Chapter 9

Deep Reinforcement Learning

"The reward of suffering is experience."—Harry S. Truman

9.1 Introduction

Human beings do not learn from a concrete notion of training data. Learning in humans is a continuous experience-driven process in which decisions are made, and the reward/punishment received from the *environment* are used to guide the learning process for future decisions. In other words, learning in intelligent beings is by reward-guided *trial and error*. Furthermore, much of human intelligence and instinct is encoded in genetics, which has evolved over millions of years with another environment-driven process, referred to as *evolution*. Therefore, almost all of biological intelligence, as we know it, originates in one form or other through an interactive process of trial and error with the environment. In his interesting book on artificial intelligence [453], Herbert Simon proposed the *ant hypothesis*:

> "Human beings, viewed as behaving systems, are quite simple. The apparent complexity of our behavior over time is largely a reflection of the complexity of the environment in which we find ourselves."

Human beings are considered simple because they are one-dimensional, selfish, and reward-driven entities (when viewed as a whole), and all of biological intelligence is therefore attributable to this simple fact. Since the goal of artificial intelligence is to simulate biological intelligence, it is therefore natural to draw inspirations from the successes of biological greed in simplifying the design of highly complex learning algorithms.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_9

A reward-driven trial-and-error process, in which a system learns to interact with a complex environment to achieve rewarding outcomes, is referred to in machine learning parlance as *reinforcement learning*. In reinforcement learning, the process of trial and error is driven by the need to maximize the expected rewards over time. Reinforcement learning can be a gateway to the quest for creating truly intelligent *agents* such as game-playing algorithms, self-driving cars, and even intelligent robots that interact with the environment. Simply speaking, it is a gateway to general forms of artificial intelligence. We are not quite there yet. However, we have made huge strides in recent years with exciting results:

1. Deep learners have been trained to play video games by using only the raw pixels of the video console as feedback. A classical example of this setting is the Atari 2600 console, which is a platform supporting multiple games. The input to the deep learner from the Atari platform is the display of pixels from the current state of the game. The reinforcement learning algorithm predicts the actions based on the display and inputs them into the Atari console. Initially, the computer algorithm makes many mistakes, which are reflected in the virtual rewards given by the console. As the learner gains experience from its mistakes, it makes better decisions. This is exactly how humans learn to play video games. The performance of a recent algorithm on the Atari platform has been shown to surpass human-level performance for a large number of games [165, 335, 336, 432]. Video games are excellent test beds for reinforcement learning algorithms, because they can be viewed as highly simplified representations of the choices one has to make in various decision-centric settings. Simply speaking, video games represent toy microcosms of real life.

2. DeepMind has trained a deep learning algorithm *AlphaGo* [445] to play the game of *Go* by using the reward-outcomes in the moves of games drawn from both human and computer self-play. *Go* is a complex game that requires significant human intuition, and the large tree of possibilities (compared to other games like chess) makes it an incredibly difficult candidate for building a game-playing algorithm. *AlphaGo* has not only convincingly defeated all top-ranked *Go* players it has played against [602, 603], but has contributed to innovations in the style of human play by using unconventional strategies in defeating these players. These innovations were a result of the reward-driven experience gained by *AlphaGo* by playing itself over time. Recently, the approach has also been generalized to chess, and it has convincingly defeated one of the top conventional engines [447].

3. In recent years, deep reinforcement learning has been harnessed in self-driving cars by using the feedback from various sensors around the car to make decisions. Although it is more common to use supervised learning (or *imitation learning*) in self-driving cars, the option of using reinforcement learning has always been recognized as a viable possibility [604]. During the course of driving, these cars now consistently make fewer errors than do human beings.

4. The quest for creating self-learning robots is a task in reinforcement learning [286, 296, 432]. For example, robot locomotion turns out to be surprisingly difficult in nimble configurations. Teaching a robot to walk can be couched as a reinforcement learning task, if we do not show a robot what walking looks like. In the reinforcement learning paradigm, we only incentivize the robot to get from point A to point B as efficiently as possible using its available limbs and motors [432]. Through reward-guided trial and error, robots learn to roll, crawl, and eventually walk.

Reinforcement learning is appropriate for tasks *that are simple to evaluate but hard to specify*. For example, it is easy to evaluate a player's performance at the end of a complex game like chess, but it is hard to specify the precise action in every situation. As in biological organisms, reinforcement learning provides a path to the *simplification of learning complex behaviors* by only defining the reward and letting the algorithm learn reward-maximizing behaviors. The complexity of these behaviors is automatically inherited from that of the environment. This is the essence of Herbert Simon's ant hypothesis [453] at the beginning of this chapter. Reinforcement learning systems are inherently *end-to-end systems* in which a complex task is not broken up into smaller components, but viewed through the lens of a simple reward.

The simplest example of a reinforcement learning setting is the *multi-armed bandit problem*, which addresses the problem of a gambler choosing one of many slot machines in order to maximize his payoff. The gambler suspects that the (expected) rewards from the various slot machines are not the same, and therefore it makes sense to play the machine with the largest expected reward. Since the expected payoffs of the slot machines are not known in advance, the gambler has to *explore* different slot machines by playing them and also *exploit* the learned knowledge to maximize the reward. Although exploration of a particular slot machine might gain some additional knowledge about its payoff, it incurs the risk of the (potentially fruitless) cost of playing it. Multi-armed bandit algorithms provide carefully crafted strategies to optimize the trade-off between exploration and exploitation. However, in this simplified setting, each decision of choosing a slot machine is identical to the previous one. This is not quite the case in settings such as video games and self-driving cars with raw sensory inputs (e.g., video game screen or traffic conditions), which define the *state* of the system. Deep learners are excellent at distilling these sensory inputs into *state-sensitive* actions by wrapping their learning process within the exploration/exploitation framework.

Chapter Organization

This chapter is organized as follows. The next section introduces multi-armed bandits, which constitutes one of the simplest stateless settings in reinforcement learning. The notion of states is introduced in Section 9.3. The Q-learning method is introduced in Section 9.4. Policy gradient methods are discussed in Section 9.5. The use of Monte Carlo tree search strategies is discussed in Section 9.6. A number of case studies are discussed in Section 9.7. The safety issues associated with deep reinforcement learning methods are discussed in Section 9.8. A summary is given in Section 9.9.

9.2 Stateless Algorithms: Multi-Armed Bandits

We revisit the problem of a gambler who repeatedly plays slot machines based on previous experience. The gambler suspects that one of the slot machines has a better expected reward than others and attempts to both explore and exploit his experience with the slot machines. Trying the slot machines randomly is wasteful but helps in gaining experience. Trying the slot machines for a very small number of times and then always picking the best machine might lead to solutions that are poor in the long-term. How should one navigate this trade-off between exploration and exploitation? Note that every trial provides the same probabilistically distributed reward as previous trials for a given action, and therefore there is no notion of *state* in such a system. This is a simplified case of traditional reinforcement learning in which the notion of state is important. In a computer video game, moving the cursor in a particular direction has a reward that heavily depends on the *state* of the video game.

There are a number of strategies that the gambler can use to regulate the trade-off between exploration and exploitation of the search space. In the following, we will briefly describe some of the common strategies used in multi-armed bandit systems. All these methods are instructive because they provide the basic ideas and framework, which are used in generalized settings of reinforcement learning. In fact, some of these stateless algorithms are also used as subroutines in general forms of reinforcement learning. Therefore, it is important to explore this simplified setting.

9.2.1 Naïve Algorithm

In this approach, the gambler plays each machine for a fixed number of trials in the exploration phase. Subsequently, the machine with the highest payoff is used forever in the exploitation phase. Although this approach might seem reasonable at first sight, it has a number of drawbacks. The first problem is that it is hard to determine the number of trials at which one can confidently predict whether a particular slot machine is better than another machine. The process of estimation of payoffs might take a long time, especially in cases where the payoff events are rare compared to non-payoff events. Using many exploratory trials will waste a significant amount of effort on suboptimal strategies. Furthermore, if the wrong strategy is selected in the end, the gambler will use the wrong slot machine forever. Therefore, the approach of fixing a particular strategy forever is unrealistic in real-world problems.

9.2.2 ϵ-Greedy Algorithm

The ϵ-greedy algorithm is designed to use the best strategy as soon as possible, without wasting a significant number of trials. The basic idea is to choose a random slot machine for a fraction ϵ of the trials. These exploratory trials are also chosen at random (with probability ϵ) from all trials, and are therefore fully interleaved with the exploitation trials. In the remaining $(1 - \epsilon)$ fraction of the trials, the slot machine with the best average payoff so far is used. An important advantage of this approach is that one is guaranteed to not be trapped in the wrong strategy forever. Furthermore, since the exploitation stage starts early, one is often likely to use the best strategy a large fraction of the time.

The value of ϵ is an algorithm parameter. For example, in practical settings, one might set $\epsilon = 0.1$, although the best choice of ϵ will vary with the application at hand. It is often difficult to know the best value of ϵ to use in a particular setting. Nevertheless, the value of ϵ needs to be reasonably small in order to gain significant advantages from the exploitation portion of the approach. However, at small values of ϵ it might take a long time to identify the correct slot machine. A common approach is to use *annealing*, in which large values of ϵ are initially used, with the values declining with time.

9.2.3 Upper Bounding Methods

Even though the ϵ-greedy strategy is better than the naïve strategy in dynamic settings, it is still quite inefficient at learning the payoffs of new slot machines. In upper bounding strategies, the gambler does not use the mean payoff of a slot machine. Rather, the gambler takes a more optimistic view of slot machines that have not been tried sufficiently, and therefore uses a slot machine with the best *statistical upper bound* on the payoff. Therefore, one can consider the upper bound U_i of testing a slot machine i as the sum of expected reward Q_i and one-sided confidence interval length C_i:

$$U_i = Q_i + C_i \tag{9.1}$$

The value of C_i is like a bonus for increased uncertainty about that slot machine in the mind of the gambler. The value C_i is proportional to the standard deviation of the *mean* reward of the tries so far. According to the central limit theorem, this standard deviation is inversely proportional to the square-root of the number of times the slot machine i is tried (under the i.i.d. assumption). One can estimate the mean μ_i and standard deviation σ_i of the ith slot machine and then set C_i to be $K \cdot \sigma_i/\sqrt{n_i}$, where n_i is the number of times the ith slot machine has been tried. Here, K decides the level of confidence interval. Therefore, rarely tested slot machines will tend to have larger upper bounds (because of larger confidence intervals C_i) and will therefore be tried more frequently.

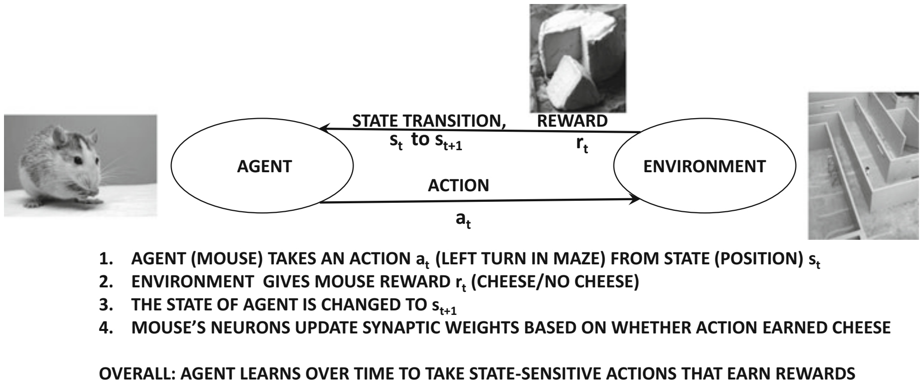
Unlike ϵ-greedy, the trials are no longer divided into two categories of exploration and exploitation; the process of selecting the slot machine with the largest upper bound has the dual effect of encoding both the exploration and exploitation aspects within each trial. One can regulate the trade-off between exploration and exploitation by using a specific level of statistical confidence. The choice of $K = 3$ leads to a 99.99% confidence interval for the upper bound under the Gaussian assumption. In general, increasing K will give large bonuses C_i for uncertainty, thereby causing exploration to comprise a larger proportion of the plays compared to an algorithm with smaller values of K.

9.3 The Basic Framework of Reinforcement Learning

The bandit algorithms of the previous section are stateless. In other words, the decision made at each time stamp has an identical environment, and the actions in the past only affect the knowledge of the agent (not the environment itself). This is not the case in generic reinforcement learning settings like video games or self-driving cars, which have a notion of *state.*

In generic reinforcement learning settings, each action is associated with a reward *in isolation.* While playing a video game, you do not get a reward only because you made a particular move. The reward of a move depends on all the other moves you made in the past, which are incorporated in the *state* of the environment. In a video game or self-driving car, we would need a different way of performing the credit assignment in a particular system state. For example, in a self-driving car, the reward for violently swerving a car in a normal state would be different from that of performing the same action in a state that indicates the danger of a collision. In other words, we need a way to quantify the reward of each action in a way that is specific to a particular system state.

In reinforcement learning, we have an *agent* that interacts with the *environment* with the use of *actions.* For example, the player is the agent in a video game, and moving the joystick in a certain direction in a video game is an action. The environment is the entire set up of the video game itself. These actions change the environment and lead to a new *state.* In a video game, the state represents all the variables describing the current position of the player at a particular point. The environment gives the agent rewards, depending on how well the goals of the learning application are being met. For example, scoring points in a video game is a reward. Note that the rewards may sometimes not be directly associated with a particular action, but with a combination of actions taken some time back. For example, the player might have cleverly positioned a cursor at a particularly convenient point a few movies back, and actions since then might have had no bearing on the reward. Furthermore, the reward for an action might itself not be deterministic in a particular state (e.g., pulling the lever of a slot machine). *One of the primary goals of reinforcement learning is to identify the inherent values of actions in different states, irrespective of the timing and stochasticity of the reward.*



1. AGENT (MOUSE) TAKES AN ACTION a_t (LEFT TURN IN MAZE) FROM STATE (POSITION) s_t
2. ENVIRONMENT GIVES MOUSE REWARD r_t (CHEESE/NO CHEESE)
3. THE STATE OF AGENT IS CHANGED TO s_{t+1}
4. MOUSE'S NEURONS UPDATE SYNAPTIC WEIGHTS BASED ON WHETHER ACTION EARNED CHEESE

OVERALL: AGENT LEARNS OVER TIME TO TAKE STATE-SENSITIVE ACTIONS THAT EARN REWARDS

Figure 9.1: The broad framework of reinforcement learning

The learning process helps the agent choose actions based on the inherent values of the actions in different states. This general principle applies to all forms of reinforcement learning in biological organisms, such as a mouse learning a path through a maze to earn a reward. The rewards earned by the mouse depend on an entire sequence of actions, rather than on only the latest action. When a reward is earned, the synaptic weights in the mouse's brain adjust to reflect how sensory inputs should be used to decide future actions in the maze. This is exactly the approach used in deep reinforcement learning, where a neural network is used to predict actions from sensory inputs (e.g., pixels of video game). This relationship between the agent and the environment is shown in Figure 9.1.

The entire set of states and actions and rules for transitioning from one state to another is referred to as a *Markov decision process*. The main property of a Markov decision process is that the state at any particular time stamp encodes all the information needed by the environment to make state transitions and assign rewards based on agent actions. Finite Markov decision processes (e.g., tic-tac-toe) terminate in a finite number of steps, which is referred to as an *episode*. A particular episode of this process is a finite sequence of actions, states, and rewards. An example of length $(n+1)$ is the following:

$$s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_t a_t r_t \ldots s_n a_n r_n$$

Note that s_t is the state *before* performing action a_t, and performing the action a_t causes a reward of r_t and transition to state s_{t+1}. This is the time-stamp convention used throughout this chapter (and several other sources), although the convention in Sutton and Barto's book [483] outputs r_{t+1} in response to action a_t in state s_t (which slightly changes the subscripts in all the results). Infinite Markov decision processes (e.g., continuously working robots) do not have finite length episodes and are referred to as *non-episodic*.

Examples

Although a system state refers to a complete description of the environment, many practical approximations are often made. For example, in an Atari video game, the system state might be defined by a fixed-length window of game snapshots. Some examples are as follows:

1. *Game of tic-tac-toe, chess, or Go:* The state is the position of the board at any point, and the actions correspond to the moves made by the agent. The reward is +1, 0, or −1 (depending on win, draw, or loss), *which is received at the end of the game.* Note that rewards are often not received immediately after strategically astute actions.

