuomotor Control from Simulation to Real World for a Multi-stage Task,” in *Proceedings of the First Annual Conference on Robot Learning, CoRL* (2017); M. Cutler, T. J. Walsh, and J. P. How, “Real-World Reinforcement Learning via Multifidelity Simulators,” *IEEE Transactions on Robotics* 31, no. 3 (2015): 655–71.

9: Game On

1. Demis Hassabis, quoted in P. Iwaniuk, “A Conversation with Demis Hassabis, the Bullfrog AI Prodigy Now Finding Solutions to the World’s Big Problems,” *PCGamesN*, accessed Dec. 7, 2018, www.pcgamesn.com/demis-hassabis-interview.
2. Quoted in “From Not Working to Neural Networking,” *Economist*, June 25, 2016.
3. M. G. Bellemare et al., “The Arcade Learning Environment: An Evaluation Platform for General Agents,” *Journal of Artificial Intelligence Research* 47 (2013): 253–79.
4. More technically, DeepMind’s program used what is called an epsilon-greedy method for choosing an action at each time step. With probability *epsilon* the program chooses an action at random; with probability (1 − *epsilon*) the program chooses the action with the highest value. *Epsilon* is a value between 0 and 1; it is initially set close to 1 and is gradually decreased over the episodes of training.
5. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. (Cambridge, Mass.: MIT Press, 2017), 124, incompleteideas.net/book/the-book-2nd.html.
6. For more details, see V. Mnih et al., “Human-Level Control Through Deep Reinforcement Learning,” *Nature* 518, no. 7540 (2015): 529.
7. V. Mnih et al., “Playing Atari with Deep Reinforcement Learning,” *Proceedings of the Neural Information Processing Systems (NIPS) Conference, Deep Learning Workshop* (2013).
8. “Arthur Samuel,” History of Computers website, history-computer.com/ModernComputer/thinkers/Samuel.html.
9. Samuel’s program used a variable number of plies, depending on the move.
10. Samuel’s program also used a method called alpha-beta pruning at each turn to determine nodes in the game tree that did not need to be evaluated. Alpha-beta pruning was also an essential part of IBM’s Deep Blue chess-playing program.
11. For details, see A. L. Samuel, “Some Studies in Machine Learning Using the Game of Checkers,” *IBM Journal of Research and Development* 3, no. 3 (1959): 210–29.
12. Ibid.
13. J. Schaeffer et al., “CHINOOK: The World Man-Machine Checkers Champion,” *AI Magazine* 17, no. 1 (1996): 21.
14. D. Hassabis, “Artificial Intelligence: Chess Match of the Century,” *Nature* 544 (2017): 413–14.
15. A. Newell, J. Calman Shaw, and H. A. Simon, “Chess-Playing Programs and the Problem of Complexity,” *IBM Journal of Research and Development* 2, no. 4 (1958): 320–35.
16. M. Newborn, *Deep Blue: An Artificial Intelligence Milestone* (New York: Springer, 2003), 236.
17. Quoted in J. Goldsmith, “The Last Human Chess Master,” *Wired*, Feb. 1, 1995.
18. Quoted in M. Y. Vardi, “Artificial Intelligence: Past and Future,” *Communications of the Association for Computing Machinery* 55, no. 1 (2012): 5.
19. A. Levinovitz, “The Mystery of Go, the Ancient Game That Computers Still Can’t Win,” *Wired*, May 12, 2014.
20. G. Johnson, “To Test a Powerful Computer, Play an Ancient Game,” *New York Times*, July 29, 1997.
21. Quoted in “S. Korean Go Player Confident of Beating Google’s AI,” Yonhap News Agency, Feb. 23, 2016, english.yonhapnews.co.kr/search1/2603000000.html?cid=AEN20160223003651315.
22. Quoted in M. Zastrow, “‘I’m in Shock!’: How an AI Beat the World’s Best Human at Go,” *New Scientist*, March 9, 2016, www.newscientist.com/article/2079871-im-in-shock-how-an-ai-beat-the-worlds-best-human-at-go.
23. C. Metz, “The Sadness and Beauty of Watching Google’s AI Play Go,” *Wired*, March 11, 2016, www.wired.com/2016/03/sadness-beauty-watching-googles-ai-play-go.

24. "For Artificial Intelligence to Thrive, It Must Explain Itself," *Economist*, Feb. 15, 2018, www.economist.com/news/science-and-technology/21737018-if-it-cannot-who-will-trust-it-artificial-intelligence-thrive-it-must.
25. P. Taylor, "The Concept of 'Cat Face,'" *London Review of Books*, Aug. 11, 2016.
26. Quoted in S. Byford, "DeepMind Founder Demis Hassabis on How AI Will Shape the Future," *Verge*, March 10, 2016, www.theverge.com/2016/3/10/11192774/demis-hassabis-interview-alphago-google-deepmind-ai.
27. D. Silver et al., "Mastering the Game of Go Without Human Knowledge," *Nature*, 550 (2017): 354–59.
28. D. Silver et al., "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play," *Science* 362, no. 6419 (2018): 1140–44.