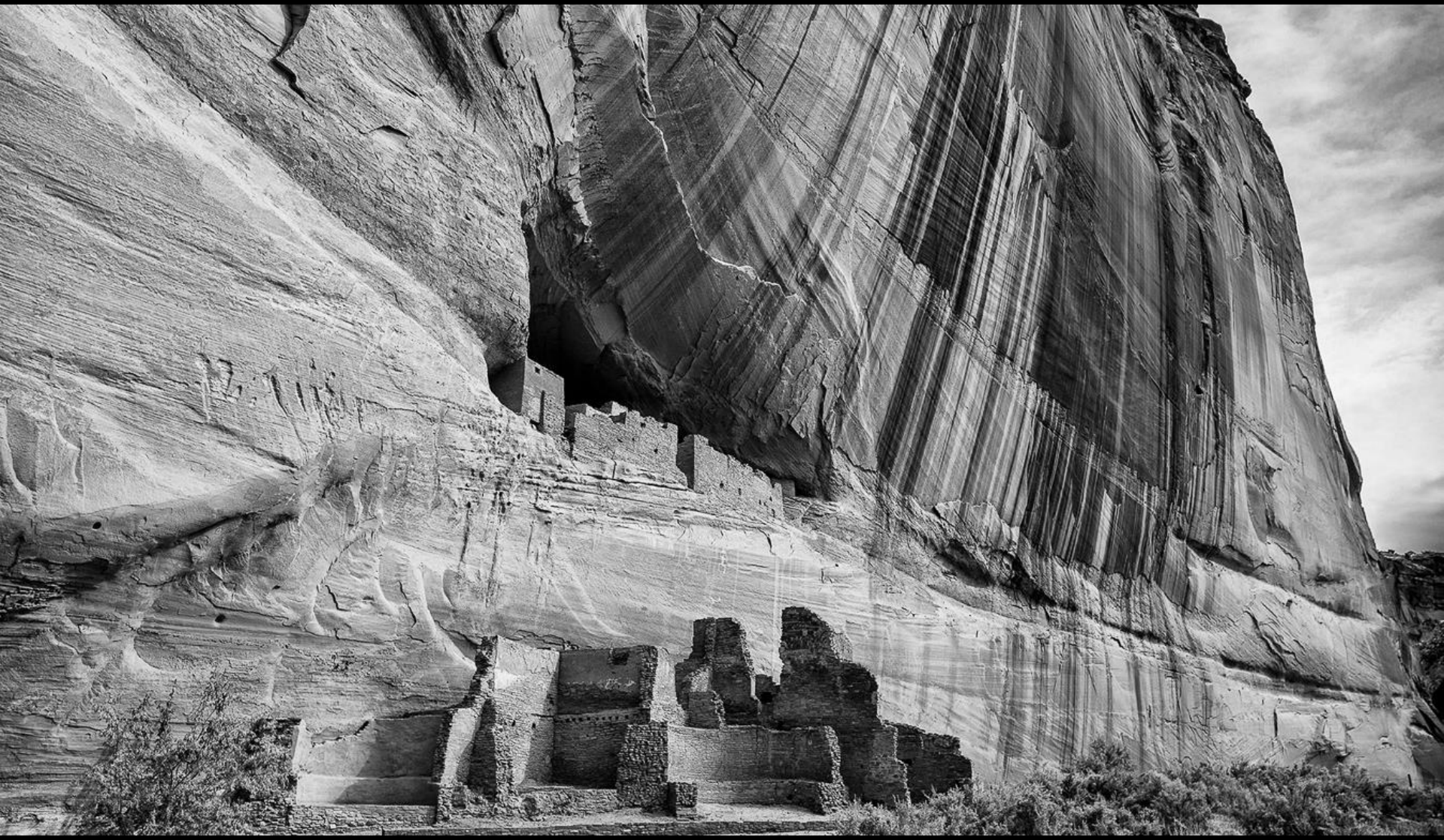
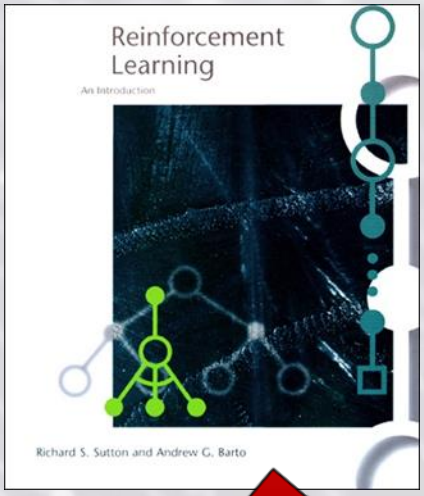


Reinforcement Learning & Deep RL
CS 446/546

Outline

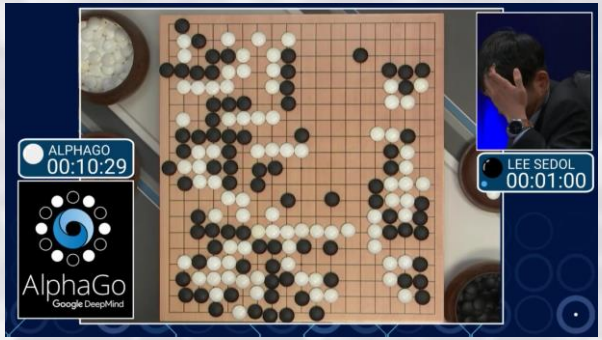
- Introduction
- n-Armed Bandits
- The Formal RL Problem
- Dynamic Programming
- Monte Carlo Methods
- Temporal-Difference Learning
- Deep RL

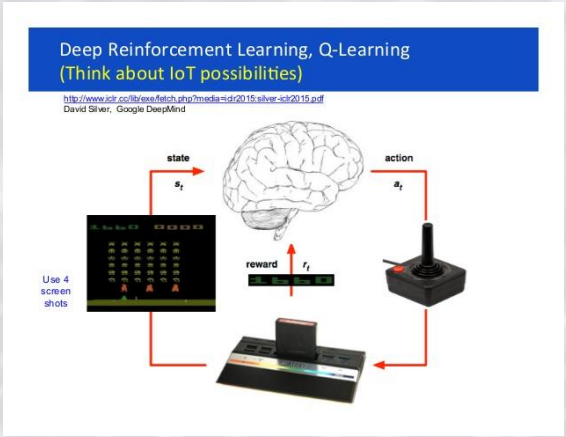


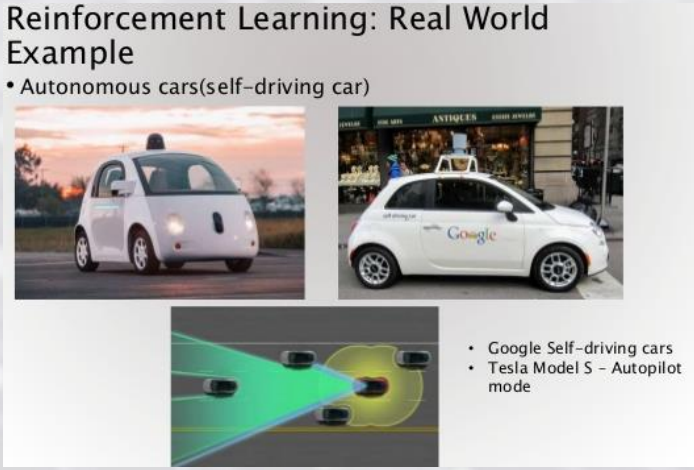
Recommended Text on RL:
Sutton/Barto

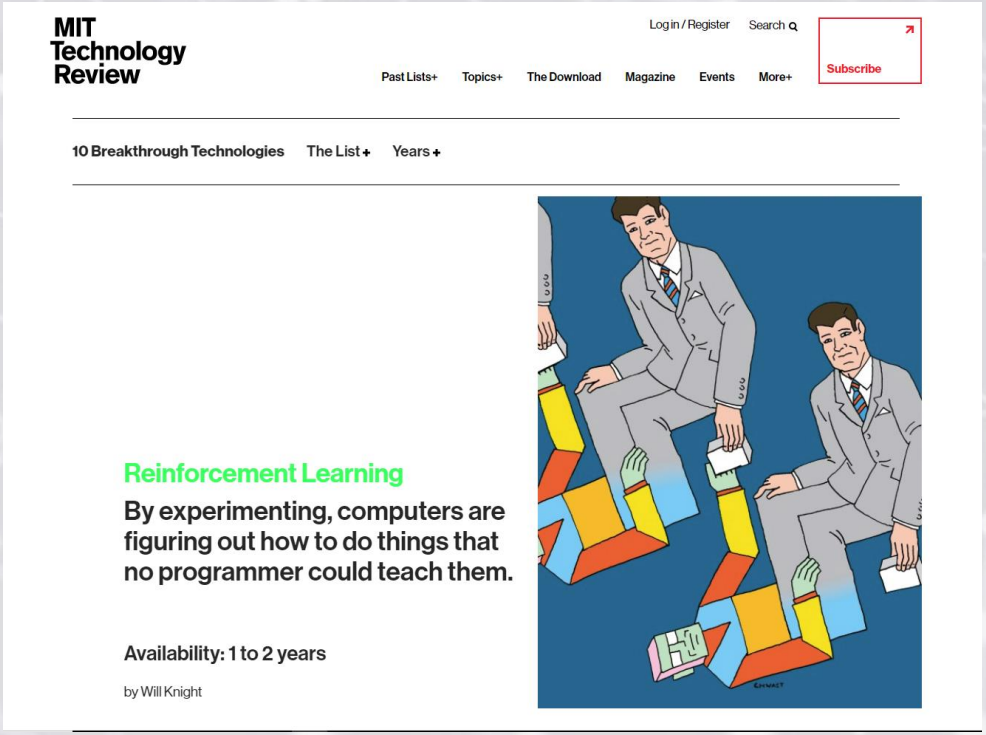
*http://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

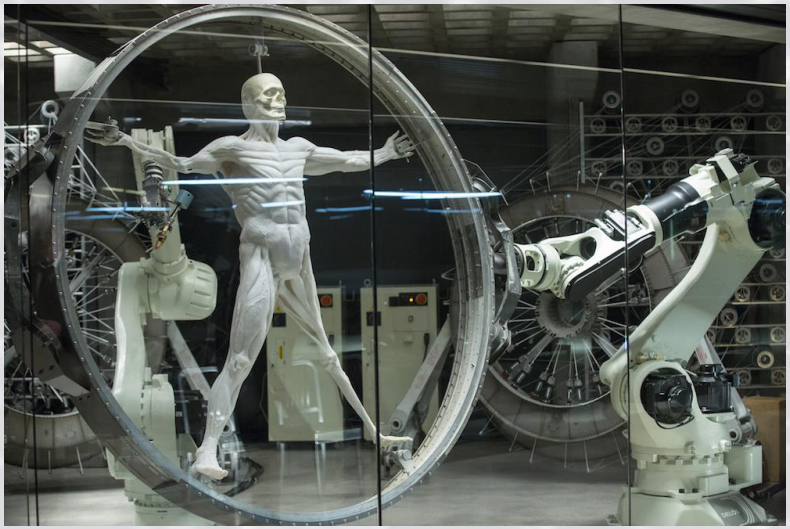
It's an exciting time for RL!