2. *Robot locomotion:* The state corresponds to the current configuration of robot joints and its position. The actions correspond to the torques applied to robot joints. The reward at each time stamp is a function of whether the robot stays upright and the amount of forward movement from point A to point B.

3. *Self-driving car:* The states correspond to the sensor inputs from the car, and the actions correspond to the steering, acceleration, and braking choices. The reward is a hand-crafted function of car progress and safety.

Some effort usually needs to be invested in defining the state representations and corresponding rewards. However, once these choices have been made, reinforcement learning frameworks are end-to-end systems.

9.3.1 Challenges of Reinforcement Learning

Reinforcement learning is more difficult than traditional forms of supervised learning for the following reasons:

1. When a reward is received (e.g., winning a game of chess), it is not exactly known how much each action has contributed to that reward. This problem lies at the heart of reinforcement learning, and is referred to as the *credit-assignment problem.* Furthermore, rewards may be probabilistic (e.g., pulling the lever of a slot machine), which can only be *estimated* approximately in a data-driven manner.

2. The reinforcement learning system might have a very large number of states (such as the number of possible positions in a board game), and must be able to make sensible decisions in states it has not seen before. This task of model generalization is the primary function of deep learning.

3. A specific choice of action affects the collected data in regard to future actions. As in multi-armed bandits, there is a natural trade-off between exploration and exploitation. If actions are taken only to learn their reward, then it incurs a cost to the player. On the other hand, sticking to known actions might result in suboptimal decisions.

4. Reinforcement learning merges the notion of data collection with learning. Realistic simulations of large physical systems such as robots and self-driving cars are limited by the need to physically perform these tasks and gather responses to actions in the presence of the practical dangers of failures. In many cases, the early portion of learning in a task may have few successes and many failures. *The inability to gather sufficient data in real settings beyond simulated and game-centric environments is arguably the single largest challenge to reinforcement learning.*

In the following sections, we will introduce a simple reinforcement learning algorithm and discuss the role of deep learning methods.

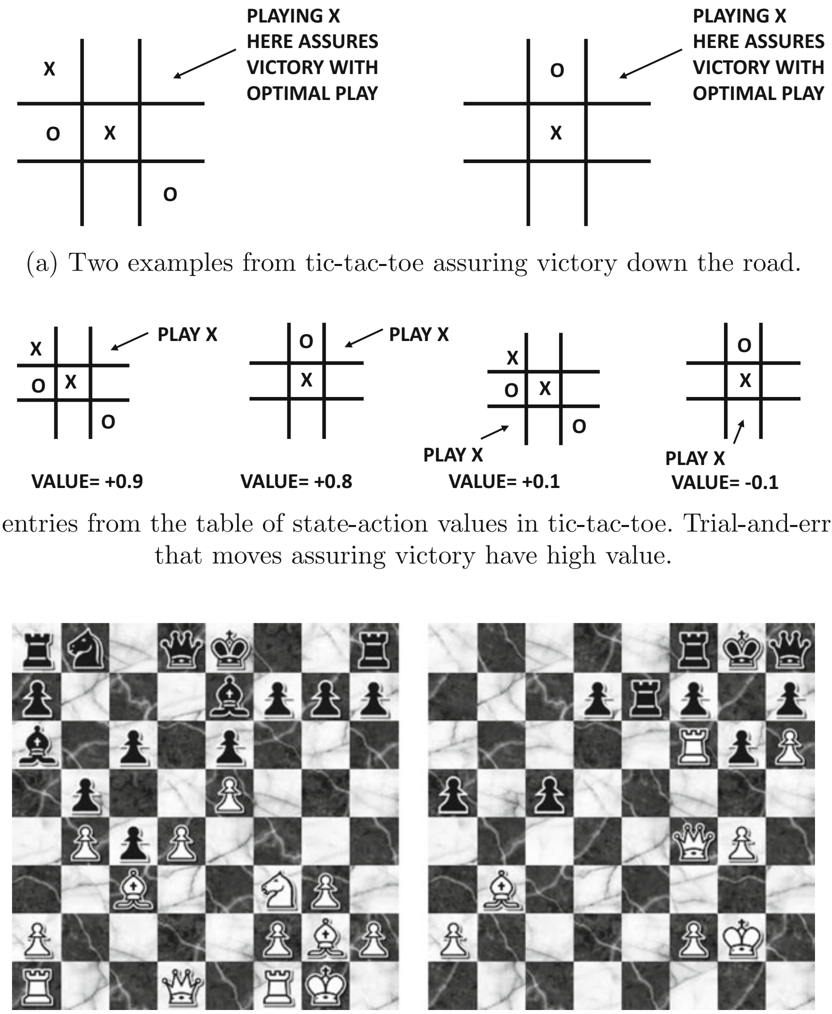
9.3.2 Simple Reinforcement Learning for Tic-Tac-Toe

One can generalize the stateless ϵ-greedy algorithm in the previous section to learn to play the game of tic-tac-toe. In this case, each board position is a state, and the action corresponds to placing 'X' or 'O' at a valid position. The number of valid states of the 3×3 board is bounded above by $3^9 = 19683$, which corresponds to three possibilities ('X', 'O', and blank) for each of 9 positions. Instead of estimating the value of each (stateless) action in multi-armed bandits, we now estimate the value of each state-action *pair* (s, a) based on the historical performance of action a in state s against a fixed opponent. Shorter wins are preferred at discount factor $\gamma < 1$, and therefore the *unnormalized* value of action a in state s is increased with γ^{r-1} in case of wins and $-\gamma^{r-1}$ in case of losses after r moves (including the current move). Draws are credited with 0. The discount also reflects the fact that the significance of an action decays with time in real-world settings. In this case, the table is updated only after all moves are made for a game (although later methods in this chapter allow *online* updates after each move). The normalized values of the actions in the table are obtained by dividing the unnormalized values with the number of times the state-action pair was updated (which is maintained separately). The table starts with small random values, and the action a in state s is chosen greedily to be the action with the highest normalized value with probability $1 - \epsilon$, and is chosen to be a random action otherwise. All moves in a game are credited after the termination of each game. Over time, the values of all state-action pairs will be learned and the resulting moves will also adapt to the play of the fixed opponent. Furthermore, one can even use self-play to generate these tables optimally. When self-play is used, the table is updated from a value in $\{-\gamma^r, 0, \gamma^r\}$ depending on win/draw/loss *from the perspective of the player for whom moves are made.* At inference time, the move with the highest normalized value from the perspective of the player are made.

9.3.3 Role of Deep Learning and a Straw-Man Algorithm

The aforementioned algorithm for tic-tac-toe did not use neural networks or deep learning, and this is also the case in many traditional algorithms for reinforcement learning [483]. The overarching goal of the ϵ-greedy algorithm for tic-tac-toe was to learn the inherent *long-term* value of each state-action pair, since the rewards are received long after valuable actions are performed. The goal of the training process is to perform the *value discovery* task of identifying which actions are truly beneficial in the long-term at a particular state. For example, making a clever move in tic-tac-toe might set a trap, which eventually results in assured victory. Examples of two such scenarios are shown in Figure 9.2(a) (although the trap on the right is somewhat less obvious). Therefore, one needs to credit a *strategically* good move favorably in the table of state-action pairs and not just the final winning move. The trial-and-error technique based on the ϵ-greedy method of Section 9.3.2 will indeed assign high values to clever traps. Examples of typical values from such a table are shown in Figure 9.2(b). Note that the less obvious trap of Figure 9.2(a) has a slightly lower value because moves assuring wins after longer periods are discounted by γ, and ϵ-greedy trial-and-error might have a harder time finding the win after setting the trap.

The main problem with this approach is that the number of states in many reinforcement learning settings is too large to tabulate explicitly. For example, the number of possible states in a game of chess is so large that the set of all known positions by humanity is


X
O X
O
PLAYING X HERE ASSURES VICTORY WITH OPTIMAL PLAY
O
X
PLAYING X HERE ASSURES VICTORY WITH OPTIMAL PLAY
PLAY X
VALUE= +0.9
PLAY X
VALUE= +0.8
PLAY X
VALUE= +0.1
PLAY X
VALUE= -0.1

(a) Two examples from tic-tac-toe assuring victory down the road.

(b) Four entries from the table of state-action values in tic-tac-toe. Trial-and-error learns that moves assuring victory have high value.

(c) Positions from two different games between *Alpha Zero* (white) and *Stockfish* (black) [447]: On the left, white sacrifices a pawn and concedes a passed pawn in order to trap black's light-square bishop behind black's own pawns. This strategy eventually resulted in a victory for white after many more moves than the horizon of a conventional chess-playing program like *Stockfish*. In the second game on the right, white has sacrificed material to incrementally cramp black to a position where all moves worsen the position. Incrementally improving positional advantage is the hallmark of the very best human players rather than chess-playing software like *Stockfish*, whose hand-crafted evaluations sometimes fail to accurately capture subtle differences in positions. The neural network in reinforcement learning, which uses the board state as input, evaluates positions in an integrated way without any prior assumptions. The data generated by trial-and-error provides the only experience for training a very complex evaluation function that is indirectly encoded within the parameters of the neural network. The trained network can therefore *generalize* these learned experiences to new positions. This is similar to how humans learn from previous games to better evaluate board positions.

Figure 9.2: Deep learners are needed for large state spaces like (c).

a minuscule fraction of the valid positions. In fact, the algorithm of Section 9.3.2 is a refined form of *rote learning* in which Monte Carlo simulations are used to refine and remember the long-term values of *seen* states. One learns about the value of a trap in tic-tac-toe only because previous Monte Carlo simulations have experienced victory many times *from that exact board position*. In most challenging settings like chess, one must *generalize* knowledge learned from prior experiences to a state that the learner has not seen before. All forms of learning (including reinforcement learning) are most useful when they are used to generalize known experiences to unknown situations. In such cases, the table-centric forms of reinforcement learning are woefully inadequate. Deep learning models serve the role of *function approximators*. Instead of learning and *tabulating* the values of all moves in all positions (using reward-driven trial and error), one learns the value of each move as a *function* of the input state, based on a *trained model* using the outcomes of prior positions. Without this approach, reinforcement learning cannot be used beyond toy settings like tic-tac-toe.

For example, a straw-man (but not very good) algorithm for chess might use the same ϵ-greedy algorithm of Section 9.3.2, but the values of actions are computed by using the board state as input to a convolutional neural network. The output is the evaluation of the board position. The ϵ-greedy algorithm is simulated to termination with the output values, and the discounted ground-truth value of each move in the simulation is selected from the set $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ depending on win/draw/loss and number of moves r to game completion (including the current move). Instead of updating a table of state-action pairs, the parameters of the neural network are updated by treating each move as a training point. The board position is input, and the output of the neural network is compared with the ground-truth value from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ to update the parameters. At inference time, the move with the best output score (with some minimax lookahead) can be used.

Although the aforementioned approach is too naive, a sophisticated system with Monte Carlo tree search, known as *Alpha Zero*, has recently been trained [447] to play chess. Two examples of positions [447] from different games in the match between *Alpha Zero* and a conventional chess program, *Stockfish-8.0*, are provided in Figure 9.2(c). In the chess position on the left, the reinforcement learning system makes a *strategically* astute move of cramping the opponent's bishop at the expense of immediate material loss, which most hand-crafted computer evaluations would not prefer. In the position on the right, *Alpha Zero* has sacrificed two pawns and a piece exchange in order to incrementally constrict black to a point where all its pieces are completely paralyzed. Even though *Alpha Zero* (probably) never encountered these specific positions during training, its deep learner has the ability to extract relevant features and patterns from previous trial-and-error experience in other board positions. In this particular case, the neural network seems to recognize the primacy of spatial patterns representing subtle positional factors over tangible material factors (much like a human's neural network).

In real-life settings, states are often described using sensory inputs. The deep learner uses this input representation of the state to learn the values of specific actions (e.g., making a move in a game) in lieu of the table of state-action pairs. Even when the input representation of the state (e.g., pixels) is quite primitive, neural networks are masters at squeezing out the relevant insights. This is similar to the approach used by humans to process primitive sensory inputs to define the *state* of the world and make decisions about *actions* using our biological neural network. We do not have a table of pre-memorized state-action pairs for every possible real-life situation. The deep-learning paradigm converts the forbiddingly large table of state-action values into a parameterized model mapping states-action pairs to values, which can be trained easily with backpropagation.

9.4 Bootstrapping for Value Function Learning

The simple generalization of the ϵ-greedy algorithm to tic-tac-toe (cf. Section 9.3.2) is a rather naive approach that does not work for *non-episodic settings.* In episodic settings like tic-tac-toe, a fixed-length sequence of at most nine moves can be used to characterize the full and final reward. In non-episodic settings like robots, the Markov decision process may not be finite or might be very long. Creating a sample of the ground-truth reward by Monte Carlo sampling becomes difficult and *online* updating might be desirable. This is achieved with the methodology of *bootstrapping.*

Intuition 9.4.1 (Bootstrapping) *Consider a Markov decision process in which we are predicting values (e.g., long-term rewards) at each time-stamp. We do not need the ground-truth at each time-stamp, as long as we can use a partial simulation of the future to improve the prediction at the current time-stamp. This improved prediction can be used as the ground-truth at the current time stamp for a model without knowledge of the future.*

For example, Samuel's checkers program [421] used the difference in evaluation at the current position and the minimax evaluation obtained by looking several moves ahead with the same function as a "prediction error" in order to update the evaluation function. The idea is that the minimax evaluation from looking ahead is stronger than the one without lookahead and can therefore be used as a "ground truth" to compute the error.

Consider a Markov decision process with the following sequence of states, actions, and rewards:

$$s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_t a_t r_t \ldots$$

For example, in a video game, each state s_t might represent a historical window of pixels [335] with a feature representation $\overline{X}_t$. In order to account for the (possibly) delayed rewards of actions, the cumulative reward R_t at time t is given by the discounted sum of the immediate rewards $r_t, r_{t+1}, r_{t+2}, \ldots r_\infty$ at all future time stamps:

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \gamma^3 \cdot r_{t+3} \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{9.2}$$

The discount factor $\gamma \in (0, 1)$ regulates how myopic we want to be in allocating rewards. The value of γ is less than 1 because future rewards are worth less than immediate rewards. Choosing $\gamma = 0$ will result in myopically setting the full reward R_t to r_t and nothing else. Therefore, it will be impossible to learn a long-term trap in tic-tac-toe. Values of γ that are too close to 1 will result in modeling instability for very long Markov decision processes.

The *Q-function* or *Q-value* for the state-action pair (s_t, a_t) is denoted by $Q(s_t, a_t)$, and is a measure of the *inherent* (i.e., long-term) value of performing the action a_t in state s_t. The Q-function $Q(s_t, a_t)$ represents the best possible reward obtained till the end of the game on performing the action a_t in state s_t. In other words, $Q(s_t, a_t)$ is equal to $\max\{E[R_{t+1}|a_t]\}$. Therefore, if A is the set of all possible actions, then the chosen action at time t is given by the action a_t^* that maximizes $Q(s_t, a_t)$. In other words, we have:

$$a_t^* = \text{argmax}_{a_t \in A} Q(s_t, a_t) \tag{9.3}$$

This predicted action is a good choice for the next move, although it is often combined with an exploratory component (e.g., ϵ-greedy policy) to improve long-term training outcomes.

9.4.1 Deep Learning Models as Function Approximators

For ease in discussion, we will work with the Atari setting [335] in which a fixed window of the last few snapshots of pixels provides the state s_t. Assume that the feature representation of s_t is denoted by $\overline{X}_t$. The neural network uses $\overline{X}_t$ as the input and outputs $Q(s_t, a)$ for each possible legal action a from the universe of actions denoted by the set A.