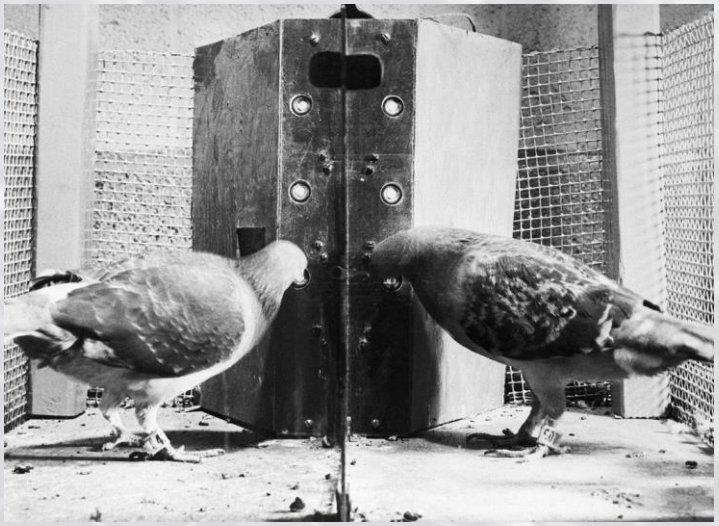




SkyNet

Introduction

- **Reinforcement learning** (RL) is *learning what to do* – how to map situations to actions – so as to maximize a numerical reward signal.



- The learner is not told what actions to take (unlike most forms of ML), but instead they must *discover* which actions yield the most reward by trying them.

- Most often, actions may affect not only the immediate reward but also the next situation, and through that, all subsequent rewards.

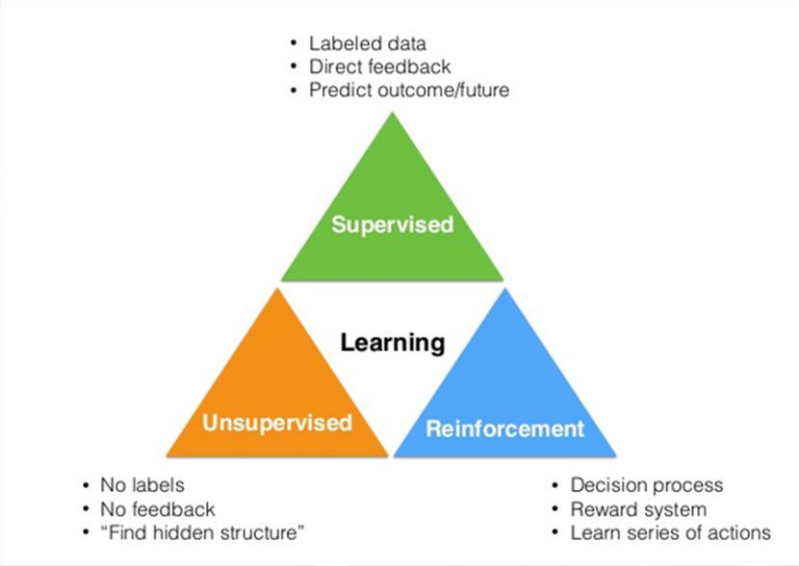
- These two characteristics – **trial-and-error search** and **delayed reward** – are the two most important distinguishing features of RL.

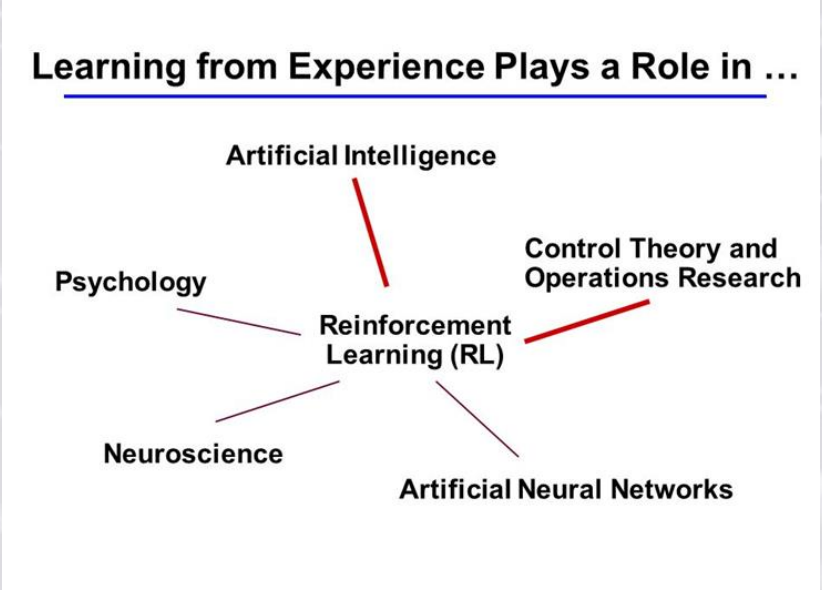
Introduction

- RL is different from *supervised learning*, which is generally inadequate for learning from *interaction*.

- In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act.

- In uncharted territory – where one would expect learning to be most beneficial – <u>an agent must be able to learn from its own experience</u>.



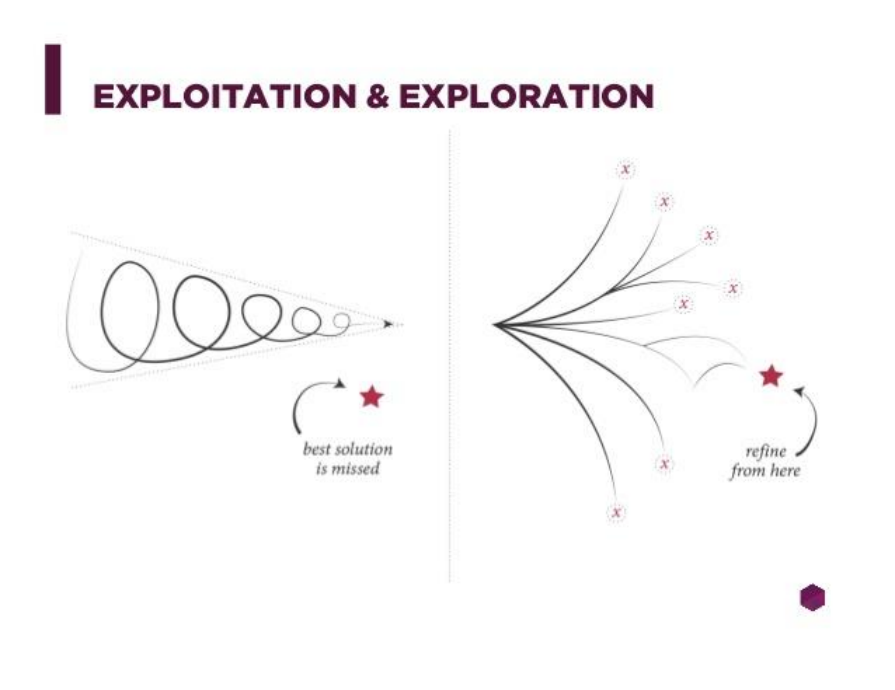


Introduction

- One of the emblematic challenges that arises in RL is the **trade-off between exploration and exploitation**.

- To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward.

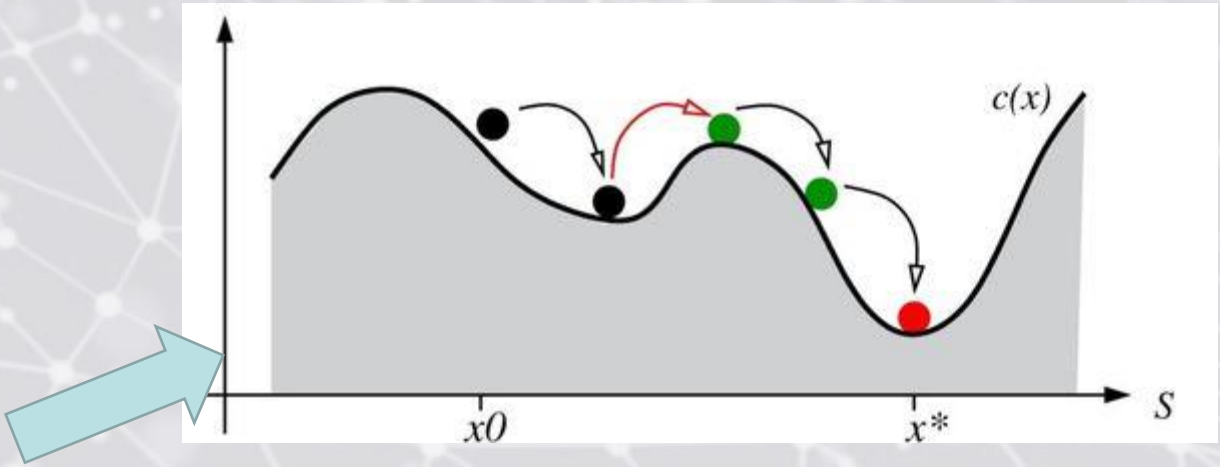
- However, in order to discover such actions, the agent has to try actions that it has not selected before.



Introduction

- Thus the agent has to *exploit* what it already knows in order to obtain rewards, but it also to *explore* in order to make between action selections in the future.

- The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must instead try a variety of actions and progressively favor those that appear to be best.



* **Simulated annealing** is a classic algorithm that makes use of both exploratory and exploitative steps.

Introduction

- Another key feature of RL is that it explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment (this is in contrast to other approaches that consider subproblems without addressing they fit into the larger picture).

- RL starts with an interactive, goal-seeking agent; the agent has explicit goals and can choose actions that influence their environment.

- The most important features distinguishing RL learning from other types of learning is that it uses training information that **evaluates** the actions taken rather than **instructs** by giving correct actions.

Introduction

- Another key feature of RL is that it explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment (this is in contrast to other approaches that consider subproblems without addressing they fit into the larger picture).

- RL starts with an interactive, goal-seeking agent; the agent has explicit goals and can choose actions that influence their environment.

- The most important features distinguishing RL learning from other types of learning is that it uses training information that **evaluates** the actions taken rather than **instructs** by giving correct actions.

- This is what creates the need for active explorations, for an explicit trial-and-error search for good behavior. Purely evaluative feedback indicates how good the action taken is, but not whether it is the best or worst action possible.

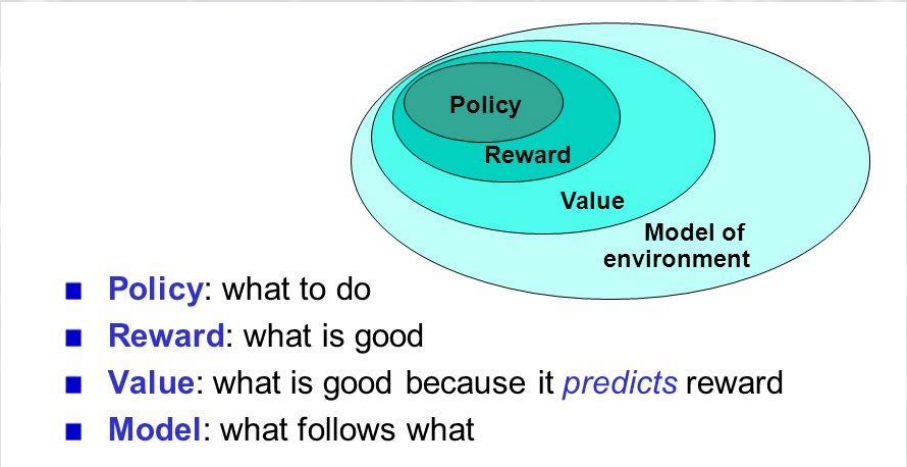
- Purely instructive feedback, on the other hand, indicates the correct action to take, independently of the action actually taken.

Introduction: Elements of RL

• In general, there are (4) main components of an RL system: (1) a *policy*, (2) a *reward function*, (3) a *value function* and (4) (optionally) a *model* of the environment.

(1) A **policy** defines the learning agent's way of behaving at a given time; a policy is a mapping from perceived states of the environment to actions to be taken when in those states (it corresponds to what in psychology would be called a set of *stimulus-response rules* or associations).

• In some cases the policy may be a simple function or lookup table, or it may involve extensive computation such as a search process. The policy is the core of an RL learning agent; note that policies may be deterministic or stochastic.



Introduction: Elements of RL

• In general, there are (4) main components of an RL system: (1) a *policy*, (2) a *reward function*, (3) a *value function* and (4) (optionally) a *model* of the environment.

(2) A **reward function** defined the goal in a RL problem. The reward function maps each perceived state (e.g. state-action pair) of the environment to a single number, a reward, indicating the intrinsic desirability of that state.

• The RL agent's sole object is to maximize the total reward received in the long run. The reward function is unalterable by the agent; it may, however, serve as a basis for altering the policy; reward functions may be stochastic.

Introduction: Elements of RL

• In general, there are (4) main components of an RL system: (1) a *policy*, (2) a *reward function*, (3) a *value function* and (4) (optionally) a *model* of the environment.

(3) Whereas a reward function indicates what is good in an immediate sense, a **value function** specifies what is good in the long run.

• Generally speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

• A state might, for example, always yield a low immediate reward but nevertheless have a high value because it is regularly followed by other states that yield high rewards (or the reverse could hold).

Introduction: Elements of RL

(3) Where as a reward function indicates what is good in an immediate sense, a **value function** specifies what is good in the long run.

• Generally speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

• A state might, for example, always yield a low immediate reward but nevertheless have a high value because it is regularly followed by other states that yield high rewards (or the reverse could hold).

• Rewards are in a sense primary – without rewards there could be no values. Nevertheless, it is values with which we are most concerned when making evaluating decisions. Action decisions are made based on value judgements; we seek actions that bring about states of highest value, not states of highest reward.

NB: In practice it is usually much harder to determine values than rewards.

Introduction: Elements of RL

• In general, there are (4) main components of an RL system: (1) a *policy*, (2) a *reward function*, (3) a *value function* and (4) (optionally) a *model* of the environment.

(4) A **model** (optional) mimics the behavior of the environment.

• For example, given a state and action, the model might predict the resultant next state and next reward.

• Models are used for *planning*, in which case an agent may consider possible future situations before they are actually experienced.

* Historically, early RL systems were explicitly trial-and-error learners; more recently researchers regularly incorporate models and planning into RL systems.

Introduction: Elements of RL

• In general, there are (4) main components of an RL system: (1) a *policy*, (2) a *reward function*, (3) a *value function* and (4) (optionally) a *model* of the environment.

Recapping:

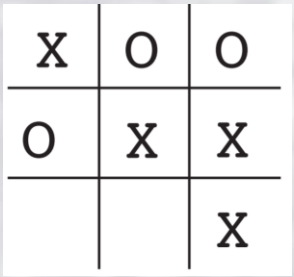
(1) Policy: defines the agent; a mapping from states → actions (e.g., given a screen configuration in space invaders, tell me what to do)

(2) Reward: mapping from states to numbers; goal of agent is to maximum reward in the long run. (e.g., agent receives +1 for winning chess game; 0 for draw and -1 for loss)

(3) Value function: value of a state is the total amount of reward expected over time, starting from this state. (e.g., the value of a chess board configuration one step removed from a forced checkmate is very large)

(4) Model: mapping from state-action pairs to new states (e.g., a physics model for an environment in which a robot is moving)

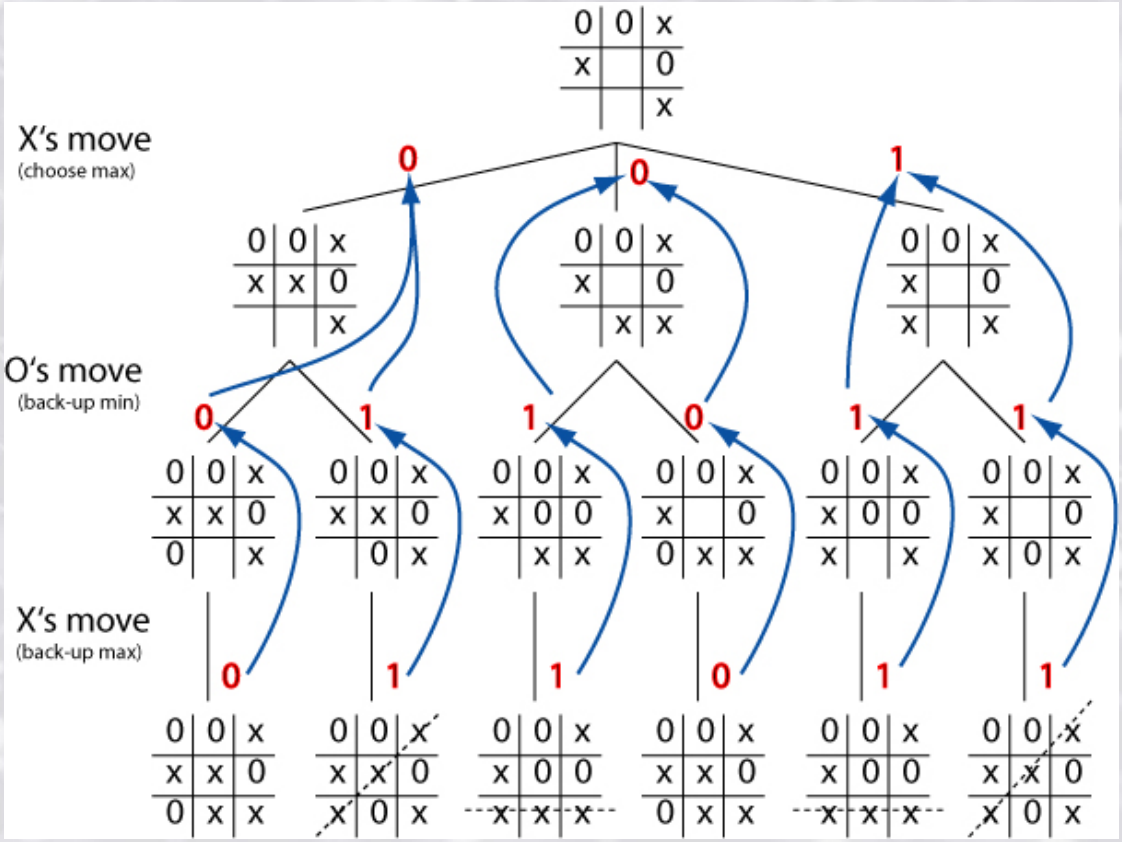
Introduction: Example



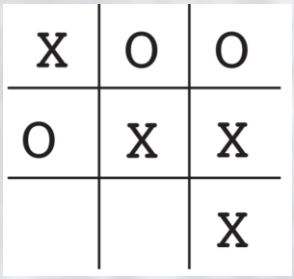
• Consider the classic tic-tac-toe game; how might we construct a player that will find the imperfections in its opponent's play (we assume a fallible opponent)?

Introduction: Example

• Despite its simplicity, tic-tac-toe cannot readily be solved in a satisfactory way using purely classical techniques. For example, **minimax** (a common AI algorithm for adversarial games) assumes a particular way of playing for an opponent; classical optimization on the other hand requires a complete specification of the opponent (including every probability of a move for a particular game state); a GA would search over the entire space of possible policies.



Introduction: Example



• How might we construct a player that will find the imperfections in its opponent's play?

• Using RL, we could generate a table of numbers, one for each possible state of the game; each number in the table represents the latest estimate of the probability of our winning from that state. We treat this estimate as the state's *value*.

• Next we play many games against the opponent. To select our moves we examine the states that would result from each of our possible moves; most of the time we move *greedily*, selecting the move that leads to the state with the greatest value.

• Occasionally, however, we select randomly from among the other moves instead; these are *exploratory moves* that cause us to experience states that we might otherwise never see.

Introduction: Example

• While we are playing, we change the values of the states in which we find ourselves during the game; we attempt to make more accurate estimates of the probabilities of winning.

• To do this, we "back up" the value of the state after each greedy move to the state before the move. More precisely, we move the earlier state's value a fraction of the way toward the value of the later state.

• Let *s* denote the state before the greedy move and *s'* the state after the move; then the update to the estimates value of V(s) is given by:

$$V(s) \leftarrow V(s) + \alpha\left[V(s') - V(s)\right]$$

Where α is the step-size parameter; this update rule is an example of **temporal-difference** (TD) learning.

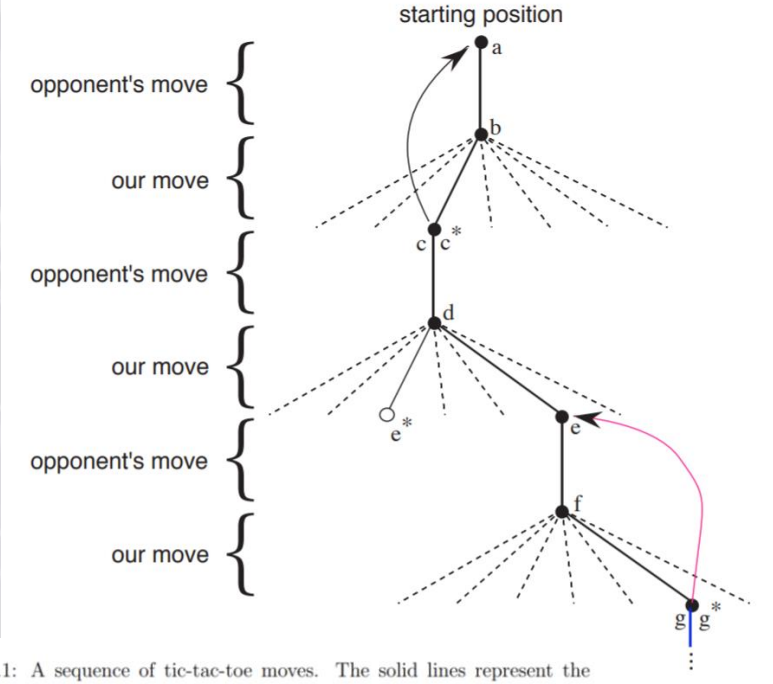


Figure 1.1: A sequence of tic-tac-toe moves. The solid lines represent the moves taken during a game; the dashed lines represent moves that we (our reinforcement learning player) considered but did not make. Our second move was an exploratory move, meaning that it was taken even though another sibling move, the one leading to e*, was ranked higher. Exploratory moves do not result in any learning, but each of our other moves does, causing *backups* as suggested by the curved arrows and detailed in the text.