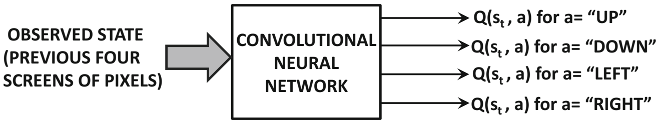

OBSERVED STATE (PREVIOUS FOUR SCREENS OF PIXELS)
CONVOLUTIONAL NEURAL NETWORK
Q(s$_t$, a) for a= “UP”
Q(s$_t$, a) for a= “DOWN”
Q(s$_t$, a) for a= “LEFT”
Q(s$_t$, a) for a= “RIGHT”

Figure 9.3: The Q-Network for the Atari video game setting

Assume that the neural network is parameterized by the vector of weights $\overline{W}$, and it has $|A|$ outputs containing the Q-values corresponding to the various actions in A. In other words, for each action $a \in A$, the neural network is able to compute the function $F(\overline{X}_t, \overline{W}, a)$, which is defined to be the *learned estimate* of $Q(s_t, a)$:

$$F(\overline{X}_t, \overline{W}, a) = \hat{Q}(s_t, a) \tag{9.4}$$

Note the circumflex on top of the Q-function in order to indicate that it is a predicted value using the learned parameters $\overline{W}$. Learning $\overline{W}$ is the key to using the model for deciding which action to use at a particular time-stamp. For example, consider a video game in which the possible moves are up, down, left, and right. In such a case, the neural network will have four outputs as shown in Figure 9.3. In the specific case of the Atari 2600 games, the input contains $m = 4$ spatial pixel maps in grayscale, representing the window of the last m moves [335, 336]. A convolutional neural network is used to convert pixels into Q-values. This network is referred to as a *Q-network*. We will provide more details of the specifics of the architecture later.

The Q-Learning Algorithm

The weights $\overline{W}$ of the neural network need to be learned via training. Here, we encounter an interesting problem. We can learn the vector of weights only if we have *observed* values of the Q-function. With observed values of the Q-function, we could easily set up a loss in terms of $Q(s_t, a) - \hat{Q}(s_t, a)$ in order to perform the learning after each action. The problem is that the Q-function represents the maximum discounted reward over all *future* combinations of actions, and there is no way of observing it at the current time.

Here, there is an interesting trick for setting up the neural network loss function. According to Intuition 9.4.1, *we do not really need the observed Q-values in order to set up a loss function as long as we know an improved estimate of the Q-values by using partial knowledge from the future.* Then, we can use this improved estimate to create a surrogate “observed” value. This “observed” value is defined by the *Bellman equation* [26], which is a dynamic programming relationship satisfied by the Q-function, and the partial knowledge is the reward observed at the current time-stamp for each action. According to the Bellman equation, we set the “ground-truth” by looking ahead one step and predicting at s_{t+1}:

$$Q(s_t, a_t) = r_t + \gamma \text{max}_a \hat{Q}(s_{t+1}, a) \tag{9.5}$$

The correctness of this relationship follows from the fact that the Q-function is designed to maximize the discounted future payoff. We are essentially looking at all actions one step ahead in order to create an improved estimate of $Q(s_t, a_t)$. It is important to set $\hat{Q}(s_{t+1}, a)$ to 0 in case the process terminates after performing a_t for episodic sequences. We can write this relationship in terms of our neural network predictions as well:

$$F(\overline{X}_t, \overline{W}, a_t) = r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a) \tag{9.6}$$

Note that one must first wait to observe the state $\overline{X}_{t+1}$ and reward r_t by performing the action a_t, before we can compute the "observed" value at time-stamp t on the right-hand side of the above equation. This provides a natural way to express the loss L_t of the neural network at time stamp t by comparing the (surrogate) observed value to the predicted value at time stamp t:

$$L_t = \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\}^2 \tag{9.7}$$

Therefore, we can now update the vector of weights $\overline{W}$ using backpropagation on this loss function. Here, it is important to note that the target values estimated using the inputs at $(t+1)$ are treated as constant ground-truths by the backpropagation algorithm. Therefore, the derivative of the loss function will treat these estimated values as constants, even though they were obtained from the parameterized neural network with input $\overline{X}_{t+1}$. Not treating $F(\overline{X}_{t+1}, \overline{W}, a)$ as a constant will lead to poor results. This is because we are treating the prediction at $(t + 1)$ as an improved estimate of the ground-truth (based on the bootstrapping principle). Therefore, the backpropagation algorithm will compute the following:

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \tag{9.8}$$

In matrix-calculus notation, the partial derivative of a function $F()$ with respect to the vector $\overline{W}$ is essentially the gradient $\nabla_{\overline{W}} F$. At the beginning of the process, the Q-values estimated by the neural network are random because the vector of weights $\overline{W}$ is initialized randomly. However, the estimation gradually becomes more accurate with time, as the weights are constantly changed to maximize rewards.

Therefore, at any given time-stamp t at which action a_t and reward r_t has been observed, the following training process is used for updating the weights $\overline{W}$:

1. Perform a forward pass through the network with input $\overline{X}_{t+1}$ to compute $\hat{Q}_{t+1} = \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)$. The value is 0 in case of termination after performing a_t. *Treating the terminal state specially is important.* According to the Bellman equations, the Q-value at previous time-stamp t should be $r_t + \gamma \hat{Q}_{t+1}$ for observed action a_t at time t. Therefore, instead of using observed values of the target, we have created a *surrogate* for the target value at time t, and we pretend that this surrogate is an observed value given to us.

2. Perform a forward pass through the network with input $\overline{X}_t$ to compute $F(\overline{X}_t, \overline{W}, a_t)$.

3. Set up a loss function in $L_t = (r_t + \gamma Q_{t+1} - F(\overline{X}_t, \overline{W}, a_t))^2$, and backpropagate in the network with input $\overline{X}_t$. Note that this loss is associated with neural network output node corresponding to action a_t, and the loss for all other actions is 0.

4. One can now use backpropagation on this loss function in order to update the weight vector $\overline{W}$. Even though the term $r_t + \gamma Q_{t+1}$ in the loss function is also obtained as a prediction from input $\overline{X}_{t+1}$ to the neural network, it is treated as a (constant) observed value during gradient computation by the backpropagation algorithm.

Both the training and the prediction are performed simultaneously, as the values of actions are used to update the weights and select the next action. It is tempting to select the action with the largest Q-value as the relevant prediction. However, such an approach might perform inadequate exploration of the search space. Therefore, one couples the optimality prediction with a policy such as the ϵ-greedy algorithm in order to select the next action. The action with the largest predicted payoff is selected with probability $(1-\epsilon)$. Otherwise, a random action is selected. The value of ϵ can be annealed by starting with large values and reducing them over time. Therefore, the *target prediction value* for the neural network is computed using the best possible action in the Bellman equation (which might eventually be different from observed action a_{t+1} based on the ϵ-greedy policy). This is the reason that Q-learning is referred to as an *off-policy algorithm* in which the target prediction values for the neural network update are computed using actions that might be different from the actually observed actions in the future.

There are several modifications to this basic approach in order to make the learning more stable. Many of these are presented in the context of the Atari video game setting [335]. First, presenting the training examples *exactly* in the sequence they occur can lead to local minima because of the strong similarity among training examples. Therefore, a fixed-length history of actions/rewards is used as a pool. One can view this as a history of experiences. Multiple experiences are sampled from this pool to perform mini-batch gradient descent. In general, it is possible to sample the same action multiple times, which leads to greater efficiency in leveraging the learning data. Note that the pool is updated over time as old actions drop out of the pool and newer ones are added. Therefore, the training is still temporal in an approximate sense, but not strictly so. This approach is referred to as *experience replay*, as experiences are replayed multiple times in a somewhat different order than the original actions.

Another modification is that the network used for estimating the target Q-values with Bellman equations (step 1 above) is not the same as the network used for predicting Q-values (step 2 above). The network used for estimating the target Q-values is updated more slowly in order to encourage stability. Finally, one problem with these systems is the sparsity of the rewards, especially at the initial stage of the learning when the moves are random. For such cases, a variety of tricks such as *prioritized experience replay* [428] can be used. The basic idea is to make more efficient use of the training data collected during reinforcement learning by prioritizing actions from which more can be learned.

9.4.2 Example: Neural Network for Atari Setting

For the convolutional neural network [335, 336], the screen sizes were set to 84 × 84 pixels, which also defined the spatial footprints of the first layer in the convolutional network. The input was in grayscale, and therefore each screen required only a single spatial feature map, although a depth of 4 was required in the input layer to represent the previous four windows of pixels. Three convolutional layers were used with filters of size 8 × 8, 4 × 4, and 3 × 3, respectively. A total of 32 filters were used in the first convolutional layer, and 64 filters were used in each of the other two, with the strides used for convolution being 4, 2, and 1, respectively. The convolutional layers were followed by two fully connected layers. The number of neurons in the penultimate layer was equal to 512, and that in the final layer was equal to the number of outputs (possible actions). The number of output layers varied between 4 and 18, and was game-specific. The overall architecture of the convolutional network is illustrated in Figure 9.4.

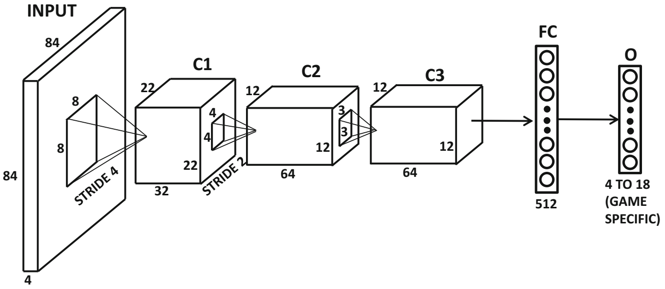


Figure 9.4: The convolutional neural network for the Atari setting

All hidden layers used the ReLU activation, and the output used linear activation in order to predict the real-valued Q-value. No pooling was used, and the strides in the convolution provided spatial compression. The Atari platform supports many games, and the same broader architecture was used across different games in order to showcase its generalizability. There was some variation in performance across different games, although human performance was exceeded in many cases. The algorithm faced the greatest challenges in games in which longer-term strategies were required. Nevertheless, the robust performance of a relatively homogeneous framework across many games was encouraging.

9.4.3 On-Policy Versus Off-Policy Methods: SARSA

The Q-Learning methodology belongs to the class of methods, referred to as *temporal difference learning.* In Q-learning, the actions are chosen according to an ϵ-greedy policy. However, the parameters of the neural network are updated based on the best possible action at each step with the Bellman equation. The best possible action at each step is not quite the same as the ϵ-greedy policy used to perform the simulation. Therefore, Q-learning is an *off-policy reinforcement learning method.* Choosing a different policy for executing actions from those for performing updates does not worsen the ability to find the optimum solutions that are goals of the updates. In fact, since more exploration is performed with a randomized policy, local optima are avoided.

In *on-policy methods*, the actions are consistent with the updates, and therefore the updates can be viewed as policy *evaluation* rather than *optimization.* In order to understand this point, we will describe the updates for the SARSA (State-Action-Reward-State-Action) algorithm, in which the optimal reward in the next step is not used for computing updates. Rather, the next step is updated using the same ϵ-greedy policy to obtain the action a_{t+1} for computing the target values. Then, the loss function for the next step is defined as follows:

$$L_t = \left\{r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1}) - F(\overline{X}_t, \overline{W}, a_t)\right\}^2 \tag{9.9}$$

The function $F(\cdot, \cdot, \cdot)$ is defined in the same way as the previous section. The weight vector is updated based on this loss, and then the action a_{t+1} is executed:

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1})]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \tag{9.10}$$

Here, it is instructive to compare this update with those used in Q-learning according to Equation 9.8. In Q-learning, one is using the *best possible* action at each state in order to update the parameters, even though the policy that is actually executed might be ϵ-greedy (which encourages exploration). In SARSA, we are using the action that was actually selected by the ϵ-greedy method in order to perform the update. Therefore, the approach is an *on-policy method*. Off-policy methods like Q-learning are able to decouple exploration from exploitation, whereas on-policy methods are not. Note that if we set the value of ϵ in the ϵ-greedy policy to 0 (i.e., vanilla greedy), then both Q-Learning and SARSA would specialize to the same algorithm. However, such an approach would not work very well because there is no exploration. SARSA is useful when learning cannot be done separately from prediction. Q-learning is useful when the learning can to be done offline, which is followed by exploitation of the learned policy with a vanilla-greedy method at $\epsilon = 0$ (and no need for further model updates). Using ϵ-greedy at inference time would be dangerous in Q-learning, because the policy never pays for its exploratory component and therefore does not learn how to keep exploration safe. For example, a Q-learning based robot will take the shortest path to get from point A to point B even if it is along the edge of the cliff, whereas a SARSA-trained robot will not.

Learning Without Function Approximators

It is possible to also learn Q-values without using function approximators *in cases where the state-space is very small.* For example, in a toy game like tic-tac-toe, one can learn $Q(s_t, a_t)$ explicitly by using trial-and-error play against a strong opponent. In this case, the Bellman equations (cf. Equation 9.5) are used at each move to update an *array* containing the explicit value of $Q(s_t, a_t)$. Using Equation 9.5 directly is too aggressive. More generally, gentle updates are performed for learning rate $\alpha < 1$:

$$Q(s_t, a_t) \Leftarrow Q(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma \text{max}_a Q(s_{t+1}, a)) \tag{9.11}$$

Using $\alpha = 1$ will result in Equation 9.5. Updating the array continually will result in a table containing the correct *strategic* value of each move; see, for example, Figure 9.2(a) for an understanding of the notion of strategic value. Figure 9.2(b) contains examples of four entries from such a table.

One can also use the SARSA algorithm without function approximators by using the action a_{t+1} based on the ϵ-greedy policy. We use a superscript p in $Q^p(\cdot, \cdot)$ to indicate that it is a policy evaluation operator of the policy p (which is ϵ-greedy in this case):

$$Q^p(s_t, a_t) \Leftarrow Q^p(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1})) \tag{9.12}$$

This approach is a more sophisticated alternative to the ϵ-greedy method discussed in Section 9.3.2. Note that if action a_t at state s_t leads to termination (for episodic processes), then $Q^p(s_t, a_t)$ is simply set to r_t.

9.4.4 Modeling States Versus State-Action Pairs

A minor variation of the theme in the previous sections is to learn the value of a particular state (rather than state-action pair). One can implement all the methods discussed earlier by maintaining values of states rather than state-action pairs. For example, SARSA can be implemented by evaluating all the values of states resulting from each possible action and selecting a good one based on a pre-defined policy like ϵ-greedy. In fact, the earliest methods for temporal difference learning (or *TD-learning*) maintained values on states rather than state-action pairs. From an efficiency perspective, it is more convenient to output the values of all actions in one shot (rather than repeatedly evaluate each forward state) for value-based decision making. Working with state values rather that state-action pairs becomes useful only when the policy cannot be expressed neatly in terms of state-action pairs. For example, we might evaluate a forward-looking tree of promising moves in chess, and report some averaged value for bootstrapping. In such cases, it is desirable to evaluate states rather than state-action pairs. This section will therefore discuss a variation of temporal difference learning in which states are directly evaluated.

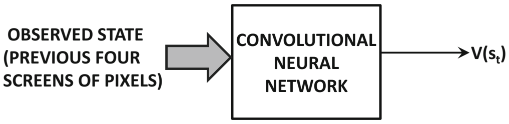


Figure 9.5: Estimating the value of a state with temporal difference learning

Let the value of the state s_t be denoted by $V(s_t)$. Now assume that you have a parameterized neural network that uses the observed attributes $\overline{X}_t$ (e.g., pixels of last four screens in Atari game) of state s_t to estimate $V(s_t)$. An example of this neural network is shown in Figure 9.5. Then, if the function computed by the neural network is $G(\overline{X}_t, \overline{W})$ with parameter vector $\overline{W}$, we have the following:

$$G(\overline{X}_t, \overline{W}) = \hat{V}(s_t) \tag{9.13}$$

Note that the policy being followed to decide the actions might use some arbitrary evaluation of forward-looking states to decide actions. For now, we will assume that we have some reasonable heuristic policy for choosing the actions that uses the forward-looking state values in some way. For example, if we evaluate each forward state resulting from an action and select one of them based on a pre-defined policy (e.g., ϵ-greedy), the approach discussed below is the same as SARSA.