Introduction: Example

• The aforementioned method performs quite well on this task. In particular, if the step-size parameter is reduced appropriately over time, this method converges – for any fixed opponent – to the true probabilities of winning from each state given an optimal opponent.

• This simple example illustrates an essential different between evolutionary methods and methods that learn value functions.

• To evaluate a policy, an evolutionary method must hold it fixed and play many games (or simulate many games) using a model of the opponent.

• The frequency of wins gives an unbiased estimate of the probability of winning with that policy; however, each policy change is made only after many games, and only the final outcome of each games is used. Moreover, what happens during the games is ignored (i.e. for a winning match, *all* of the agent's actions are given credit for the win).

Introduction: Example

• By contrast, with RL, value function methods allow individual states to be evaluated. Learning a value function takes explicit advantage of information available during the course of play.

• With RL, there is an emphasis on learning while interacting with an environment; in addition, there is a clear goal, and correct behavior requires planning or foresight that takes into account delayed effects of one's choices.

Introduction: Example

• By contrast, with RL, value function methods allow individual states to be evaluated. Learning a value function takes explicit advantage of information available during the course of play.

• With RL, there is an emphasis on learning while interacting with an environment; in addition, there is a clear goal, and correct behavior requires planning or foresight that takes into account delayed effects of one's choices.

* It is a striking feature of RL that it can achieve the effects of planning and lookahead without using a model of the opponent and without conducting an explicit search over possible sequences of future states and actions.

* RL can also be applied in non-episodic environments (e.g. when agent behavior continues indefinitely); furthermore, RL can be used in the absence of an external adversary, i.e. in the case of a "game against nature."

n-Armed Bandits & Action-Value Methods

• Consider the following learning problem:

You are faced repeatedly with a choice among *n* different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections. Each action selection is called a play.

* This is the original form of the **n-armed bandit problem**. Each action selection is like a play of one of the slot machine levers, and the rewards are the payoffs for hitting the jackpot. Through repeated plays you are to maximize your winnings by concentrating your play on the best levers.

• Generally, each machine payout follows a probability distribution, p_i, with mean μ_i; the agent should identify the machine with the largest μ_i.

n-Armed Bandits & Action-Value Methods

• In the n-armed bandit problem, each action has an expected or mean reward give that a particular action is selected; call this the *value* of the action.

• Naturally, if we knew the exact value of each action, solving the problem would be trivial.

• If you maintain estimates of the action values, then at any time there is at least one action whose estimated value is greatest; call this the *greedy action*.

n-Armed Bandits & Action-Value Methods

• In the n-armed bandit problem, each action has an expected or mean reward give that a particular action is selected; call this the *value* of the action.

• Naturally, if we knew the exact value of each action, solving the problem would be trivial.

• If you maintain estimates of the action values, then at any time there is at least one action whose estimated value is greatest; call this the *greedy action*.

•If you select the greedy action, you are **exploiting** your current knowledge of the values of the actions; otherwise, if you select a non-greedy action, then you are **exploring**, because this behavior allows you to improve your estimate of the non-greedy action's value.

• Exploitation is the prudent thing to do to maximize the expected reward on the one play – but exploitation may produce the greater total reward in the long run. This is the essence of the **exploitation-exploration** "**dilemma**."

n-Armed Bandits & Action-Value Methods

• Now we consider several elementary methods for estimating the values of actions and for using the estimates to make action selection decisions.

Denote the *true value* of action a as Q*(a), and the estimated value at the t*th* play as $Q_t(a)$. Recall that the true value of an action is the mean reward received when the action is selected.

(I) One natural method to estimate this quantity, which we call the **sample-average method**, is by simply averaging the rewards actually received:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a}$$

Where k_a denotes the number of times action *a* has been chosen prior to time *t*, yielding rewards $r_1, r_2, \ldots, r_{ka}$ (if $k_a=0$ then define $Q_t(a)=0$, etc.).

n-Armed Bandits & Action-Value Methods

(I) One natural method to estimate this quantity is by simply averaging the rewards actually received:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a}$$

Where k_a denotes the number of times action a has been chosen prior to time t, yielding rewards $r_1, r_2, \ldots, r_{ka}$ (if $k_a = 0$ then define $Q_t(a) = 0$, etc.).

• As $k_a \to \infty$, by the *law of large numbers*, $Q_t(a)$ converges to $Q^*(a)$.

• The simplest action selection rule is to select the action with highest estimated value. A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ε, instead select an action at random, uniformly, independently of the action-value estimates.

* This rule is known as the **ε-greedy method**.

n-Armed Bandits & Action-Value Methods

•To roughly assess the relative effectiveness of the greedy and ε-greedy methods, we can compare them numerically (see plots).

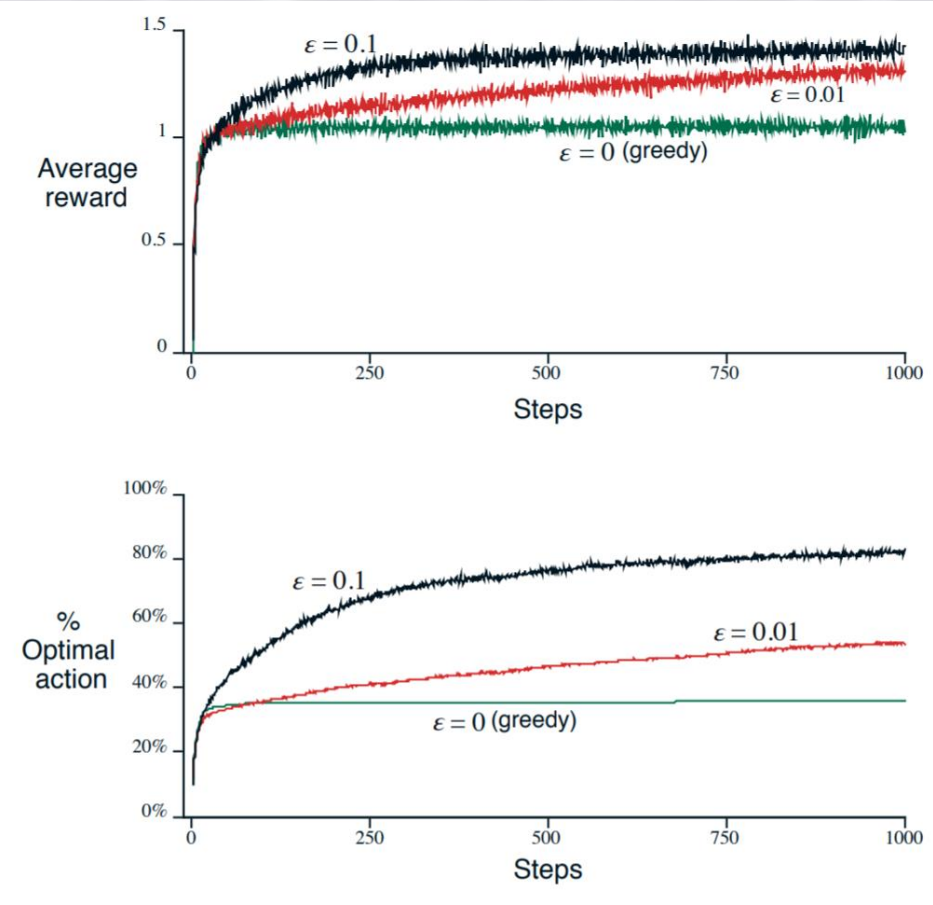


Figure 2.1: Average performance of ε-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 tasks. All methods used sample averages as their action-value estimates. The detailed structure at the beginning of these curves depends on how actions are selected when multiple actions have the same maximal action value. Here such ties were broken randomly. An alternative that has a similar effect is to add a very small amount of randomness to each of the initial action values, so that ties effectively never happen.

* Note that with noisy rewards it would take more exploration to find the optimal solution, and ε-greedy methods would fare generally even better than baseline greedy methods.

n-Armed Bandits & Action-Value Methods

•Although ε-greedy action selection is an effective and popular means of balancing exploration and exploitation in RL, one drawback is that when it explores it chooses equally among all actions. This means that it is as likely to choose the worst-appearing action as it is to choose the next-to-best action.

The obvious solution is to vary the action probabilities as a graded function of estimates value.

(II) Define the **softmax action selection**:

$$Q_t(a) \leftarrow \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$$

Where τ is a positive parameter call the **temperature**; high temperatures cause the actions to be all (nearly) equiprobable; low temperatures cause a greater difference in selection probability for actions that differ in their value estimates. In the limit $\tau \rightarrow 0$, softmax action selection becomes equivalent to greedy action selection.

n-Armed Bandits & Action-Value Methods

• The aforementioned action-value methods all estimate action values as sample averages of observed rewards.

A practical issue associated with estimating action values from samples of observed rewards is that these procedure will not scale well; larger samples will grow over time without bound.

As a remedy, we can devise an (III) **incremental update formula** for computing averages with small, constant computation required to process each new reward. For some action, let Q_k denote the average of its first k rewards; given this average and a (k+1)*st* reward, r_{k+1}, then the average of all k+1 rewards can be computed by:

$$Q_{k+1} = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i = \frac{1}{k+1}\left(r_{k+1} + \sum_{i=1}^{k} r_i\right) = \frac{1}{k+1}\left(r_{k+1} + kQ_k + Q_k - Q_k\right)$$

Why?

n-Armed Bandits & Action-Value Methods

• The aforementioned action-value methods all estimate action values as sample averages of observed rewards.

A practical issue associated with estimating action values from samples of observed rewards is that these procedure will not scale well; larger samples will grow over time without bound.

As a remedy, we can devise an (III) **incremental update formula** for computing averages with small, constant computation required to process each new reward. For some action, let Q_k denote the average of its first k rewards; given this average and a (k+1)*st* reward, r_{k+1}, then the average of all k+1 rewards can be computed by:

$$Q_{k+1} = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i = \frac{1}{k+1}\left(r_{k+1} + \sum_{i=1}^{k} r_i\right) = \frac{1}{k+1}\left(r_{k+1} + kQ_k + Q_k - Q_k\right)$$

$$= \frac{1}{k+1}\left(r_{k+1} + (k+1)Q_k - Q_k\right) = Q_k + \frac{1}{k+1}\left(r_{k+1} - Q_k\right)$$

* Note that this implementation requires memory only for Q_k and k.

n-Armed Bandits & Action-Value Methods

$$Q_{k+1} = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i = \frac{1}{k+1}\left(r_{k+1} + \sum_{i=1}^{k} r_i \right) = \frac{1}{k+1}\left(r_{k+1} + kQ_k + Q_k - Q_k \right)$$

$$= \frac{1}{k+1}\left(r_{k+1} + (k+1)Q_k - Q_k \right) = Q_k + \frac{1}{k+1}\left(r_{k+1} - Q_k \right)$$

- This update rule is of a familiar form:

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

n-Armed Bandits & Action-Value Methods

$$Q_{k+1} = Q_k + \frac{1}{k+1}\left(r_{k+1} - Q_k\right)$$

- This update rule is of a familiar form:

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

- The expression [Target-OldEstimate] is an *error* in the estimate; it is reduced by taking a step toward the "target."

- Averaging methods discussed previously are appropriate for **stationary environments** (i.e. environments that do not change over time). With non-stationary environments, it is common to add a constant *step-size parameter* $0 < \alpha \leq 1$ to the previous update rule, giving:

$$Q_{k+1} = Q_k + \alpha\left(r_{k+1} - Q_k\right) = \ldots = \left(1-\alpha\right)^k Q_0 + \sum_{i=1}^{k} \alpha\left(1-\alpha\right)^{k-i} r_i$$

- This is sometimes called an *exponential recency-weighted average*; the basic idea is that the weight given to reward r_i decreases as the number of intervening rewards increases.

n-Armed Bandits & Action-Value Methods

• The previous methods are all dependent on the initial action-value estimates $Q_0(a)$; *viz.*, these methods are *biased* by their initial estimates.

• For sample-average methods, the bias disappears once all actions have been selected at least once, but for methods with constant α, the bias is permanent, though decreasing over time.

• In practice, this kind of bias is usually not a problem, and can even be helpful. The downside is that these initial estimates become *de facto* hyperparameters.

n-Armed Bandits & Action-Value Methods

• Initial action values can be used as a simple way of encouraging exploration. If we, say, initially choose **wildly optimistic** action values (e.g. very large parameter settings), this will encourage the agent to explore, being "disappointed" with the rewards received.

• Using optimistic initial values is a simple and often effective trick; however, it is generally poorly-suited to non-stationary cases.

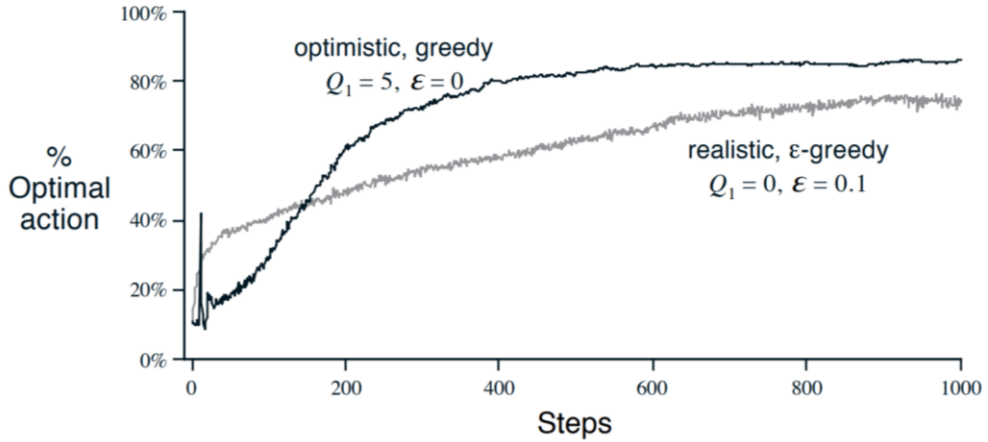


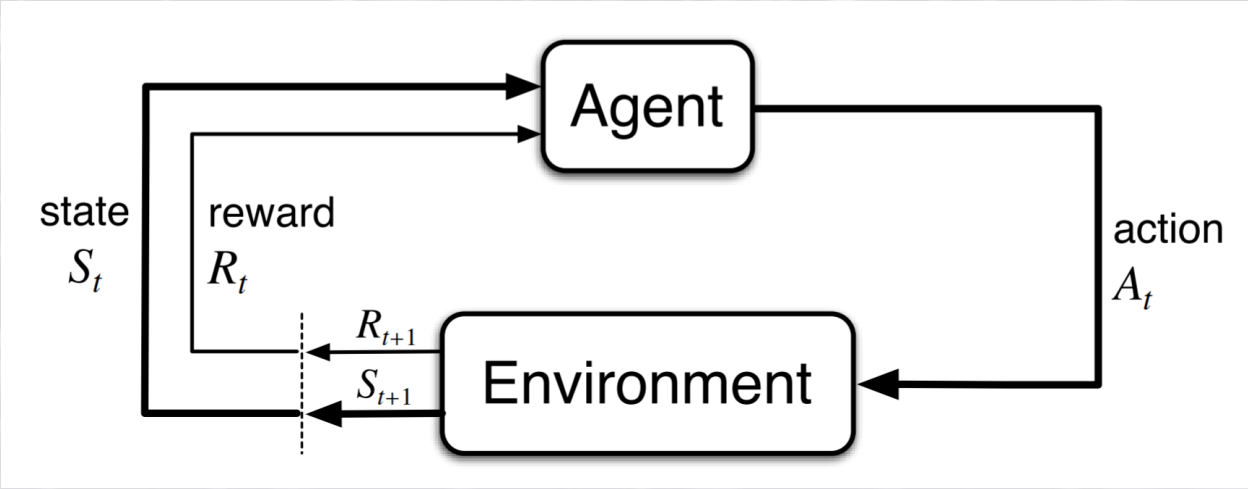
Figure 2.2: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$.

The Formal RL Problem

- In the general RL framework, an agent interactions with its environment at each of a sequence of discrete time steps, t = 0, 1, 2, 3, … .

- At each time step *t*, the agent receives some representation of the environment's **state**, $s_t \in S$, where *S* is the set of possible states, and on that basis selects an **action**, $a_t \in A(s_t)$, where *A* is the set of actions available in state s_t.

- One time step later, in part as a consequence of its action, the agent receives a numerical reward $r_{t+1} \in \mathbb{R}$ and finds itself in a new state, s_{t+1}.



The Formal RL Problem

- At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's **policy** and is denoted π_t, where $\pi_t(s,a)$ is the probability that $a_t=a$ if $s_t=s$.

- RL methods specify how the agent changes its policy as a result of its experience.

- The agent's goal, roughly speaking, is to maximize the total amount of reward it receives in the long run.

The Formal RL Problem

- At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy and is denoted π_t, where $\pi_t(s,a)$ is the probability that $a_t=a$ if $s_t=s$.

- RL methods specify how the agent changes its policy as a result of its experience.

- The agent's goal, roughly speaking, is to maximize the total amount of reward it receives in the long run.

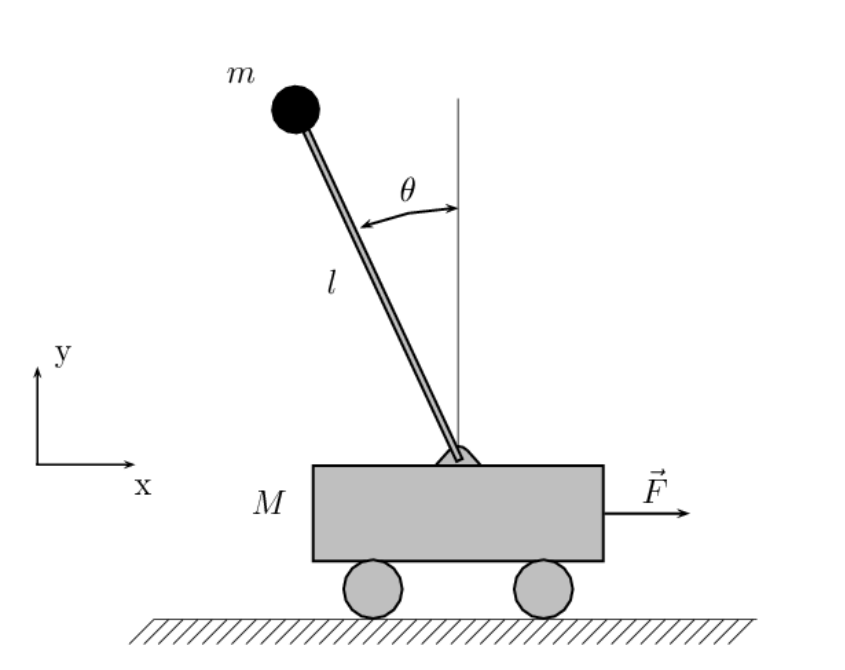
- The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of RL. Although this approach may appear superficially limiting, in practice it has proven to be a flexible and widely applicable method.

- For instance, if we want to make a robot learn to walk, we can provide a reward on each time step proportional to the robot's forward motion; in making a robot learn to escape from a maze, the reward is often zero until it escapes, at which time it receives +1 reward.

Applications of reinforcement learning: A few examples

- Learning to play backgammon (and more recently, Go)
- Robot arm control (juggling)
- Robot Locomotion
- Robot navigation
- Elevator dispatching
- Power systems stability control
- Job-shop scheduling
- Air traffic control
- Autonomous Driving

Cart-Pole Problem



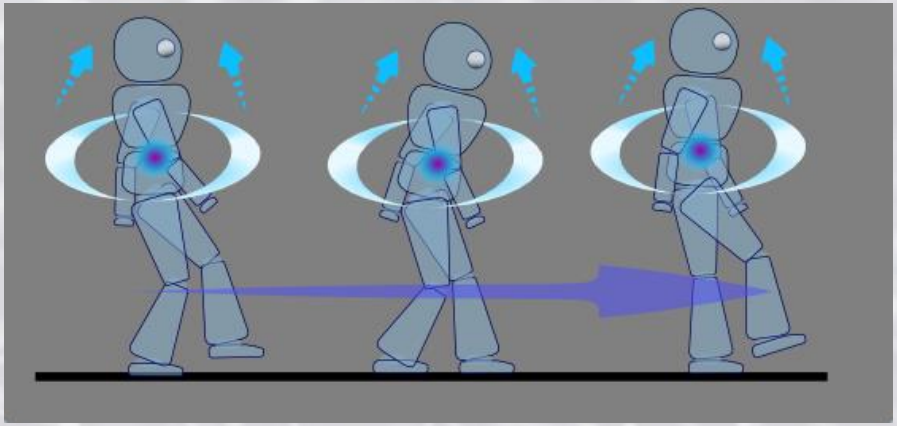
Objective: Balance a pole on top of movable cart.

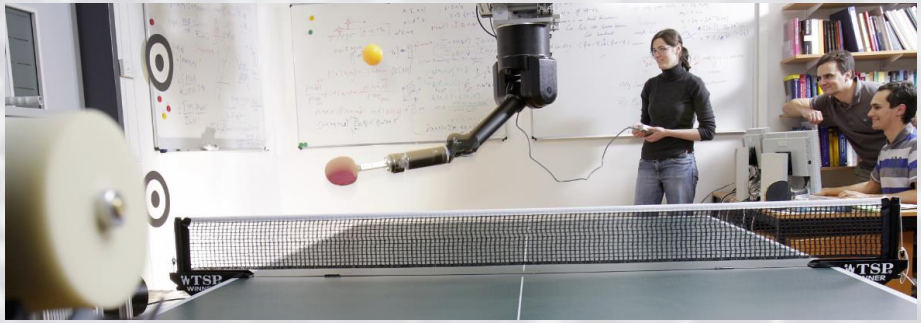
State: Angle, angular speed, position, horizontal velocity.

Action: Horizontal force applied to cart.
Reward: +1 at each time step if the pole is upright.

https://www.youtube.com/watch?v=_Mmc3i7jZ2c

Robot Locomotion (and pancake flipping!)





Objective: Make the robot move forward successfully.