If the action a_t is performed with reward r_t, the resulting state is s_{t+1} with value $V(s_{t+1})$. Therefore, the bootstrapped ground-truth estimate for $V(s_t)$ can be obtained with the help of this lookahead:

$$V(s_t) = r_t + \gamma V(s_{t+1}) \tag{9.14}$$

This estimate can also be stated in terms of the neural network parameters:

$$G(\overline{X}_t, \overline{W}) = r_t + \gamma G(\overline{X}_{t+1}, \overline{W}) \tag{9.15}$$

During the training phase, one needs to shift the weights so as to push $G(\overline{X}_t, \overline{W})$ towards the improved "ground truth" value of $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$. As in the case of Q-learning, we work with the boot-strapping pretension that the value $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$ is an observed value given to us. Therefore, we want to minimize the *TD-error* defined by the following:

$$\delta_t = \underbrace{r_t + \gamma G(\overline{X}_{t+1}, \overline{W})}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \tag{9.16}$$

Therefore, the loss function L_t is defined as follows:

$$L_t = \delta_t^2 = \left\{ \underbrace{r_t + \gamma G(\overline{X}_{t+1}, \overline{W})}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \right\}^2 \tag{9.17}$$

As in Q-learning, one would first compute the "observed" value of the state at time stamp t using the input $\overline{X}_{t+1}$ into the neural network to compute $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$. Therefore, one would have to wait till the action a_t has been observed, and therefore the observed features $\overline{X}_{t+1}$ of state s_{t+1} are available. This "observed" value (defined by $r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$) of state s_t is then used as the (constant) target to update the weights of the neural network, when the input $\overline{X}_t$ is used to predict the value of the state s_t. Therefore, one would need to move the weights of the neural network based on the gradient of the following loss function:

$$\begin{aligned}
\overline{W} &\Leftarrow \overline{W} - \alpha \frac{\partial L_t}{\partial \overline{W}} \\
&= \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma G(\overline{X}_{t+1}, \overline{W})]}_{\text{"Observed" value}} - G(\overline{X}_t, \overline{W}) \right\} \frac{\partial G(\overline{X}_t, \overline{W})}{\partial \overline{W}} \\
&= \overline{W} + \alpha \delta_t (\nabla G(\overline{X}_t, \overline{W}))
\end{aligned}$$

This algorithm is a special case of the $TD(\lambda)$ algorithm with λ set to 0. This special case only updates the neural network by creating a bootstrapped "ground-truth" for the current time-stamp based on the evaluations of the next time-stamp. This type of ground-truth is an inherently myopic *approximation.* For example, in a chess game, the reinforcement learning system might have inadvertently made some mistake many steps ago, and it is suddenly showing high errors in the bootstrapped predictions without having shown up earlier. The errors in the bootstrapped predictions are indicative of the fact that we have received new information about each past state $\overline{X}_k$, which we can use to alter its prediction. One possibility is to bootstrap by looking ahead for multiple steps (see Exercise 7). Another solution is the use of $TD(\lambda)$, which explores the continuum between perfect Monte Carlo ground truth and single-step approximation with smooth decay. The adjustments to older predictions are increasingly discounted at the rate $\lambda < 1$. In such a case, the update can be shown to be the following [482]:

$$\overline{W} \Leftarrow \overline{W} + \alpha \delta_t \sum_{k=0}^{t} \underbrace{(\lambda\gamma)^{t-k} (\nabla G(\overline{X}_k, \overline{W}))}_{\text{Alter prediction of } \overline{X}_k} \tag{9.18}$$

At $\lambda = 1$, the approach can be shown to be equivalent to a method in which Monte-Carlo evaluations (i.e., rolling out an episodic process to the end) are used to compute the ground-truth [482]. This is because we are always using new information about errors to fully correct

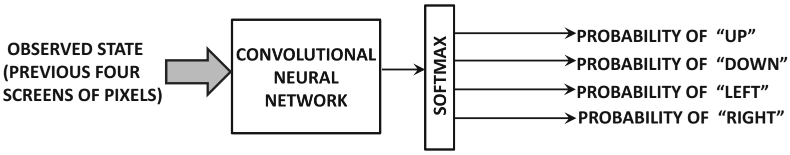


Figure 9.6: The policy network for the Atari video game setting. It is instructive to compare this configuration with the Q-network of Figure 9.3.

our past mistakes without discount at $\lambda = 1$, thereby creating an unbiased estimate. Note that λ is only used for discounting the steps, whereas γ is also used in computing the TD-error δ_t according to Equation 9.16. The parameter λ is *algorithm-specific*, whereas γ is *environment-specific*. Using $\lambda = 1$ or Monte Carlo sampling leads to lower bias and higher variance. For example, consider a chess game in which agents Alice and Bob each make three errors in a single game but Alice wins in the end. This single Monte Carlo roll out will not be able to distinguish the impact of each specific error and will assign the discounted credit for final game outcome to each board position. On the other hand, an n-step temporal difference method (i.e., n-ply board evaluation) might see a temporal difference error for each board position in which the agent made a mistake and was detected by the n-step lookahead. It is only with sufficient data (i.e., more games) that the Monte Carlo method will distinguish between different types of errors. However, choosing very small values of λ will have difficulty in learning openings (i.e., greater bias) because errors with long-term consequences will not be detected. Such problems with openings are well documented [22, 496].

Temporal difference learning was used in Samuel's celebrated checkers program [421], and also motivated the development of TD-Gammon for Backgammon by Tesauro [492]. A neural network was used for state value estimation, and its parameters were updated using temporal-difference bootstrapping over successive moves. The final inference was performed with minimax evaluation of the improved evaluation function over a shallow depth such as 2 or 3. TD-Gammon was able to defeat several expert players. It also exhibited some unusual strategies of game play that were eventually adopted by top-level players.

9.5 Policy Gradient Methods

The value-based methods like Q-learning attempt to predict the value of an action with the neural network and couple it with a generic policy (like ϵ-greedy). On the other hand, policy gradient methods estimate the *probability* of each action at each step with the goal of maximizing the overall reward. Therefore, the policy is itself parameterized, rather than using the value estimation as an intermediate step for choosing actions.

The neural network for estimating the policy is referred to as a *policy network* in which the input is the current state of the system, and the output is a set of probabilities associated with the various actions in the video game (e.g., moving up, down, left, or right). As in the case of the Q-network, the input can be an observed representation of the agent state. For example, in the Atari video game setting, the observed state can be the last four screens of pixels. An example of a policy network is shown in Figure 9.6, which is relevant for the Atari setting. It is instructive to compare this policy network with the Q-network of Figure 9.3. Given an output of probabilities for various actions, we throw a biased die with the faces associated with these probabilities, and select one of these actions. Therefore, for each action a, observed state representation $\overline{X}_t$, and current parameter $\overline{W}$, the neural network is able to compute the function $P(\overline{X}_t, \overline{W}, a)$, which is the probability that the action a should be performed. One of the actions is sampled, and a reward is observed for that action. If the policy is poor, the action will more likely to be a mistake and the reward will be poor as well. Based on the reward obtained from executing the action, the weight vector $\overline{W}$ is updated for the next iteration. The update of the weight vector is based on the notion of policy gradient with respect to the weight vector $\overline{W}$. One challenge in estimating the policy gradient is that the reward of an action is often not observed immediately, but is tightly integrated into the future sequence of rewards. Often *Monte Carlo policy roll-outs* must be used in which the neural network is used to follow a particular policy to estimate the discounted rewards over a longer horizon.

We want to update the weight vector of the neural network along the gradient of increasing the reward. As in Q-Learning, the expected discounted rewards over a given horizon H are computed as follows:

$$J = E[r_0 + \gamma \cdot r_1 + \gamma^2 \cdot r_2 + \ldots + \gamma^H \cdot r_H] = \sum_{i=0}^{H} E[\gamma^i r_i] \tag{9.19}$$

Therefore, the goal is to update the weight vector as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha \nabla J \tag{9.20}$$

The main problem in estimating the gradient ∇J is that the neural network only outputs probabilities. The observed rewards are only Monte Carlo samples of these outputs, whereas we want to compute the gradients of *expected* rewards (cf. Equation 9.19). Common policy gradients methods include *finite difference methods*, *likelihood ratio methods*, and *natural policy gradients*. In the following, we will only discuss the first two methods.

9.5.1 Finite Difference Methods

The method of finite differences side-steps the problem of stochasticity with empirical simulations that provide estimates of the gradient. Finite difference methods use weight perturbations in order to estimate gradients of the reward. The idea is to use s different perturbations of the neural network weights, and examine the expected change ΔJ in the reward. Note that this will require us to run the perturbed policy for the horizon of H moves in order to estimate the change in reward. Such a sequence of H moves is referred to as a *roll-out*. For example, in the case of the Atari game, we will need to play it for a trajectory of H moves for each of these s different sets of perturbed weights in order to estimate the changed reward. In games where an opponent of sufficient strength is not available to train against, it is possible to play a game against a version of the opponent based on parameters learned a few iterations back.

In general, the value of H might be large enough that we might reach the end of the game, and therefore the score used will be the one at the end of the game. In some games like *Go*, the score is available only at the end of the game, with a $+1$ for a win and -1 for a loss. In such cases, it becomes more important to choose H large enough so as to play till the end of the game. As a result, we will have s different weight (change) vectors $\Delta\overline{W}_1 \ldots \Delta\overline{W}_s$, together with corresponding changes $\Delta J_1 \ldots \Delta J_s$ in the total reward. Each of these pairs roughly satisfies the following relationship:

$$(\Delta\overline{W}_r)\nabla J^T \approx \Delta J_r \quad \forall r \in \{1 \ldots s\} \tag{9.21}$$

We can create an s-dimensional column vector $\overline{y} = [\Delta J_1 \dots \Delta J_s]^T$ of the changes in the objective function and an $N \times s$ matrix D by stacking the rows $\Delta \overline{W}_r$ on top of each other, where N is the number of parameters in the neural network. Therefore, we have the following:

$$D[\nabla J]^T \approx \overline{y} \tag{9.22}$$

Then, the policy gradient is obtained by performing a straightforward linear regression of the change in objective functions with respect to the change in weight vectors. By using the formula for linear regression (cf. Section 2.2.2.2 of Chapter 2), we obtain the following:

$$\nabla J^T = (D^T D)^{-1} D^T \overline{y} \tag{9.23}$$

This gradient is used for the update in Equation 9.20. It is required to run the policy for a sequence of H moves for each of the s samples to estimate the gradients. This process can sometimes be slow.

9.5.2 Likelihood Ratio Methods

Likelihood-ratio methods were proposed by Williams [533] in the context of the REINFORCE algorithm. Consider the case in which we are following the policy with probability vector $\overline{p}$ and we want to maximize $E[Q^p(s,a)]$, which is the long-term expected value of state s and each sampled action a from the neural network. Consider the case in which the probability of action a is $p(a)$ (which is output by the neural network). In such a case, we want to find the gradient of $E[Q^p(s,a)]$ with respect to the weight vector $\overline{W}$ of the neural network for stochastic gradient ascent. Finding the gradient of an expectation from sampled events is non-obvious. However, the log-probability trick allows us to convert it into the expectation of a gradient, which is additive over the samples of state-action pairs:

$$\nabla E[Q^p(s,a)] = E[Q^p(s,a) \nabla \text{log}(p(a))] \tag{9.24}$$

We show the proof of the above result in terms of the partial derivative with respect to a single neural network weight w under the assumption that a is a discrete variable:

$$\begin{aligned}\frac{\partial E[Q^p(s,a)]}{\partial w} &= \frac{\partial \left[\sum_a Q^p(s,a)p(a)\right]}{\partial w} = \sum_a Q^p(s,a)\frac{\partial p(a)}{\partial w} = \sum_a Q^p(s,a)\left[\frac{1}{p(a)}\frac{\partial p(a)}{\partial w}\right]p(a)\\ &= \sum_a Q^p(s,a)\left[\frac{\partial \text{log}(p(a))}{\partial w}\right]p(a) = E\left[Q^p(s,a)\frac{\partial \text{log}(p(a))}{\partial w}\right]\end{aligned}$$

The above result can also be shown for the case in which a is a continuous variable (cf. Exercise 1). Continuous actions occur frequently in robotics (e.g., distance to move arm).

It is easy to use this trick for neural network parameter estimation. Each action a sampled by the simulation is associated with the long-term reward $Q^p(s,a)$, which is obtained by Monte Carlo simulation. Based on the relationship above, the gradient of the expected advantage is obtained by multiplying the gradient of the log-probability $\log(p(a))$ of that action (computable from the neural network in Figure 9.6 using backpropagation) with the long-term reward $Q^p(s,a)$ (obtained by Monte Carlo simulation).

Consider a simple game of chess with a win/loss/draw at the end and discount factor γ In this case, the long-term reward of each move is simply obtained as a value from $\{+\gamma^{r-1}, 0, -\gamma^{r-1}\}$, when r moves remain to termination. The value of the reward depends on the final outcome of the game, and number of remaining moves (because of reward discount). Consider a game containing at most H moves. Since multiple roll-outs are used, we get a whole bunch of training samples for the various input states and corresponding outputs in the neural network. For example, if we ran the simulation for 100 roll-outs, we would get at most $100 \times H$ different samples. Each of these would have a long-term reward drawn from $\{+\gamma^{r-1}, 0, -\gamma^{r-1}\}$. For each of these samples, the reward serves as a weight during a gradient-ascent update of the log-probability of the sampled action.

$$\overline{W} \Leftarrow \overline{W} + Q^p(s,a)\nabla \log(p(a)) \tag{9.25}$$

Here, $p(a)$ is the neural network's output probability of the sampled action. The gradients are computed using backpropagation, and these updates are similar to those in Equation 9.20. This process of sampling and updating is carried through to convergence.

Note that the gradient of the log-probability of the ground-truth class is often used to update softmax classifiers with cross-entropy loss in order to increase the probability of the correct class (which is intuitively similar to the update here). The difference here is that we are weighting the update with the Q-values because we want to push the parameters more aggressively in the direction of highly rewarding actions. One could also use mini-batch gradient ascent over the actions in the sampled roll-outs. Randomly sampling from different roll-outs can be helpful in avoiding the local minima arising from correlations because the successive samples from each roll-out are closely related to one another.

Reducing Variance with Baselines: Although we have used the long-term reward $Q^p(s,a)$ as the quantity to be optimized, it is more common to subtract a baseline value from this quantity in order to obtain its *advantage* (i.e, differential impact of the action over expectation). The baseline is ideally state-specific, but can be a constant as well. In the original work of REINFORCE, a constant baseline was used (which is typically some measure of average long-term reward over all states). Even this type of simple measure can help in speeding up learning because it reduces the probabilities of less-than-average performers and increases the probabilities of more-than-average performers (rather than increasing both at differential rates). A constant choice of baseline does not affect the bias of the procedure, but it reduces the variance. A *state-specific* option for the baseline is the value $V^p(s)$ of the state s immediately *before* sampling action a. Such a choice results in the advantage $(Q^p(s,a) - V^p(s))$ becoming identical to the temporal difference error. This choice makes intuitive sense, because the temporal difference error contains *additional* information about the differential reward of an action beyond what we would know before choosing the action. Discussions on baseline choice may be found in [374, 433].

Consider an example of an Atari game-playing agent, in which a roll-out samples the move UP and output probability of UP was 0.2. Assume that the (constant) baseline is 0.17, and the long-term reward of the action is +1, since the game results in win (and there is no reward discount). Therefore, the score of every action in that roll-out is 0.83 (after subtracting the baseline). Then, the gain associated with all actions (output nodes of the neural network) other than UP at that time-step would be 0, and the gain associated with the output node corresponding to UP would be $0.83 \times \log(0.2)$. One can then backpropagate this gain in order to update the parameters of the neural network.

Adjustment with a state-specific baseline is easy to explain intuitively. Consider the example of a chess game between agents Alice and Bob. If we use a baseline of 0, then each move will only be credited with a reward corresponding to the final result, and the difference between good moves and bad moves will not be evident. In other words, we need to simulate a lot more games to differentiate positions. On the other hand, if we use the value of the state (before performing the action) as the baseline, then the (more refined) temporal difference error is used as the advantage of the action. In such a case, moves that have greater state-specific impact will be recognized with a higher advantage (within a single game). As a result, fewer games will be required for learning.

9.5.3 Combining Supervised Learning with Policy Gradients

Supervised learning is useful for initializing the weights of the policy network before applying reinforcement learning. For example, in a game of chess, one might have prior examples of expert moves that are already known to be good. In such a case, we simply perform gradient ascent with the same policy network, except that each expert move is assigned the fixed credit of 1 for evaluating the gradient according to Equation 9.24. This problem becomes identical to that of softmax classification, where the goal of the policy network is to predict the same move as the expert. One can sharpen the quality of the training data with some examples of bad moves with a negative credit obtained from computer evaluations. This approach would be considered supervised learning rather than reinforcement learning because we are simply using prior data, and not generating/simulating the data that we learn from (as is common in reinforcement learning). This general idea can be extended to any reinforcement learning setting, where some prior examples of actions and associated rewards are available. Supervised learning is extremely common in these settings for initialization because of the difficultly in obtaining high-quality data in the early stages of the process. Many published works also interleave supervised learning and reinforcement learning in order to achieve greater data efficiency [286].

9.5.4 Actor-Critic Methods

So far, we have discussed methods that are either dominated by *critics* or by *actors* in the following way:

1. The Q-learning and $TD(\lambda)$ methods work with the notion of a value function that is optimized. This value function is a critic, and the policy (e.g., ϵ-greedy) of the actor is directly derived from this critic. Therefore, the actor is subservient to the critic, and such methods are considered *critic-only* methods.

2. The policy-gradient methods do not use a value function at all, and they directly learn the probabilities of the policy actions. The values are often estimated using Monte Carlo sampling. Therefore, these methods are considered *actor-only* methods.

Note that the policy-gradient methods do need to evaluate the advantage of intermediate actions, and this estimation has so far been done with the use of Monte Carlo simulations. The main problem with Monte Carlo simulations is its high complexity and inability to use in an online setting.

However, it turns out that one can learn the advantage of intermediate actions using value function methods. As in the previous section, we use the notation $Q^p(s_t, a)$ to denote the value of action a, when the policy p followed by the policy network is used. Therefore, we would now have two coupled neural networks– a policy network and a Q-network. The policy network learns the probabilities of actions, and the Q-network learns the values $Q^p(s_t, a)$ of various actions in order to provide an estimation of the advantage to the policy network. Therefore, the policy network uses $Q^p(s_t, a)$ (with baseline adjustments) to weight its gradient ascent updates. The Q-network is updated using an on-policy update as in SARSA, where the policy is controlled by the policy network (rather than ϵ-greedy). The Q-network, however, does not directly decide the actions as in Q-learning, because the policy decisions are outside its control (beyond its role as a critic). Therefore, the policy network is the actor and the value network is the critic. To distinguish between the policy network and the Q-network, we will denote the parameter vector of the policy network by $\overline{\Theta}$, and that of the Q-network by $\overline{W}$.

We assume that the state at time stamp t is denoted by s_t, and the observable features of the state input to the neural network are denoted by $\overline{X}_t$. Therefore, we will use s_t and $\overline{X}_t$ interchangeably below. Consider a situation at the tth time-stamp, where the action a_t has been observed after state s_t with reward r_t. Then, the following sequence of steps is applied for the $(t+1)$th step:

1. Sample the action a_{t+1} using the current state of the parameters in the policy network. Note that the current state is s_{t+1} because the action a_t is already observed.

2. Let $F(\overline{X}_t, \overline{W}, a_t) = \hat{Q}^p(s_t, a_t)$ represent the estimated value of $Q^p(s_t, a_t)$ by the Q-network using the observed representation $\overline{X}_t$ of the states and parameters $\overline{W}$. Estimate $Q^p(s_t, a_t)$ and $Q^p(s_{t+1}, a_{t+1})$ using the Q-network. Compute the TD-error δ_t as follows:

$$\begin{aligned}\delta_t &= r_t + \gamma \hat{Q}^p(s_{t+1}, a_{t+1}) - \hat{Q}^p(s_t, a_t)\\ &= r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1}) - F(\overline{X}_t, \overline{W}, a_t)\end{aligned}$$

3. **[Update policy network parameters]:** Let $P(\overline{X}_t, \overline{\Theta}, a_t)$ be the probability of the action a_t predicted by policy network. Update the parameters of the policy network as follows:

$$\overline{\Theta} \leftarrow \overline{\Theta} + \alpha \hat{Q}^p(s_t, a_t) \nabla_\Theta \log(P(\overline{X}_t, \overline{\Theta}, a_t))$$

   Here, α is the learning rate for the policy network and the value of $\hat{Q}^p(s_t, a_t) = F(\overline{X}_t, \overline{W}, a_t)$ is obtained from the Q-network.

4. **[Update Q-Network parameters]:** Update the Q-network parameters as follows:

$$\overline{W} \Leftarrow \overline{W} + \beta \delta_t \nabla_W F(\overline{X}_t, \overline{W}, a_t)$$

   Here, β is the learning rate for the Q-network. A caveat is that the learning rate of the Q-network is generally higher than that of the policy network.

The action a_{t+1} is then executed in order to observe state s_{t+2}, and the value of t is incremented. The next iteration of the approach is executed (by repeating the above steps) at this incremented value of t. The iterations are repeated, so that the approach is executed to convergence. The value of $\hat{Q}^p(s_t, a_t)$ is the same as the value $\hat{V}^p(s_{t+1})$.

If we use $\hat{V}^p(s_t)$ as the baseline, the advantage $\hat{A}^p(s_t, a_t)$ is defined by the following:

$$\hat{A}^p(s_t, a_t) = \hat{Q}^p(s_t, a_t) - \hat{V}^p(s_t)$$

This changes the updates as follows:

$$\overline{\Theta} \leftarrow \overline{\Theta} + \alpha \hat{A}^p(s_t, a_t) \nabla_\Theta \log(P(\overline{X}_t, \overline{\Theta}, a_t))$$

Note the replacement of $\hat{Q}(s_t, a_t)$ in the original algorithm description with $\hat{A}(s_t, a_t)$. In order to estimate the value $\hat{V}^p(s_t)$, one possibility is to maintain another set of parameters representing the value network (which is different from the Q-network). The TD-algorithm can be used to update the parameters of the value network. However, it turns out that a single value-network is enough. This is because we can use $r_t + \gamma \hat{V}^p(s_{t+1})$ in lieu of $\hat{Q}(s_t, a_t)$. This results in an advantage function, which is the same as the TD-error:

$$\hat{A}^p(s_t, a_t) = r_t + \gamma \hat{V}^p(s_{t+1}) - \hat{V}^p(s_t)$$

In other words, we need the single value-network (cf. Figure 9.5), which serves as the critic. The above approach can also be generalized to use the $TD(\lambda)$ algorithm at any value of λ.

9.5.5 Continuous Action Spaces

The methods discussed to this point were all associated with discrete action spaces. For example, in a video game, one might have a discrete set of choices such as whether to move the cursor up, down, left, and right. However, in a robotics application, one might have continuous action spaces, in which we wish to move the robot's arm a certain distance. One possibility is to discretize the action into a set of fine-grained intervals, and use the midpoint of the interval as the representative value. One can then treat the problem as one of discrete choice. However, this is not a particularly satisfying design choice. First, the ordering among the different choices will be lost by treating inherently ordered (numerical) values as categorical values. Second, it blows up the space of possible actions, especially if the action space is multidimensional (e.g., separate dimensions for distances moved by the robot's arm and leg). Such an approach can cause overfitting, and greatly increase the amount of data required for learning.

A commonly used approach is to allow the neural network to output the parameters of a continuous distribution (e.g., mean and standard deviation of Gaussian), and then sample from the parameters of that distribution in order to compute the value of the action in the next step. Therefore, the neural network will output the mean μ and standard deviation σ for the distance moved by the robotic arm, and the actual action a will be sampled from the Gaussian $\mathcal{N}(\mu, \sigma)$ with this parameter:

$$a \sim \mathcal{N}(\mu, \sigma) \tag{9.26}$$

In this case, the action a represents the distance moved by the robot arm. The values of μ and σ can be learned using backpropagation. In some variations, σ is fixed up front as a hyper-parameter, with only the mean μ needing to be learned. The likelihood ratio trick also applies to this case, except that we use the logarithm of the density at a, rather than the discrete probability of the action a.

9.5.6 Advantages and Disadvantages of Policy Gradients

Policy gradient methods represent the most natural choice in applications like robotics that have continuous sequences of states and actions. For cases in which there are multidimensional and continuous action spaces, the number of possible combinations of actions can be very large. Since Q-learning methods require the computation of the maximum Q-value over all such actions, this step can turn out to be computationally intractable. Furthermore, policy gradient methods tend to be stable and have good convergence properties. However, policy gradient methods are susceptible to local minima. While Q-learning methods are less stable in terms of convergence behavior than are policy-gradient methods, and can sometimes oscillate around particular solutions, they have better capacity to reach near global optima.

Policy-gradient methods do possess one additional advantage in that they can learn stochastic policies, leading to better performance in settings where deterministic policies are known to be suboptimal (such as guessing games) due to being able to be exploited by the opponent. Q-learning provides deterministic policies, and so policy gradients are preferable in these settings because they provide a probability distribution on the possible actions from which the action is sampled.

9.6 Monte Carlo Tree Search

Monte Carlo tree search is a way of improving the strengths of learned policies and values at inference time by combining them with lookahead-based exploration. This improvement also provides a basis for lookahead-based bootstrapping like temporal difference learning. It is also leveraged as a probabilistic alternative to the deterministic minimax trees that are used by conventional game-playing software (although the applicability is not restricted to games). Each node in the tree corresponds to a state, and each branch corresponds to a possible action. The tree grows over time during the search as new states are encountered. The goal of the tree search is to select the best branch to recommend the predicted action of the agent. Each branch is associated with a value based on previous outcomes in tree search from that branch as well as an upper bound "bonus" that reduces with increased exploration. This value is used to set the priority of the branches during exploration. The learned goodness of a branch is adjusted after each exploration, so that branches leading to positive outcomes are favored in later explorations.

In the following, we will describe the Monte Carlo tree search used in *AlphaGo* as a case study for exposition. Assume that the probability $P(s, a)$ of each action (move) a at state (board position) s can be estimated using a policy network. At the same time, for each move we have a quantity $Q(s, a)$, which is the quality of the move a at state s. For example, the value of $Q(s, a)$ increases with increasing number of wins by following action a from state s in simulations. The *AlphaGo* system uses a more sophisticated algorithm that also incorporates some neural evaluations of the board position after a few moves (cf. Section 9.7.1). Then, in each iteration, the "upper bound" $u(s, a)$ of the quality of the move a at state s is given by the following:

$$u(s, a) = Q(s, a) + K \cdot \frac{P(s, a)\sqrt{\sum_b N(s, b)}}{N(s, a) + 1} \tag{9.27}$$

Here, $N(s, a)$ is the number of times that the action a was followed from state s over the course of the Monte Carlo tree search. In other words, the upper bound is obtained by starting with the quality $Q(s, a)$, and adding a "bonus" to it that depends on $P(s, a)$ and the number of times that branch is followed. The idea of scaling $P(s, a)$ by the number of visits is to discourage frequently visited branches and encourage greater exploration. The Monte Carlo approach is based on the strategy of selecting the branch with the largest upper bound, as in multi-armed bandit methods (cf. Section 9.2.3). Here, the second term on the right-hand side of Equation 9.27 plays the role of providing the confidence interval for computing the upper bound. As the branch is played more and more, the exploration "bonus" for that branch is reduced, because the width of its confidence interval drops. The hyperparameter K controls the degree of exploration.

At any given state, the action a with the largest value of $u(s, a)$ is followed. This approach is applied recursively until following the optimal action does not lead to an existing node. This new state s' is now added to the tree as a leaf node with initialized values of each $N(s', a)$ and $Q(s', a)$ set to 0. Note that the simulation up to a leaf node is fully deterministic, and no randomization is involved because $P(s, a)$ and $Q(s, a)$ are deterministically computable. Monte Carlo simulations are used to estimate the value of the newly added leaf node s'. Specifically, Monte Carlo rollouts from the policy network (e.g., using $P(s, a)$ to sample actions) return either $+1$ or -1, depending on win or loss. In Section 9.7.1, we will discuss some alternatives for leaf-node evaluation that use a value network as well. After evaluating the leaf node, the values of $Q(s'', a'')$ and $N(s'', a'')$ on all edges (s'', a'') on the path from the current state s to the leaf s' are updated. The value of $Q(s'', a'')$ is maintained as the average value of all the evaluations at leaf nodes reached from that branch during the Monte Carlo tree search. After multiple searches have been performed from s, the most visited edge is selected as the relevant one, and is reported as the desired action.

Use in Bootstrapping

Traditionally, Monte Carlo tree search has been used during inference rather than during training. However, since Monte Carlo tree search provides an improved estimate $Q(s, a)$ of the value of a state-action pair (as a result of lookaheads), it can also be used for bootstrapping (Intuition 9.4.1). Monte Carlo tree search provides an excellent alternative to n-step temporal-difference methods. One point about on-policy n-step temporal-difference methods is that they explore a single sequence of n-moves with the ϵ-greedy policy, and therefore tend to be too weak (with increased depth but not width of exploration). One way to strengthen them is to examine all possible n-sequences and use the optimal one with an off-policy technique (i.e., generalizing Bellman's 1-step approach). In fact, this was the approach used in Samuel's checkers program [421], which used the best option in the minimax tree for bootstrapping (and later referred to as *TD-Leaf* [22]). This results in increased complexity of exploring all possible n-sequences. Monte Carlo tree search can provide a robust alternative for bootstrapping, because it can explore multiple branches from a node to generate averaged target values. For example, the lookahead-based ground truth can use the averaged performance over all the explorations starting at a given node.

AlphaGo Zero [447] bootstraps policies rather than state values, which is extremely rare. *AlphaGo Zero* uses the relative visit probabilities of the branches at each node as *posterior* probabilities of the actions at that state. These posterior probabilities are improved over the probabilistic outputs of the policy network by virtue of the fact that the visit decisions use knowledge about the future (i.e., evaluations at deeper nodes of the Monte Carlo tree). The posterior probabilities are therefore bootstrapped as ground-truth values with respect to the policy network probabilities and used to update the weight parameters (cf. Section 9.7.1.1).

9.7 Case Studies

In the following, we present case studies from real domains to showcase different reinforcement learning settings. We will present examples of reinforcement learning in *Go*, robotics, conversational systems, self-driving cars, and neural-network hyperparameter learning.

9.7.1 AlphaGo: Championship Level Play at Go

Go is a two-person board game like chess. The complexity of a two-person board game largely depends on the size of the board and the number of valid moves at each position. The simplest example of a board game is tic-tac-toe with a 3×3 board, and most humans can solve it optimally without the need for a computer. Chess is a significantly more complex game with an 8×8 board, although clever variations of the brute-force approach of *selectively* exploring the minimax tree of moves up to a certain depth can perform significantly better than the best human today. *Go* occurs at the extreme end of complexity because of its 19×19 board.

Players play with white or black *stones*, which are kept in bowls next to the *Go* board. An example of a *Go* board is shown in Figure 9.7. The game starts with an empty board, and it fills up as players put stones on the board. Black makes the first move and starts with 181 stones in her bowl, whereas white starts with 180 stones. The total number of junctions is equal to the total number of stones in the bowls of the two players. A player places a stone of her color in each move at a particular position (from the bowl), and does not move it once it is placed. A stone of the opponent can be captured by encircling it. The objective of the game is for the player to control a larger part of the board than her opponent by encircling it with her stones.

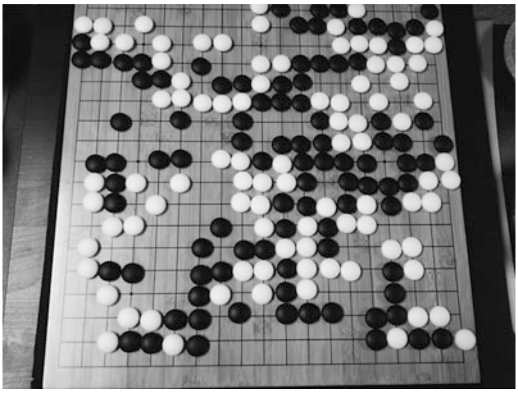


Figure 9.7: Example of a *Go* board with stones.