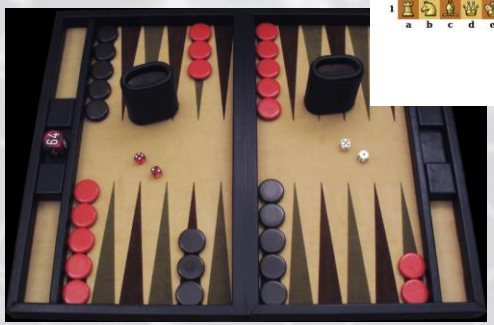
State: Angle and position of joints.
Action: Torques applied on joints.
Reward: +1 at each time step the robot is upright and moving forward.

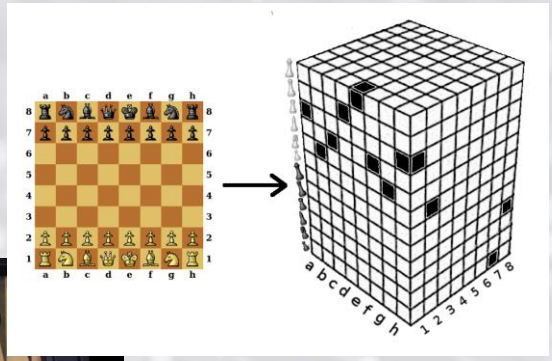
https://www.youtube.com/watch?v=gn4nRCC9TwQ

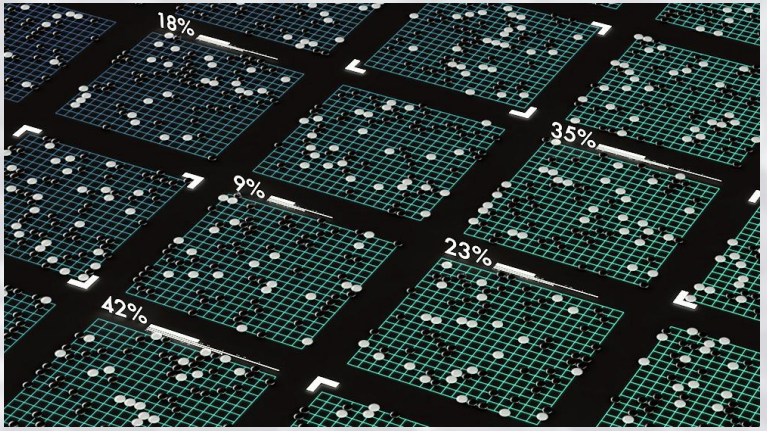
https://www.youtube.com/watch?v=W_gxLKSsSIE

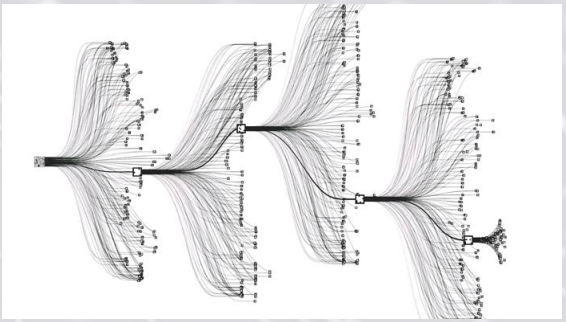
https://www.youtube.com/watch?v=SH3bADiB7uQ

Board Games (Backgammon, Chess, Go)









Objective: Win the game.

State: Position of pieces.
Action: Next move/placement of next piece.
Reward: +1 for win, 0 for loss.

Robby the Robot can learn via reinforcement learning

Sensors:

H(ere), N,S,E,W,

Actions:

Move N

Move S

Move E

Move W

Pick up can

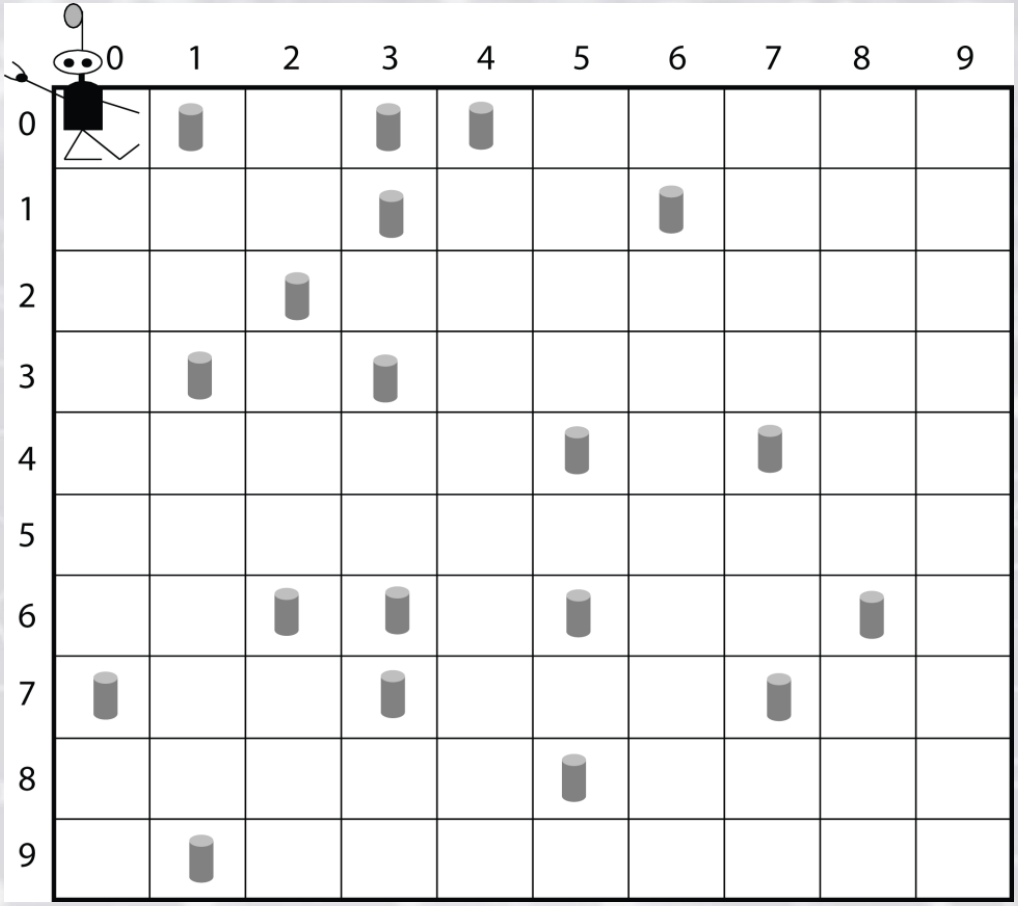
Rewards/Penalties (points):

Picks up can: 10

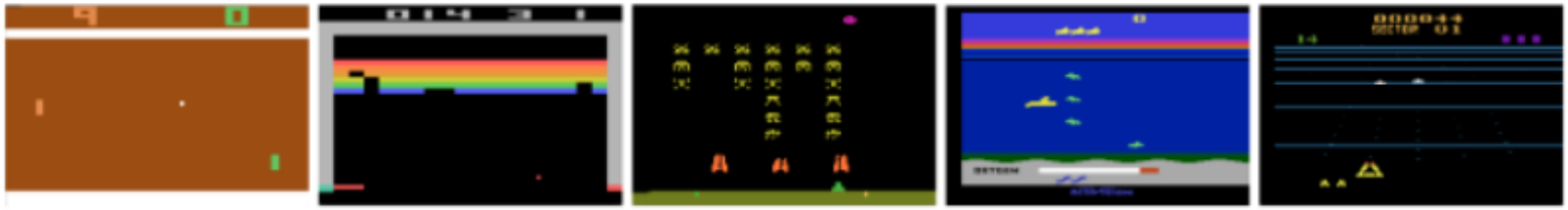
Pick up can on empty site: -1

Crashes into wall: -5

"policy" = "strategy"

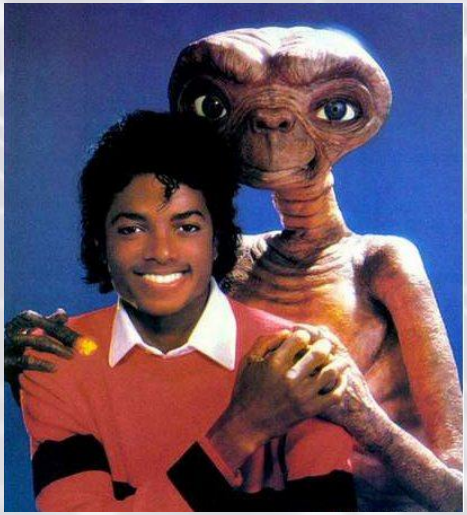


Atari Games



Objective: ~~World domination~~ Obtain high score (make lots of human friends in the process).

State: Raw pixel inputs.
Action: Game controls, e.g., movement and zap!
Reward: Score differential.



The Formal RL Problem: Goals & Rewards

- Part of the "art" of developing an effective RL algorithm rests in the choice of reward function. We want the reward to truly indicate what we want accomplished.

- In particular, the reward signal is **not** the place to impart to the agent prior knowledge about *how* to achieve what we want it to do; for example, a chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponent's pieces or gaining control of the center of the board.

- Explicitly rewarding subgoals may cause the agent to learn to achieve these subgoals to the detriment of any long-term objectives (e.g. the agent might learn to capture pieces effectively and yet still lose the chess match).

The Formal RL Problem: Returns

- The precise aspect of the sequence of rewards that we wish to maximize is the **expected return**, defined as:

$$R_t = r_{t+1} + r_{t+2} + \ldots r_T$$

where T is the final time step; this quantity is just the sum of rewards after step *t*.

This definition makes perfect sense when there is a natural notion of a "final" time step; such an agent-environment interaction consists of *episodes*.

- When the agent-environment interaction does not break naturally into identifiable episodes, and instead goes on continually without limit, we call these *continuing tasks* (i.e. $T = \infty$).

The Formal RL Problem: Returns

- An additional, common feature used in RL is **discounting**. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized.

- In particular, the agent chooses action a_t to maximize the *expected discounted return*:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $0 \leq \gamma \leq 1$, is called the *discount rate*. If $\gamma = 0$, we say the agent is "myopic."

Markov Decision Processes

- An RL learning task that satisfies the *Markov property* is called a **Markov Decision Process** (MDP).

The **Markov property** implies that the environment's response at time $t+1$ depends only on the state and action representations at time t. For example, a checkers position (i.e., the current piece configuration) would serve as a Markov state because the current state summarizes everything important about the complete sequence of positions that led to it.

More formally, if the Markov property holds, then:

$$P\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}$$

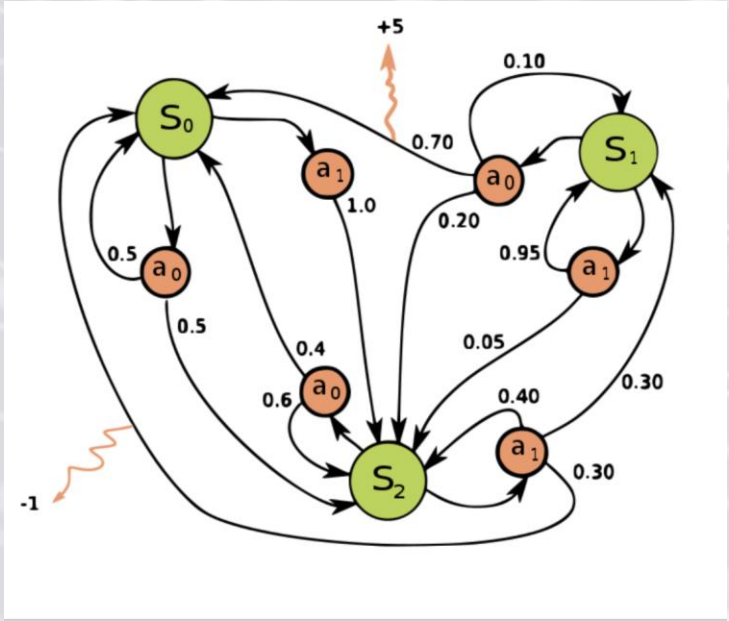
- Which is to say that the next state (and reward) only depend on the current state-action pair.