Whereas one can make about 35 possible moves (i.e., tree branch factor) in a particular position in chess, the average number of possible moves at a particular position in *Go* is 250, which is almost an order of magnitude larger. Furthermore, the average number of sequential moves (i.e., tree depth) of a game of *Go* is about 150, which is around twice as large as chess. All these aspects make *Go* a much harder candidate from the perspective of automated game-playing. The typical strategy of chess-playing software is to construct a minimax tree with all combinations of moves the players can make up to a certain depth, and then evaluate the final board positions with chess-specific heuristics (such as the amount of remaining material and the safety of various pieces). Suboptimal parts of the tree are pruned in a heuristic manner. This approach is simply a improved version of a brute-force strategy in which all possible positions are explored up to a given depth. The number of nodes in the minimax tree of *Go* is larger than the number of atoms in the observable universe, even at modest depths of analysis (20 moves for each player). As a result of the importance of spatial intuition in these settings, humans always perform better than brute force strategies at *Go*. The use of reinforcement learning in *Go* is much closer to what humans attempt to do. We rarely try to explore all possible combinations of moves; rather, we visually learn patterns on the board that are predictive of advantageous positions, and try to make moves in directions that are expected to improve our advantage.

The automated learning of spatial patterns that are predictive of good performance is achieved with a convolutional neural network. The state of the system is encoded in the board position at a particular point, although the board representation in *AlphaGo* includes some additional features about the status of junctions or the number of moves since a stone was played. Multiple such spatial maps are required in order to provide full knowledge of the state. For example, one feature map would represent the status of each intersection, another would encode the number of turns since a stone was played, and so on. Integer feature maps were encoded into multiple one-hot planes. Altogether, the game board could be represented using 48 binary planes of 19 × 19 pixels.

AlphaGo uses its win-loss experience with repeated game playing (both using the moves of expert players and with games played against itself) to learn good policies for moves in various positions with a policy network. Furthermore, the evaluation of each position on the *Go* board is achieved with a value network. Subsequently, Monte Carlo tree search is used for final inference. Therefore, *AlphaGo* is a multi-stage model, whose components are discussed in the following sections.

Policy Networks

The policy network takes as its input the aforementioned visual representation of the board, and outputs the probability of action a in state s. This output probability is denoted by $p(s, a)$. Note that the actions in the game of *Go* correspond to the probability of placing a stone at each legal position on the board. Therefore, the output layer uses the softmax activation. Two separate policy networks are trained using different approaches. The two networks were identical in structure, containing convolutional layers with ReLU nonlinearities. Each network contained 13 layers. Most of the convolutional layers convolve with 3×3 filters, except for the first and final convolutions. The first and final filters convolve with 5×5 and 1×1 filters, respectively. The convolutional layers were zero padded to maintain their size, and 192 filters were used. The ReLU nonlinearity was used, and no maxpooling was used in order to maintain the spatial footprint.

The networks were trained in the following two ways:

- *Supervised learning:* Randomly chosen samples from expert players were used as training data. The input was the state of the network, while the output was the action performed by the expert player. The score (advantage) of such a move was always +1, because the goal was to train the network to *imitate* expert moves, which is also referred to as *imitation learning*. Therefore, the neural network was backpropagated with the log-likelihood of the probability of the chosen move as its gain. This network is referred to as the SL-policy network. It is noteworthy that these supervised forms of imitation learning are often quite common in reinforcement learning for avoiding cold-start problems. However, subsequent work [446] showed that dispensing with this part of the learning was a better option.

- *Reinforcement learning:* In this case, reinforcement learning was used to train the network. One issue is that *Go* needs two opponents, and therefore the network was played against itself in order to generate the moves. The current network was always played against a randomly chosen network from a few iterations back, so that the reinforcement learning could have a pool of randomized opponents. The game was played until the very end, and then an advantage of +1 or −1 was associated with each move depending on win or loss. This data was then used to train the policy network. This network was referred to as the RL-policy network.

Note that these networks were already quite formidable *Go* players compared to state-of-the-art software, and they were combined with Monte Carlo tree search to strengthen them.

Value Networks

This network was also a convolutional neural network, which uses the state of the network as the input and the predicted score in $[-1, +1]$ as output, where $+1$ indicates a perfect probability of 1. The output is the predicted score of the next player, whether it is white or black, and therefore the input also encodes the "color" of the pieces in terms of "player" or "opponent" rather than white or black. The architecture of the value network was very similar to the policy network, except that there were some differences in terms of the input and output. The input contained an additional feature corresponding to whether the next player to play was white or black. The score was computed using a single tanh unit at the end, and therefore the value lies in the range $[-1, +1]$. The early convolutional layers of the value network are the same as those in the policy network, although an additional convolutional layer is added in layer 12. A fully connected layer with 256 units and ReLU activation follows the final convolutional layer. In order to train the network, one possibility is to use positions from a data set [606] of *Go* games. However, the preferred choice was to generate the data set using self-play with the SL-policy and RL-policy networks all the way to the end, so that the final outcomes were generated. The state-outcome pairs were used to train the convolutional neural network. Since the positions in a single game are correlated, using them sequentially in training causes overfitting. It was important to sample positions from different games in order to prevent overfitting caused by closely related training examples. Therefore, each training example was obtained from a distinct game of self-play.

Monte Carlo Tree Search

A simplified variant of Equation 9.27 was used for exploration, which is equivalent to setting $K = 1/\sqrt{\sum_b N(s, b)}$ at each node s. Section 9.6 described a version of the Monte Carlo tree search method in which only the RL-policy network is used for evaluating leaf nodes. In the case of *AlphaGo*, two approaches are combined. First, fast Monte Carlo rollouts were used from the leaf node to create evaluation e_1. While it is possible to use the policy network for rollout, *AlphaGo* trained a simplified softmax classifier with a database of human games and some hand-crafted features for faster speed of rollouts. Second, the value network created a separate evaluation e_2 of the leaf nodes. The final evaluation e is a convex combination of the two evaluations as $e = \beta e_1 + (1 - \beta) e_2$. The value of $\beta = 0.5$ provided the best performance, although using only the value network also provided closely matching performance (and a viable alternative). The most visited branch in Monte Carlo tree search was reported as the predicted move.

9.7.1.1 Alpha Zero: Enhancements to Zero Human Knowledge

A later enhancement of the idea, referred to as *AlphaGo Zero* [446], removed the need for human expert moves (or an SL-network). Instead of separate policy and value networks, a single network outputs both the policy (i.e., action probabilities) $p(s, a)$ and the value $v(s)$ of the position. The cross-entropy loss on the output policy probabilities and the squared loss on the value output were added to create a single loss. Whereas the original version of *AlphaGo* used Monte Carlo tree search only for inference from trained networks, the zero-knowledge versions also use the visit counts in Monte Carlo tree search for training. One can view the visit count of each branch in tree search as a policy *improvement* operator over $p(s, a)$ by virtue of its lookahead-based exploration. This provides a basis for creating bootstrapped ground-truth values (Intuition 9.4.1) for neural network learning. While temporal difference learning bootstraps state values, this approach bootstraps visit counts for learning policies. The predicted probability of Monte Carlo tree search for action a in board state s is $\pi(s, a) \propto N(s, a)^{1/\tau}$, where τ is a temperature parameter. The value of $N(s, a)$ is computed using a similar Monte Carlo search algorithm as used for *AlphaGo*, where the *prior* probabilities $p(s, a)$ output by the neural network are used for computing Equation 9.27. The value of $Q(s, a)$ in Equation 9.27 is set to the average value output $v(s')$ from the neural network of the newly created leaf nodes s' reached from state s.

AlphaGo Zero updates the neural network by bootstrapping $\pi(s, a)$ as a ground-truth, whereas ground-truth *state values* are generated with Monte Carlo simulations. At each state s, the probabilities $\pi(s, a)$, values $Q(s, a)$ and visit counts $N(s, a)$ are updated by running the Monte Carlo tree search procedure (repeatedly) starting at state s. The neural network from the previous iteration is used for selecting branches according to Equation 9.27 until a state is reached that does not exist in the tree or a terminal state is reached. For each non-existing state, a new leaf is added to the tree with its Q-values and visit values initialized to zero. The Q-values and visit counts of all edges on the path from s to the leaf node are updated based on leaf evaluation by the neural network (or by game rules for terminal states). After multiple searches starting from node s, the *posterior* probability $\pi(s, a)$ is used to sample an action for self-play and reach the next node s'. The entire procedure discussed in this paragraph is repeated at node s' to recursively obtain the next position s''. The game is recursively played to completion and the final value from $\{-1, +1\}$ is returned as the ground-truth value $z(s)$ of uniformly sampled states s on the game path. Note that $z(s)$ is defined from the perspective of the player at state s. The ground-truth values of the probabilities are already available in $\pi(s, a)$ for various values of a. Therefore, one can create a training instance for the neural network containing the input representation of state s, the bootstrapped ground-truth probabilities in $\pi(s, a)$, and the Monte Carlo ground-truth value $z(s)$. This training instance is used to update the neural network parameters. Therefore, if the probability and value outputs for the neural network are $p(s, a)$ and $v(s)$, respectively, the loss for a neural network with weight vector $\overline{W}$ is as follows:

$$L = [v(s) - z(s)]^2 - \sum_a \pi(s, a) \log[p(s, a)] + \lambda ||\overline{W}||^2 \tag{9.28}$$

Here, $\lambda > 0$ is the regularization parameter.

Further advancements were proposed in the form of *Alpha Zero* [447], which could play multiple games such as *Go*, shogi, and chess. *Alpha Zero* has handily defeated the best chess-playing software, *Stockfish*, and has also defeated the best shogi software (*Elmo*). The victory in chess was particularly unexpected by most top players, because it was always assumed that chess required too much domain knowledge for a reinforcement learning system to win over a system with hand-crafted evaluations.

Comments on Performance

AlphaGo has shown extraordinary performance against a variety of computer and human opponents. Against a variety of computer opponents, it won 494 out of 495 games [445]. Even when *AlphaGo* was handicapped by providing four free stones to the opponent, it won 77%, 86%, and 99% of the games played against (the software programs named) *Crazy Stone*, *Zen*, and *Pachi*, respectively. It also defeated notable human opponents, such as the European champion, the World champion, and the top-ranked player.

A more notable aspect of its performance was the way in which it achieved its victories. In several of its games, *AlphaGo* made many unconventional and brilliantly unorthodox moves, which would sometimes make sense only in hindsight after the victory of the program [607, 608]. There were cases in which the moves made by *AlphaGo* were contrary to conventional wisdom, but eventually revealed innovative insights acquired by *AlphaGo* during self-play. After this match, some top *Go* players reconsidered their approach to the entire game.

The performance of *Alpha Zero* in chess was similar, where it often made material sacrifices in order to incrementally improve its position and constrict its opponent. This type of behavior is a hallmark of human play and is very different from conventional chess software (which is already much better than humans). Unlike hand-crafted evaluations, it seemed to have no pre-conceived notions on the material values of pieces, or on when a king was safe in the center of the board. Furthermore, it discovered most well-known chess openings on its own using self-play, and seemed to have its own opinions on which ones were "better." In other words, it had the ability to discover knowledge on its own. A key difference of reinforcement learning from supervised learning is that *it has the ability to innovate beyond known knowledge through learning by reward-guided trial and error.* This behavior represents some promise in other applications.

9.7.2 Self-Learning Robots

Self-learning robots represent an important frontier in artificial intelligence, in which robots can be trained to perform various tasks such as locomotion, mechanical repairs, or object retrieval by using a reward-driven approach. For example, consider the case in which one has constructed a robot that is *physically* capable of locomotion (in terms of how it is constructed and the movement choices available to it), but it has to learn the precise *choice* of movements in order to keep itself balanced and move from point A to point B. As bipedal humans, we are able to walk and keep our balance naturally without even thinking about it, but this is not a simple matter for a bipedal robot in which an incorrect choice of joint movement could easily cause it to topple over. The problem becomes even more difficult when uncertain terrain and obstacles are placed in the way of a robot.

This type of problem is naturally suited to reinforcement learning, because it is easy to judge whether a robot is walking correctly, but it is hard to specify precise rules about what the robot should do in every possible situation. In the reward-driven approach of reinforcement learning, the robot is given (virtual) rewards every time it makes progress in locomotion from point A to point B. Otherwise, the robot is free to take any actions, and it is not pre-trained with knowledge about the specific choice of actions that would help keep it balanced and walk. In other words, the robot is not seeded with any knowledge of what walking looks like (beyond the fact that it will be rewarded for using its available actions for making progress from point A to point B). This is a classical example of reinforcement learning, because the robot now needs to learn the specific sequence of actions to take in order to earn the goal-driven rewards. Although we use locomotion as a specific example in this case, this general principle applies to any type of learning in robots. For example, a second problem is that of teaching a robot manipulation tasks such as grasping an object or screwing the cap on a bottle. In the following, we will provide a brief discussion of both cases.

9.7.2.1 Deep Learning of Locomotion Skills

In this case, locomotion skills were taught to virtual robots [433], in which the robot was simulated with the *MuJoCo* physics engine [609], which stands for *Multi-Joint Dynamics with Contact.* It is a physics engine aiming to facilitate research and development in robotics, biomechanics, graphics, and animation, where fast and accurate simulation is needed without having to construct an actual robot. Both a humanoid and a quadruped robot were used. An example of the biped model is shown in Figure 9.8. The advantage of this type of simulation is that it is inexpensive to work with a virtual simulation, and one avoids the natural safety and expense issues that arise with the physical damages in an experimentation framework that is likely to be marred by high levels of mistakes/accidents. On the flip side, a physical model provides more realistic results. In general, a simulation can often be used for smaller scale testing before building a physical model.

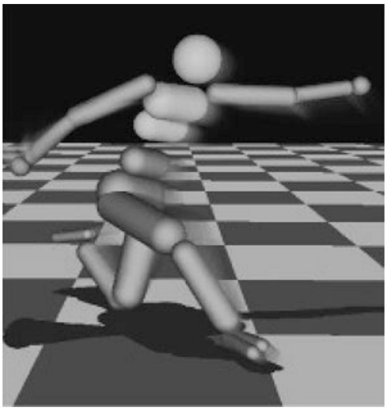


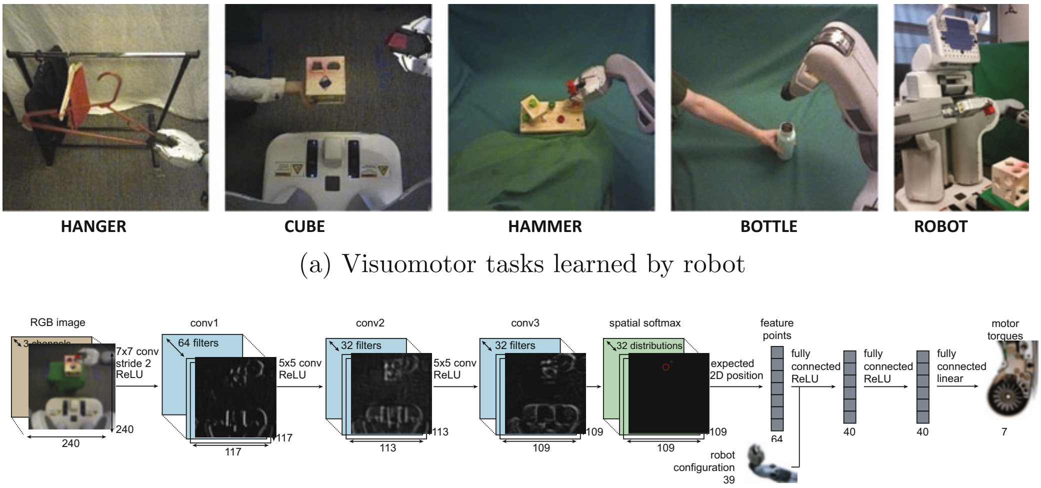
Figure 9.8: Example of the virtual humanoid robot. Original image is available at [609].

The humanoid model has 33 state dimensions and 10 actuated degrees of freedom, while the quadruped model has 29 state dimensions and 8 actuated degrees of freedom. Models were rewarded for forward progress, although episodes were terminated when the center of mass of the robot fell below a certain point. The actions of the robot were controlled by joint torques. A number of features were available to the robot, such as sensors providing the positions of obstacles, the joint positions, angles, and so on. These features were fed into the neural networks. Two neural networks were used; one was used for value estimation, and the other was used for policy estimation. Therefore, a policy gradient method was used in which the value network played the role of estimating the advantage. Such an approach is an instantiation of an actor-critic method.

A feed-forward neural network was used with three hidden layers, with 100, 50, and 25 tanh units, respectively. The approach in [433] requires the estimation of both a policy function and a value function, and the same architecture was used in both cases for the hidden layers. However, the value estimator required only one output, whereas the policy estimator required as many outputs as the number of actions. Therefore, the main difference between the two architectures was in terms of how the output layer and the loss function was designed. The generalized advantage estimator (GAE) was used in combination with trust-based policy optimization (TRPO). The bibliographic notes contain pointers to specific details of these methods. On training the neural network for 1000 iterations with reinforcement learning, the robot learned to walk with a visually pleasing gait. A video of the final results of the robot walking is available at [610]. Similar results were also later released by Google DeepMind with more extensive abilities of avoiding obstacles or other challenges [187].

9.7.2.2 Deep Learning of Visuomotor Skills

A second and interesting case of reinforcement learning is provided in [286], in which a robot was trained for several household tasks such as placing a coat hanger on a rack, inserting a block into a shape-sorting cube, fitting the claw of a toy hammer under a nail with various grasps, and screwing a cap onto a bottle. Examples of these tasks are illustrated in Figure 9.9(a) along with an image of the robot. The actions were 7-dimensional joint motor torque commands, and each action required a sequence of commands in order to optimally perform the task. In this case, an actual physical model of a robot was used for training. A camera image was used by the robot in order to locate the objects and manipulate them. This camera image can be considered the robot's eyes, and the convolutional neural network used by the robot works on the same conceptual principle as the visual cortex (based on Hubel and Wiesel's experiments). Even though this setting seems very different from that of the Atari video games at first sight, there are significant similarities in terms of how image frames can help in mapping to policy actions. For example, the Atari setting also works with a convolutional neural network on the raw pixels. However, there were some additional inputs here, corresponding to the robot and object positions. These tasks require a high level of learning in visual perception, coordination, and contact dynamics, all of which need to learned automatically.



(a) Visuomotor tasks learned by robot

(b) Architecture of the convolutional neural network

Figure 9.9: Deep learning of visuomotor skills. These figures appear in [286]. (©2016 Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel)

A natural approach is to use a convolutional neural network for mapping image frames to actions. As in the case of Atari games, spatial features need to be learned in the layers of the convolutional neural network that are suitable for earning the relevant rewards in a task-sensitive manner. The convolutional neural network had 7 layers and 92,000 parameters. The first three layers were convolutional layers, the fourth layer was a spatial softmax, and the fifth layer was a fixed transformation from spatial feature maps to a concise set of two coordinates. The idea was to apply a softmax function to the responses across the spatial feature map. This provides a probability of each position in the feature map. The expected position using this probability distribution provides the 2-dimensional coordinate, which is referred to as a *feature point*. Note that each spatial feature map in the convolution layer creates a feature point. The feature point can be viewed as a kind of soft argmax over the spatial probability distribution. The fifth layer was quite different from what one normally sees in a convolutional neural network, and was designed to create a precise representation of the visual scene that was suitable for feedback control. The spatial feature points are concatenated with the robot's configuration, which is an additional input occurring only after the convolution layers. This concatenated feature set is fed into two fully connected layers, each with 40 rectified units, followed by linear connections to the torques. Note that only the observations corresponding to the camera were fed to the first layer of the convolutional neural network, and the observations corresponding to the robot state were fed to the first fully connected layer. This is because the convolutional layers cannot make much use of the robot states, and it makes sense to concatenate the state-centric inputs after the visual inputs have been processed by the convolutional layers. The entire network contained about 92,000 parameters, of which 86,000 were in the convolutional layers. The architecture of the convolutional neural network is shown in Figure 9.9(b). The observations consist of the RGB camera image, joint encoder readings, velocities, and end-effector pose.

The full robot states contained between 14 and 32 dimensions, such as the joint angles, end-effector pose, object positions, and their velocities. This provided a practical notion of a state. As in all policy-based methods, the outputs correspond to the various actions (motor torques). One interesting aspect of the approach discussed in [286] is that it transforms the reinforcement learning problem into supervised learning. A *guided policy search* method was used, which is not discussed in this chapter. This approach converts portions of the reinforcement learning problem into supervised learning. Interested readers are referred to [286], where a video of the performance of the robot (trained using this system) may also be found.

9.7.3 Building Conversational Systems: Deep Learning for Chatbots

Chatbots are also referred to as *conversational systems* or *dialog systems*. The ultimate goal of a chatbot is to build an agent that can freely converse with a human about a variety of topics in a natural way. We are very far from achieving this goal. However, significant progress has been made in building chatbots for specific domains and particular applications (e.g., negotiation or shopping assistant). An example of a relatively general-purpose system is Apple's Siri, which is a digital personal assistant. One can view Siri as an open-domain system, because it is possible to have conversations with it about a wide variety of topics. It is reasonably clear to anyone using Siri that the assistant is sometimes either unable to provide satisfactory responses to difficult questions, and in some cases hilarious responses to common questions are hard-coded. This is, of course, natural because the system is relatively general-purpose, and we are nowhere close to building a human-level conversational system. In contrast, closed-domain systems have a specific task in mind, and can therefore be more easily trained in a reliable way.

In the following, we will describe a system built by *Facebook* for end-to-end learning of negotiation skills [290]. This is a closed-domain system because it is designed for the particular purpose of negotiation. As a test-bed, the following negotiation task was used. Two agents are shown a collection of items of different types (e.g., two books, one hat, three balls). The agents are instructed to divide these items among themselves by negotiating a split of the items. A key point is that the value of each of the types of items is different for the two agents, but they are not aware of the value of the items for each other. This is often the case in real-life negotiations, where users attempt to reach a mutually satisfactory outcome by negotiating for items of value to them.

The values of the items are always assumed to be non-negative and generated randomly in the test-bed under some constraints. First, the total value of all items for a user is 10. Second, each item has non-zero value to at least one user so that it makes little sense to ignore an item. Last, some items have nonzero values to both users. Because of these constraints, it is impossible for both users to achieve the maximum score of 10, which ensures a competitive negotiation process. After 10 turns, the agents are allowed the option to complete the negotiation with no agreement, which has a value of 0 points for both users. The three item types of books, hats, and balls were used, and a total of between 5 and 7 items existed in the pool. The fact that the values of the items are different for the two users (without knowledge about each other's assigned values) is significant; if both negotiators are capable, they will be able to achieve a total value of larger than 10 for the items between them. Nevertheless, the better negotiator will be able to capture the larger value by optimally negotiating for items with a high value for them.

The reward function for this reinforcement learning setting is the final value of the items attained by the user. One can use supervised learning on previous dialogs in order to maximize the likelihood of utterances. A straightforward use of recurrent networks to maximize the likelihood of utterances resulted in agents that were too eager to compromise. Therefore, the approach combined supervised learning with reinforcement learning. The incorporation of supervised learning within the reinforcement learning helps in ensuring that the models do not diverge from human language. A form of planning for dialogs called *dialog roll-out* was introduced. The approach uses an encoder-decoder recurrent architecture, in which the decoder maximizes the reward function rather than the likelihood of utterances. This encoder-decoder architecture is based on sequence-to-sequence learning, as discussed in Section 7.7.2 of Chapter 7.

To facilitate supervised learning, dialogs were collected from *Amazon Mechanical Turk*. A total of 5808 dialogs were collected in 2236 unique scenarios, where a scenario is defined by assignment of a particular set of values to the items. Of these cases, 252 scenarios corresponding to 526 dialogs were held out. Each scenario results in two training examples, which are derived from the perspective of each agent. A concrete training example could be one in which the items to be divided among the two agents correspond to 3 books, 2 hats, and 1 ball. These are part of the input to each agent. The second input could be the value of each item to the agent, which are (i) Agent A: book:1, hat:3, ball:1, and (ii) Agent B: book:2, hat:1, ball:2. Note that this means that agent A should secretly try to get as many hats as possible in the negotiation, whereas agent B should focus on books and balls. An example of a dialog in the training data is given below [290]:

Agent A: I want the books and the hats, you get the ball.

Agent B: Give me a book too and we have a deal.

Agent A: Ok, deal.

Agent B: ⟨choose⟩

The final output for agent A is 2 books and 2 hats, whereas the final output for agent B is 1 book and 1 ball. Therefore, each agent has her own set of inputs and outputs, and the dialogs for each agent are also viewed from their own perspective in terms of the portions that are reads and the portions that are writes. Therefore, each scenario generates two training examples and the same recurrent network is shared for generating the writes and the final output of each agent. The dialog x is a list of tokens $x_0 \ldots x_T$, containing the turns of each agent interleaved with symbols marking whether the turn was written by an agent or their partner. A special token at the end indicates that one agent has marked that an agreement has been reached.

The supervised learning procedure uses four different gated recurrent units (GRUs). The first gated recurrent unit GRU_g encodes the input goals, the second gated recurrent unit GRU_q generates the terms in the dialog, a forward-output gated recurrent unit $GRU_{\overrightarrow{o}}$, and a backward-output gated recurrent unit $GRU_{\overleftarrow{o}}$. The output is essentially produced by a bi-directional GRU. These GRUs are hooked up in end-to-end fashion. In the supervised learning approach, the parameters are trained using the inputs, dialogs, and outputs available from the training data. The loss for the supervised model for a weighted sum of the token-prediction loss of the dialog and the output choice prediction loss of the items.

However, for reinforcement learning, dialog roll-outs are used. Note that the group of GRUs in the supervised model is, in essence, providing probabilistic outputs. Therefore, one can adapt the same model to work for reinforcement learning by simply changing the loss function. In other words, the GRU combination can be considered a type of policy network. One can use this policy network to generate Monte Carlo roll-outs of various dialogs and their final rewards. Each of the sampled actions becomes a part of the training data, and the action is associated with the final reward of the roll-out. In other words, the approach uses *self-play* in which the agent negotiates with itself to learn better strategies. The final reward achieved by a roll-out is used to update the policy network parameters. This reward is computed based on the value of the items negotiated at the end of the dialog. This approach can be viewed as an instance of the REINFORCE algorithm [533]. One issue with self-play is that the agents tend to learn their own language, which deviates from natural human language when both sides use reinforcement learning. Therefore, one of the agents is constrained to be a supervised model.

For the final prediction, one possibility is to directly sample from the probabilities output by the GRU. However, such an approach is often not optimal when working with recurrent networks. Therefore, a two-stage approach is used. First, c candidate utterances are created by using sampling. The expected reward of each candidate utterance is computed and the one with the largest expected value is selected. In order to compute the expected reward, the output was scaled by the probability of the dialog because low-probability dialogs were unlikely to be selected by either agent.

A number of interesting observations were made in [290] about the performance of the approach. First, the supervised learning methods often tended to give up easily, whereas the reinforcement learning methods were more persistent in attempting to obtain a good deal. Second, the reinforcement learning method would often exhibit human-like negotiation tactics. In some cases, it feigned interest in an item that was not really of much value in order to obtain a better deal for another item.

9.7.4 Self-Driving Cars

As in the case of the robot locomotion task, the car is rewarded for progressing from point A to point B without causing accidents or other undesirable road incidents. The car is equipped with various types of video, audio, proximity, and motion sensors in order to record observations. The objective of the reinforcement learning system is for the car to go from point A to point B safely irrespective of road conditions.

Driving is a task for which it is hard to specify the proper rules of action in every situation; on the other hand, it is relatively easy to judge when one is driving correctly. This is precisely the setting that is well suited to reinforcement learning. Although a fully self-driving car would have a vast array of components corresponding to inputs and sensors of various types, we focus on a simplified setting in which a single camera is used [46, 47]. This system is instructive because it shows that even a single front-facing camera is sufficient to accomplish quite a lot when paired with reinforcement learning. Interestingly, this work was inspired by the 1989 work of Pomerleau [381], who built the *Autonomous Land Vehicle in a Neural Network (ALVINN)* system, and the main difference from the work done over 25 years back was one of increased data and computational power. In addition, the work uses some advances in convolutional neural networks for modeling. Therefore, this work showcases the great importance of increased data and computational power in building reinforcement learning systems.

The training data was collected by driving in a wide variety of roads and conditions. The data was collected primarily from central New Jersey, although highway data was also collected from Illinois, Michigan, Pennsylvania, and New York. Although a single front-facing camera in the driver position was used as the primary data source for making decisions, the training phase used two additional cameras at other positions in the front to collect rotated and shifted images. These auxiliary cameras, which were not used for final decision making, were however useful for collecting additional data. The placement of the additional cameras ensured that their images were shifted and rotated, and therefore they could be used to train the network to recognize cases where the car position had been compromised. In short, these cameras were useful for data augmentation. The neural network was trained to minimize the error between the steering command output by the network and the command output by the human driver. Note that this approach tends to make the approach closer to supervised learning rather than reinforcement learning. These types of learning methods are also referred to as *imitation learning* [427]. Imitation learning is often used as a first step to buffer the cold-start inherent in reinforcement learning systems.

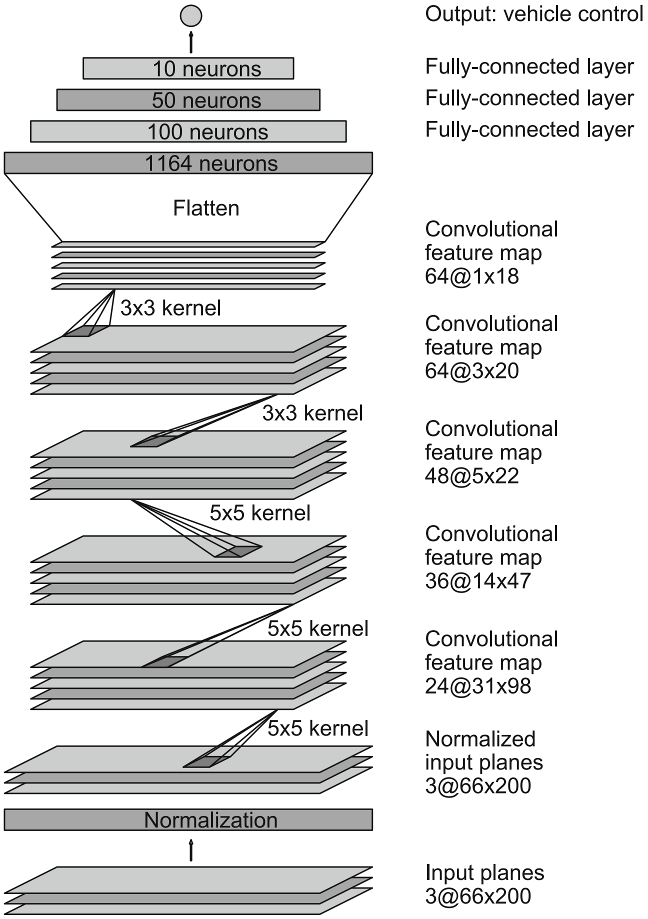


Figure 9.10: The neural network architecture of the control system in the self-driving car discussed in [46] (Courtesy NVIDIA).

Scenarios involving imitation learning are often similar to those involving reinforcement learning. It is relatively easy to use reinforcement setting in this scenario by giving a reward when the car makes progress without human intervention. On the other hand, if the car either does not make progress or requires human intervention, it is penalized. However, this does not seem to be the way in which the self-driving system of [46, 47] is trained. One issue with settings like self-driving cars is that one always has to account for safety issues during training. Although published details on most of the available self-driving cars are limited, it seems that supervised learning has been the method of choice compared to reinforcement learning in this setting. Nevertheless, the differences between using supervised learning and reinforcement learning are not significant in terms of the broader architecture of the neural network that would be useful. A general discussion of reinforcement learning in the context of self-driving cars may be found in [612].

The convolutional neural network architecture is shown in Figure 9.10. The network consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. The first convolutional layer used a 5×5 filter with a stride of 2. The next two convolutional layers each used non-strided convolution with a 3×3 filter. These convolutional layers were followed with three fully connected layers. The final output value was a control value, corresponding to the inverse turning radius. The network had 27 million connections and 250, 000 parameters. Specific details of how the deep neural network performs the steering are provided in [47].