Markov Decision Processes

A Markov decision process is a 5-tuple (S, A, P_a, R_a, γ), where

- S is a finite set of states,
- A is a finite set of actions (alternatively, A_s is the finite set of actions available from state s),
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a,
- $\gamma \in [0, 1]$ is the discount factor, which represents the difference in importance between future rewards and present rewards.

Where $P_a(s,s')$ are called *transition probabilities*; note that the quantities: $P_a(s,s')$ and $R_a(s,s')$ completely specify the most important aspects of the dynamics of a MDP.



Value Functions

• Almost all RL learning algorithms are based on estimating value functions – functions of states (or of state-action pairs) that estimate "how good" it is for the agent to be in a given state (or how good it is to perform a given action in a given state).

• The notion of "how good" here is defined in terms of future rewards that can be expected, i.e., *expected return*.

• Recall that a policy, π, is a mapping from each state $s \in S$ and action, $a \in A(s)$ the probability π (s,a) of taking action *a* when in state *s*.

Value Functions

• Almost all RL learning algorithms are based on estimating value functions – functions of states (or of state-action pairs) that estimate "how good" it is for the agent to be in a given state (or how good it is to perform a given action in a given state).

• The notion of "how good" here is defined in terms of future rewards that can be expected, i.e., *expected return*.

• Recall that a policy, π, is a mapping from each state $s \in S$ and action, $a \in A(s)$ the probability π (s,a) of taking action *a* when in state *s*.

For MDPs, we can define $V^{\pi}(s)$, the **value of state *s* under policy π** as:

$$V^{\pi}(s) = E_{\pi}\left[R_t \mid s_t = s\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

where $E_{\pi}[\cdot]$ denotes the expected value *given that the agent follows policy π*. The function V^{π} is called the **state-value function for policy π**.

Value Functions

• Similarly, we define the value of taking action *a* in state *s* under policy π, denoted $Q^{\pi}(s,a)$ as the expected return stating from *s*, taking action *a*, and thereafter following policy π:

$$Q^{\pi}(s,a) = E_{\pi}\left[R_t \mid s_t = s, a_t = a\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right]$$

We call Q^{π} the *action-value function for policy π*.

Value Functions

• Similarly, we define the value of taking action *a* in state *s* under policy π, denoted $Q^{\pi}(s,a)$ as the expected return stating from *s*, taking action *a*, and thereafter following policy π:

$$Q^{\pi}(s,a) = E_{\pi}\left[R_t \mid s_t = s, a_t = a\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right]$$

We call Q^{π} the *action-value function for policy π.*

• The value functions V^{π} and Q^{π} can be estimated from experience. For example, if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state's value V(s), as the number of times that state is encountered approaches infinity.

• If separate averages are kept for each action taken in a state, then these averages will similarly converge to the action values, $Q^{\pi}(s,a)$.

• We call estimate methods of this kind **Monte Carlo methods** because they involve averaging over random samples of actual returns.

Value Functions

- A fundamental property of value functions used in RL and dynamic programming is that they satisfy recursive relationships.

- For any policy π and any state *s*, the following consistency condition holds between the value of *s* and the value of its possible successor states:

$$V^{\pi}(s) = E_{\pi}\left[R_t \mid s_t = s\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

Value Functions

- A fundamental property of value functions used in RL and dynamic programming is that they satisfy recursive relationships.

- For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$V^{\pi}(s) = E_{\pi}\left[R_t \mid s_t = s\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

$$= E_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\right]$$

Why?

Value Functions

- A fundamental property of value functions used in RL and dynamic programming is that they satisfy recursive relationships.

- For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$V^{\pi}(s) = E_{\pi}\left[R_t \mid s_t = s\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

$$= E_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\right]$$

$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_{t+1} = s'\right]\right]$$

Why?

Value Functions

- A fundamental property of value functions used in RL and dynamic programming is that they satisfy recursive relationships.

- For any policy π and any state *s*, the following consistency condition holds between the value of *s* and the value of its possible successor states:

$$V^{\pi}(s) = E_{\pi}\left[R_t \mid s_t = s\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

$$= E_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\right]$$

$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_{t+1} = s'\right]\right]$$

$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}(s')\right]$$

Why?

Value Functions

- A fundamental property of value functions used in RL and dynamic programming is that they satisfy recursive relationships.

- For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$V^{\pi}(s) = E_{\pi}\left[R_t \mid s_t = s\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

$$= E_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\right]$$

$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_{t+1} = s'\right]\right]$$

$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}(s')\right]$$

("Bellman Equation" for V^{π})

Where it is implicit that the actions, a, are taken from the set $A(s)$, and the next states, s', are taken from the set S.

Bellman Equation

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

- The **Bellman Equation** for V^{π} expresses a relationship between the value of a state and the values of its successor states.

- Think of "looking ahead" from one state to its possible success states (see "backup diagram").

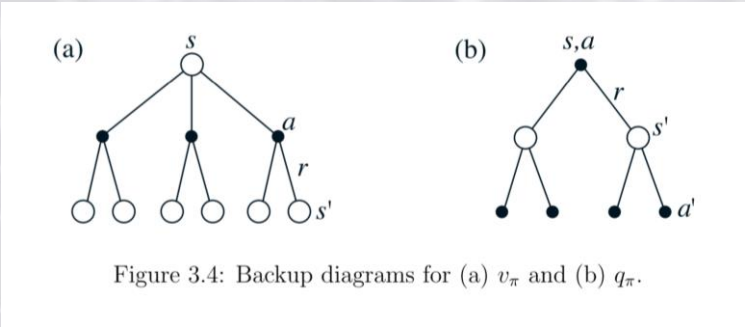


Figure 3.4: Backup diagrams for (a) v_π and (b) q_π.

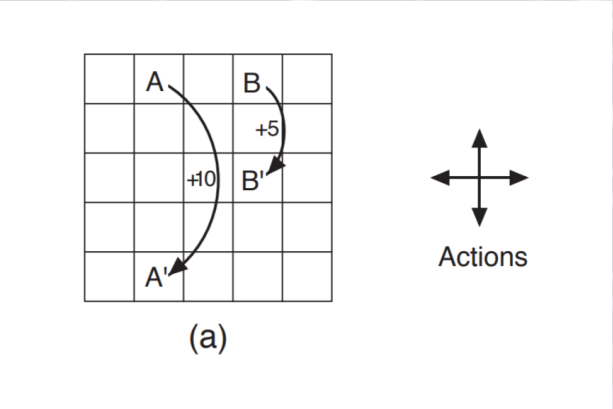
- Each open circle represents a state and each solid circle represents a state-action pair. Starting from state *s*, the root node at the top, the agent could take any of some set of actions – three are shown. From each of these, the environment could respond with one of several next states, *s'*, along with a reward, *r*.

- The Bellman equation averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.

Value Functions: Gridwold Example

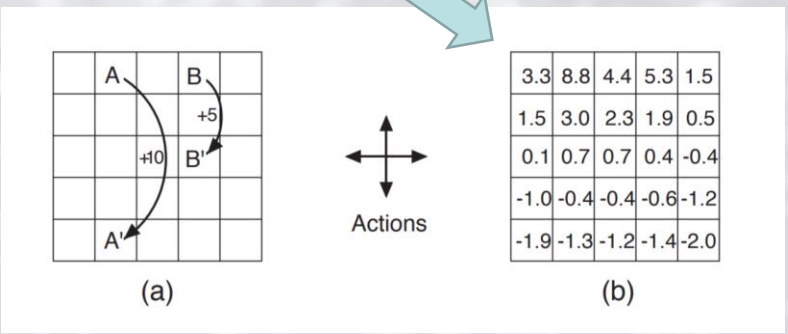
- Consider the MDP: the cells of the grid correspond to the states of the environment; at each cell four actions are possible: north, south, east, and west, which deterministically cause the agent to move one cell in the corresponding direction on the grid.

- Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From state A, all four actions yield a reward of +10 and take the agent to A'. From state B, all actions yield a reward of +5 and take the agent to B'.



Value Functions: Gridwold Example

- Suppose the agent selects all four actions with equal probability in all states. The corresponding value function, V^{π}, for this policy, for the discounted reward case with $\gamma = 0.9$ is shown.



- V^{π} is computed using the Bellman equation:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

Optimal Value Functions

- Solving a RL learning task means, roughly, finding a policy that achieves a lot of reward over the long run.

- For finite MDPs, we can precisely define an optimal policy by relying on the fact that value functions define a **partial ordering** over policies.

- A policy π is defined to be **better than or equal** to policy π' if its expected return is greater than or equal to that of π' for all stages.

Optimal Value Functions

- Solving a RL learning task means, roughly, finding a policy that achieves a lot of reward over the long run.
- For finite MDPs, we can precisely define an optimal policy by relying on the fact that value functions define a **partial ordering** over policies.
- A policy π is defined to be **better than or equal** to policy π' if its expected return is greater than or equal to that of π' for all stages.
- In other words, $\pi \geq \pi'$ *if and only if* $V^{\pi}(s) \geq V^{\pi'}(s)$ for all $s \in S$.
- There is always exists at least one policy that is better than or equal to all other policies; this is an **optimal policy**.

Optimal Value Functions

- Denote the optimal policy by π*; the optimal state-value function, denoted V* is defined:

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

for all $s \in S$.

- Optimal policies also share the same *optimal action-value function*, denoted Q*, defined:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

for all $s \in S$ and a $\in$ A(s).

- For the state-action pair (s,a), this function gives the expected return for taking action *a* in state *s* and thereafter following an optimal policy. Thus, we can write Q* in terms of V* as follows:

$$Q^*(s,a) = E\left[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right]$$

Optimal Value Functions

- Because V* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

- In addition, because it is the optimal value function, V*'s consistency condition can be written in a special form without reference to any specific policy.

- This is known as the **Bellman optimality equation**. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned}
V^{*}(s) &= \max_{a \in A(s)} Q^{\pi^{*}}(s,a) \\
&= \max_{a} E_{\pi^{*}} \left[R_t \mid s_t = s, a_t = a \right] \\
&= \max_{a} E_{\pi^{*}} \left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^{k} r_{t+k+2} \mid s_t = s, a_t = a \right] \\
&= \max_{a} E \left[r_{t+1} + \gamma V^{*}(s_{t+1}) \mid s_t = s, a_t = a \right] \\
&= \max_{a} \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + V^{*}(s') \right]
\end{aligned}$$

Optimal Value Functions

$$V^*(s) = \max_a E\left[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right]$$
$$= \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^*(s')\right]$$

- The **Bellman optimality equation** for Q* is given by:

$$Q^*(s,a) = E\left[r_{t+1} + \gamma Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\right]$$
$$= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right]$$

- The backup diagrams show graphically the spans of future states and actions considered in the Bellman optimality equations for V* and Q*.