The resulting car was tested both in simulation and in actual road conditions. A human driver was always present in the road tests to perform interventions when necessary. On this basis, a measure was computed on the percentage of time that human intervention was required. It was found that the vehicle was autonomous 98% of the time. A video demonstration of this type of autonomous driving is available in [611]. Some interesting observations were obtained by visualizing the activation maps of the trained convolutional neural network (based on the methodology discussed in Chapter 8). In particular, it was observed that the features were heavily biased towards learning aspects of the image that were important to driving. In the case of unpaved roads, the feature activation maps were able to detect the outlines of the roads. On the other hand, if the car was located in a forest, the feature activation maps were full of noise. Note that this does not happen in a convolutional neural network that is trained on *ImageNet* because the feature activation maps would typically contain useful characteristics of trees, leaves, and so on. This difference in the two cases is because the convolutional network of the self-driving setting is trained in a goal-driven matter, and it learns to detect features that are relevant to driving. The specific characteristics of the trees in a forest are not relevant to driving.

9.7.5 Inferring Neural Architectures with Reinforcement Learning

An interesting application of reinforcement learning is to learn the neural network architecture for performing a specific task. For discussion purposes, let us consider a setting in which we wish to determine the structure of a convolutional neural architecture for classify a data set like CIFAR-10 [583]. Clearly, the structure of the neural network depends on a number of hyper-parameters, such as the number of filters, filter height, filter width, stride height, and stride width. These parameters depend on one another, and the parameters of later layers depend on those from earlier layers.

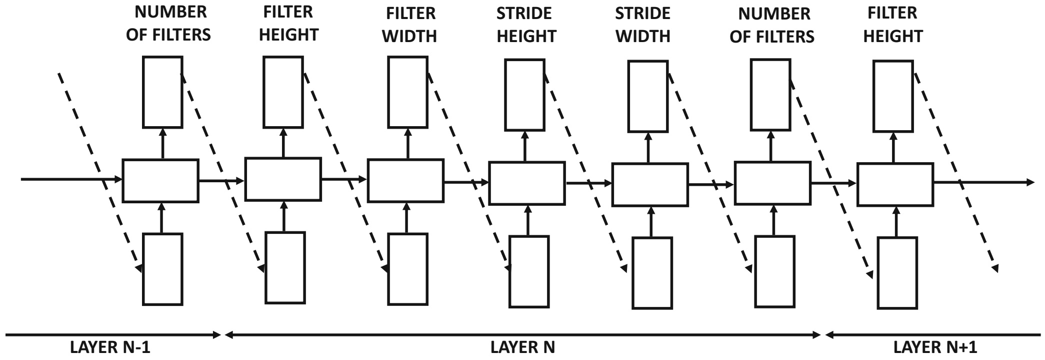


Figure 9.11: The controller network for learning the convolutional architecture of the child network [569]. The controller network is trained with the REINFORCE algorithm.

The reinforcement learning method uses a recurrent network as the *controller* to decide the parameters of the convolutional network, which is also referred to as the *child network* [569]. The overall architecture of the recurrent network is illustrated in Figure 9.11.

The choice of a recurrent network is motivated by the sequential dependence between different architectural parameters. The softmax classifier is used to predict each output as a token rather than a numerical value. This token is then used as an input into the next layer, which is shown by the dashed lines in Figure 9.11. The generation of the parameter as a token results in a discrete action space, which is generally more common in reinforcement learning as compared to a continuous action space.

The performance of the child network on a validation set drawn from CIFAR-10 is used to generate the reward signal. Note that the child network needs to be trained on the CIFAR-10 data set in order to test its accuracy. Therefore, this process requires a full training procedure of the child network, which is quite expensive. This reward signal is used in conjunction with the REINFORCE algorithm in order to train the parameters of the controller network. Therefore, the controller network is really the policy network in this case, which generates a sequence of inter-dependent parameters.

A key point is about the number of layers of the child network (which also decides the number of layers in the recurrent network). This value is not held constant but it follows a certain schedule as training progresses. In the early iterations, the number of layers is fewer, and therefore the learned architecture of the convolutional network is shallow. As training progresses, the number of layers slowly increases over time. The policy gradient method is not very different from what is discussed earlier in this chapter, except that a recurrent network is trained with the reward signal rather than a feed-forward network. Various types of optimizations are also discussed in [569], such as efficient implementations with parallelism and the learning of advanced architectural designs like skip connections.

9.8 Practical Challenges Associated with Safety

Simplifying the design of highly complex learning algorithms with reinforcement learning can sometimes have unexpected effects. By virtue of the fact that reinforcement learning systems have larger levels of freedom than other learning systems, it naturally leads to some safety related concerns. While biological greed is a powerful factor in human intelligence, it is also a source of many undesirable aspects of human behavior. The simplicity that is the greatest strength of reward-driven learning is also its greatest pitfall in biological systems. Simulating such systems therefore results in similar pitfalls from the perspective of artificial intelligence. For example, poorly designed rewards can lead to unforeseen consequences, because of the exploratory way in which the system learns its actions. Reinforcement learning systems can frequently learn unknown "cheats" and "hacks" in imperfectly designed video games, which tells us a cautionary tale of what might happen in a less-than-perfect real world. Robots learn that simply pretending to screw caps on bottles can earn faster rewards, as long as the human or automated evaluator is fooled by the action. In other words, the design of the reward function is sometimes not a simple matter.

Furthermore, a system might try to earn virtual rewards in an "unethical" way. For example, a cleaning robot might try to earn rewards by first creating messes and then cleaning them [10]. One can imagine even darker scenarios for robot nurses. Interestingly, these types of behaviors are sometimes also exhibited by humans. These undesirable similarities are a direct result of simplifying the learning process in machines by leveraging the simple greed-centric principles with which biological organisms learn. Striving for simplicity results in ceding more control to the machine, which can have unexpected effects. In some cases, there are ethical dilemmas in even designing the reward function. For example, if it becomes inevitable that an accident is going to occur, should a self-driving car save its driver or two pedestrians? Most humans would save themselves in this setting as a matter of reflexive biological instinct; however, it is an entirely different matter to incentivize a learning system to do so. At the same time, it would be hard to convince a human operator to trust a vehicle where her safety is not the first priority for the learning system. Reinforcement learning systems are also susceptible to the ways in which their human operators interact with them and manipulate the effects of their underlying reward function; there have been occasions where a chatbot was taught to make offensive or racist remarks.

Learning systems have a harder time in generalizing their experiences to new situations. This problem is referred to as *distributional shift*. For example, a self-driving car trained in one country might perform poorly in another. Similarly, the exploratory actions in reinforcement learning can sometimes be dangerous. Imagine a robot trying to solder wires in an electronic device, where the wires are surrounded with fragile electronic components. Trying exploratory actions in this setting is fraught with perils. These issues tell us that we cannot build AI systems with no regard to safety. Indeed, some organizations like *OpenAI* [613] have taken the lead in these matters of ensuring safety. Some of these issues are also discussed in [10] with broader frameworks of possible solutions. In many cases, it seems that the human would have to be involved in the loop to some extent in order to ensure safety [424].

9.9 Summary

This chapter studies the problem of reinforcement learning in which agents interact with the environment in a reward-driven manner in order to learn the optimal actions. There are several classes of reinforcement learning methods, of which the Q-learning methods and the policy-driven methods are the most common. Policy-driven methods have become increasingly popular in recent years. Many of these methods are end-to-end systems that integrate deep neural networks to take in sensory inputs and learn policies that optimize rewards. Reinforcement learning algorithms are used in many settings like playing video or other types of games, robotics, and self-driving cars. The ability of these algorithms to learn via experimentation often leads to innovative solutions that are not possible with other forms of learning. Reinforcement learning algorithms also pose unique challenges associated with safety because of the oversimplification of the learning process with reward functions.

9.10 Bibliographic Notes

An excellent overview on reinforcement learning may be found in the book by Sutton and Barto [483]. A number of surveys on reinforcement learning are available at [293]. David Silver's lectures on reinforcement learning are freely available on *YouTube* [619]. The method of temporal differences was proposed by Samuel in the context of a checkers program [421] and formalized by Sutton [482]. Q-learning was proposed by Watkins in [519], and a convergence proof is provided in [520]. The SARSA algorithm was introduced in [412]. Early methods for using neural networks in reinforcement learning were proposed in [296, 349, 492, 493, 494]. The work in [492] developed TD-Gammon, which was a backgammon playing program.

A system that used a convolutional neural network to create a deep Q-learning algorithm with raw pixels was pioneered in [335, 336]. It has been suggested in [335] that the approach presented in the paper can be improved with other well-known ideas such as prioritized sweeping [343]. Asynchronous methods that use multiple agents in order to perform the learning are discussed in [337]. The use of multiple asynchronous threads avoids the problem of correlation within a thread, which improves convergence to higher-quality solutions. This type of asynchronous approach is often used in lieu of the experience replay technique. Furthermore, an n-step technique, which uses a lookahead of n steps (instead of 1 step) to predict the Q-values, was proposed in the same work.

One drawback of Q-learning is that it is known to overestimate the values of actions under certain circumstances. An improvement over Q-learning, referred to as *double Q-learning*, was proposed in [174]. In the original form of Q-learning, the same values are used to both select and evaluate an action. In the case of double Q-learning, these values are decoupled, and therefore one is now learning two separate values for selection and evaluation. This change tends to make the approach less sensitive to the overestimation problem. The use of prioritized experience replay to improve the performance of reinforcement learning algorithms under sparse data is discussed in [428]. Such an approach significantly improves the performance of the system on Atari games.

In recent years, policy gradients have become more popular than Q-learning methods. An interesting and simplified description of this approach for the Atari game of *Pong* is provided in [605]. Early methods for using finite difference methods for policy gradients are discussed in [142, 355]. Likelihood methods for policy gradients were pioneered by the REINFORCE algorithm [533]. A number of analytical results on this class of algorithms are provided in [484]. Policy gradients have been used in for learning in the game of *Go* [445], although the overall approach combines a number of different elements. Natural policy gradients were proposed in [230]. One such method [432] has been shown to perform well at learning locomotion in robots. The use of *generalized advantage estimation (GAE)* with continuous rewards is discussed in [433]. The approach in [432, 433] uses natural policy gradients for optimization, and the approach is referred to as *trust region policy optimization* (TRPO). The basic idea is that bad steps in learning are penalized more severely in reinforcement learning (than supervised learning) because the quality of the collected data worsens. Therefore, the TRPO method prefers second-order methods with conjugate gradients (see Chapter 3), in which the updates tend to stay within good regions of trust. Surveys are also available on specific types of reinforcement learning methods like actor-critic methods [162].

Monte Carlo tree search was proposed in [246]. Subsequently, it was used in the game of *Go* [135, 346, 445, 446]. A survey on these methods may be found in [52]. Later versions of *AlphaGo* dispensed with the supervised portions of learning, adapted to chess and shogi, and performed better with zero initial knowledge [446, 447]. The *AlphaGo* approach combines several ideas, including the use of policy networks, Monte Carlo tree search, and convolutional neural networks. The use of convolutional neural networks for playing the game of *Go* has been explored in [73, 307, 481]. Many of these methods use supervised learning in order to mimic human experts at *Go*. Some TD-learning methods for chess, such as *NeuroChess* [496], *KnightCap* [22], and *Giraffe* [259] have been explored, but were not as successful as conventional engines. The pairing of convolutional neural networks and reinforcement learning for spatial games seems to be a new (and successful) recipe that distinguishes *Alpha Zero* from these methods. Several methods for training self-learning robots are presented in [286, 432, 433]. An overview of deep reinforcement learning methods for dialog generation is provided in [291]. Conversation models that use only supervised learning with recurrent networks are discussed in [440, 508]. The negotiation chatbot discussed in this chapter is described in [290]. The description of self-driving cars is based on [46, 47]. An MIT course on self-driving cars is available at [612]. Reinforcement learning has also been used to generate structured queries from natural language [563], or for learning neural architectures in various tasks [19, 569].

Reinforcement learning can also improve deep learning models. This is achieved with the notion of *attention* [338, 540], in which reinforcement learning is used to focus on selective parts of the data. The idea is that large parts of the data are often irrelevant for learning, and learning how to focus on selective portions of the data can significantly improve results. The selection of relevant portions of the data is achieved with reinforcement learning. Attention mechanisms are discussed in Section 10.2 of Chapter 10. In this sense, reinforcement learning is one of the topics in machine learning that is more tightly integrated with deep learning than seems at first sight.

9.10.1 Software Resources and Testbeds

Although significant progress has been made in designing reinforcement learning algorithms in recent years, commercial software using these methods is still relatively limited. Nevertheless, numerous software testbeds are available that can be used in order to test various algorithms. Perhaps the best source for high-quality reinforcement learning baselines is available from *OpenAI* [623]. *TensorFlow* [624] and *Keras* [625] implementations of reinforcement learning algorithms are also available.

Most frameworks for testing and development of reinforcement learning algorithms are specialized to specific types of reinforcement learning scenarios. Some frameworks are lightweight, and can be used for quick testing. For example, the ELF framework [498], created by *Facebook*, is designed for real-time strategy games, and is an open-source and light-weight reinforcement learning framework. The *OpenAI Gym* [620] provides environments for development of reinforcement learning algorithms for Atari games and simulated robots. The *OpenAI Universe* [621] can be used to turn reinforcement learning programs into Gym environments. For example, self-driving car simulations have been added to this environment. An Arcade learning environment for developing agents in the context of Atari games is described in [25]. The *MuJoCo* simulator [609], which stands for Multi-Joint dynamics with Contact, is a physics engine, and is designed for robotics simulations. An application with the use of *MuJoCo* is described in this chapter. *ParlAI* [622] is an open-source framework for dialog research by *Facebook*, and is implemented in Python. Baidu has created an open-source platform of its self-driving car project, referred to as *Apollo* [626].

9.11 Exercises

1. The chapter gives a proof of the likelihood ratio trick (cf. Equation 9.24) for the case in which the action a is discrete. Generalize this result to continuous-valued actions.

2. Throughout this chapter, a neural network, referred to as the policy network, has been used in order to implement the policy gradient. Discuss the importance of the choice of network architecture in different settings.

3. You have two slot machines, each of which has an array of 100 lights. The probability distribution of the reward from playing each machine is an unknown (and possibly machine-specific) function of the pattern of lights that are currently lit up. Playing a slot machine changes its light pattern in some well-defined but unknown way. Discuss why this problem is more difficult than the multi-armed bandit problem. Design a deep learning solution to optimally choose machines in each trial that will maximize the average reward per trial at steady-state.

4. Consider the well-known game of rock-paper-scissors. Human players often try to use the history of previous moves to guess the next move. Would you use a Q-learning or a policy-based method to learn to play this game? Why? Now consider a situation in which a human player samples one of the three moves with a probability that is an unknown function of the history of 10 previous moves of each side. Propose a deep learning method that is designed to play with such an opponent. Would a well-designed deep learning method have an advantage over this human player? What policy should a human player use to ensure probabilistic parity with a deep learning opponent?

5. Consider the game of tic-tac-toe in which a reward drawn from $\{-1, 0, +1\}$ is given at the end of the game. Suppose you learn the values of all states (assuming optimal play from both sides). Discuss why states in non-terminal positions will have non-zero value. What does this tell you about credit-assignment of intermediate moves to the reward value received at the end?

6. Write a Q-learning implementation that learns the value of each state-action pair for a game of tic-tac-toe by repeatedly playing against human opponents. No function approximators are used and therefore the entire table of state-action pairs is learned using Equation 9.5. Assume that you can initialize each Q-value to 0 in the table.

7. The two-step TD-error is defined as follows:

$$\delta_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t)$$

(a) Propose a TD-learning algorithm for the 2-step case.

(b) Propose an on-policy n-step learning algorithm like SARSA. Show that the update is truncated variant of Equation 9.16 after setting $\lambda = 1$. What happens for the case when $n = \infty$?

(c) Propose an off-policy n-step learning algorithm like Q-learning and discuss its advantages/disadvantages with respect to (b).