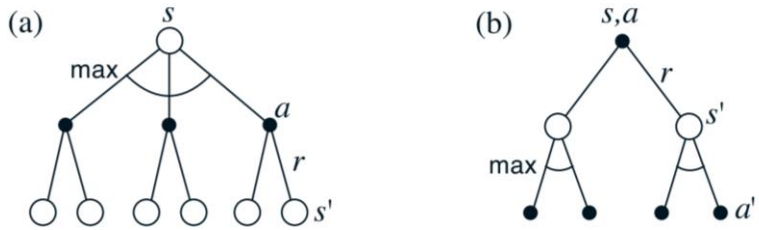


Figure 3.7: Backup diagrams for (a) v_* and (b) q_*

Optimal Value Functions

$$Q^*(s,a) = E\left[r_{t+1} + \gamma Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\right]$$
$$= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right]$$

- For finite MDPs, the Bellman optimality equation for V* has a unique solution independent of the policy. The Bellman optimality "equation" is actually a system of equations, one for each state (thus for N states one has N equations and N unknowns).

- If the dynamics of the environment are known (i.e. $R^a_{ss'}$ and $P^a_{ss'}$ known), then in principle one can solve this system of equations for V*; one can, in addition, solve a related set of equations for Q*.

Optimal Value Functions

- Once one has V*, it is relatively easy to determine an optimal policy. For each state *s*, there will be one or more actions at which the maximum is attained in the Bellman optimality equation.

- If you have the optimal value function V*, then the actions that appear best after a one-step search will be optimal actions. Put another way, any policy that is greedy with respect to the optimal value function V* is an optimal policy.

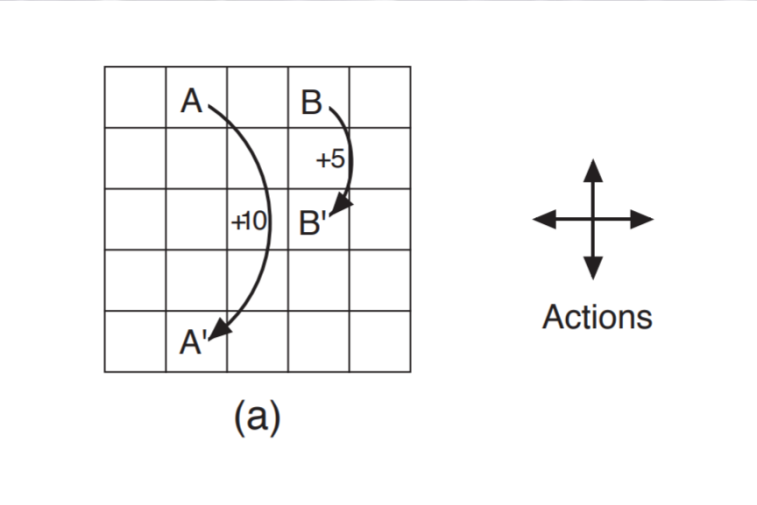
- The beauty of V* is that if one uses it to evaluate the short-term consequences of actions – specifically, the one-step consequences – then a greedy policy is actually optimal in the long-term sense because V* already takes into account the reward consequences of all future behavior.

- Having Q* makes choosing optimal actions still easier. With Q*, the agent does not even have to do a one-step-ahead search: for any state s, it can simply find any action that maximizes Q*(s,a).

Value Functions: Gridwold Example

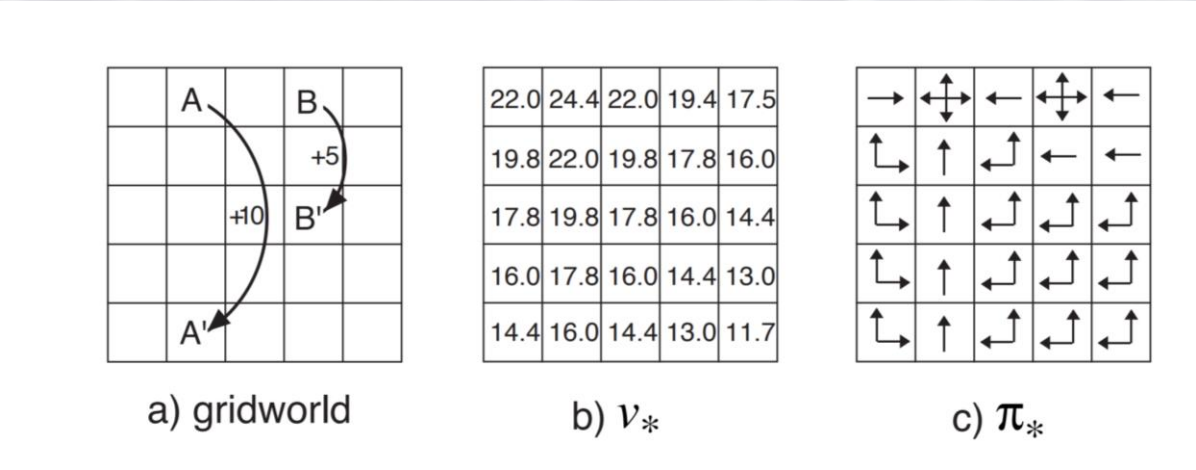
Returning to the Gridworld example from before:

- Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From state A, all four actions yield a reward of +10 and take the agent to A'. From state B, all actions yield a reward of +5 and take the agent to B'.



Value Functions: Gridwold Example

- The optimal value solutions are given as follows:



where V^* is computed using the Bellman optimality equation:

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + V^*(s') \right]$$

Value Functions: Practical Concerns

Recall the Bellman optimality equation:

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + V^*(s') \right]$$

• Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the RL problem.

• However, this solution is rarely used in practice, as it is akin to an exhaustive search: looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards.

This solution relies on at least three assumptions that are rarely true in practice:

(1) We actually know the dynamics of the environment
(2) We have enough computational resources to complete the computation of the solution
(3) Markov Property

Value Functions: Practical Concerns

• Naturally, an agent that learns an optimal policy has done very well, but in practice this rarely happens.

• For "interesting", real-world problems, optimal policies can be generated only with extreme computational cost. However, a well-defined notion of optimality nevertheless helps frame RL in a mathematically rigorous way.

In practice, optimal policies represent an ideal that agents can only approximate to varying degrees.

Value Functions: Practical Concerns

• Oftentimes, it is also impossible to directly use *tabular methods* to build up approximations of value functions and policies, because there are far more states than could possibly be entries in a table. In these cases the functions must be approximated, using some sort of more compact parameterized function representation (e.g. a DNN).

• Many useful techniques exist for dealing with very large search spaces, including heuristic search methods. In approximating optimal behavior, there may be many states the agent faces with very low probability; the on-line nature of RL makes it possible to approximate optimal policies in a way that puts more effort into learning to make good decisions for frequently encountered states.

• With **TD-Gammon** (1992, IBM Watson Research Center), a classic RL-based Backgammon AI program, for instance, although the program performed at near human expert level, it nonetheless makes bad decisions on board configurations that rarely (or never) appear in games. Backgammon has on the order of $\sim 10^{20}$ states.

Dynamic Programming

• **Dynamic Programming** (DP) techniques can be used to compute optimal policies given a perfect model of the environment as a MDP; in practice DP techniques can be computationally expensive for RL, but they nevertheless provide an essential foundation across RL frameworks.

• The key idea of DP in conjunction with RL is the use of value functions to organize and structure the search for good policies.

• One can easily obtain optimal policies once we have found the optimal value functions, V* or Q*, which satisfy the *Bellman equations* (from before):

$$V^*(s) = \max_a E\left[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right]$$
$$= \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^*(s')\right]$$

$$Q^*(s,a) = E\left[r_{t+1} + \gamma Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\right]$$
$$= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right]$$

(*) **Key idea**: Turn the Bellman equations into iterative assignment updates for approximating the desired value functions.

Dynamic Programming: Policy Evaluation

• First we consider how to compute the state-value function V^π for any arbitrary policy π; this is called policy evaluation.

Recall that for all $s \in S$:

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right]$$

- If the **environment's dynamics are completely known** (*viz.*, we have a complete *model*), then the equation above is a system of |S| unknowns; we consider an iterative solution.

- Consider a sequence of approximate value functions: V_0, V_1, V_2, where V_0 is initialized arbitrarily.

- Each successive approximation for V^π can be updated as follows:

$$\begin{aligned} V_{k+1}(s) &= E_\pi \left[r_{t+1} + \gamma V_k(s+1) \mid s_t = s \right] \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right] \end{aligned}$$

Dynamic Programming: Policy Evaluation

$$V_{k+1}(s) = E_\pi\left[r_{t+1} + \gamma V_k(s+1) \mid s_t = s\right]$$
$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V_k(s')\right]$$

- This is known as *iterative policy evaluation*.

- NB: It can be shown that $\{V_k\}$ converges to V^π as $k \to \infty$ (as $V_k = V^\pi$ is a fixed point for the Bellman equation).

- For implementation, iterative policy evaluation uses a "full backup", meaning that in order to approximate V_{k+1} from V_k, we replace the old value of s with a new value obtained from the old values of the successor states of s.

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Figure 4.1: Iterative policy evaluation.

Dynamic Programming: Policy Improvement

• The reason for computing value functions for a policy is to assist in the search for better policies; naturally, given a policy, we would like to determine whether we should change its action for a particular state in order to improve the policy.

To this end, define:

$$Q^{\pi}(s,a) = E_{\pi}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s, a_t = a\right]$$
$$= \sum_{s'} P^{a}_{ss'}\left[R^{a}_{ss'} + \gamma V^{\pi}(s')\right]$$

This quantity considers selecting *a* in state *s* and thereafter following the existing policy, π.

*The key criterion is whether this is greater than or less than $V^{\pi}(s)$. If it is greater, then one would expect it to be better still to select *a* every time *s* is encountered.

This is in general true, as stated by the *policy improvement theorem*.

Dynamic Programming: Policy Improvement

- *Policy improvement theorem:*

Let π and π' be any pair of deterministic policies such that, for all $s \in S$:

$$Q^{\pi}\left(s, \pi'(s)\right) \geq V^{\pi}(s)$$

Then policy π' must be as good as, or better than, π. Thus for all $s \in S$, it follows that:

$$V^{\pi'}(s) \geq V^{\pi}(s)$$

Dynamic Programming: Policy Improvement

• In summary, given a policy and its value function, we can easily evaluate a change in the policy at a single state to a particular action.

• As an extension, we can consider changes at all states and to all possible actions, selecting at each state the action that appears best according to $Q^{\pi}(s,a)$.

In other words, to consider the new greedy policy, π', given by:

$$\pi'(s) = \arg\max_a Q^{\pi}(s,a) = \arg\max_a E_{\pi}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s, a_t = a\right]$$
$$= \arg\max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}(s')\right]$$

The greedy policy takes the action that looks best in the short term – after one step of lookahead – according to V^{π}.

* By construction, the greedy policy meets the conditions of the policy improvement theorem; these results are naturally extended to the case of stochastic policies.

Policy Improvement: GridWorld

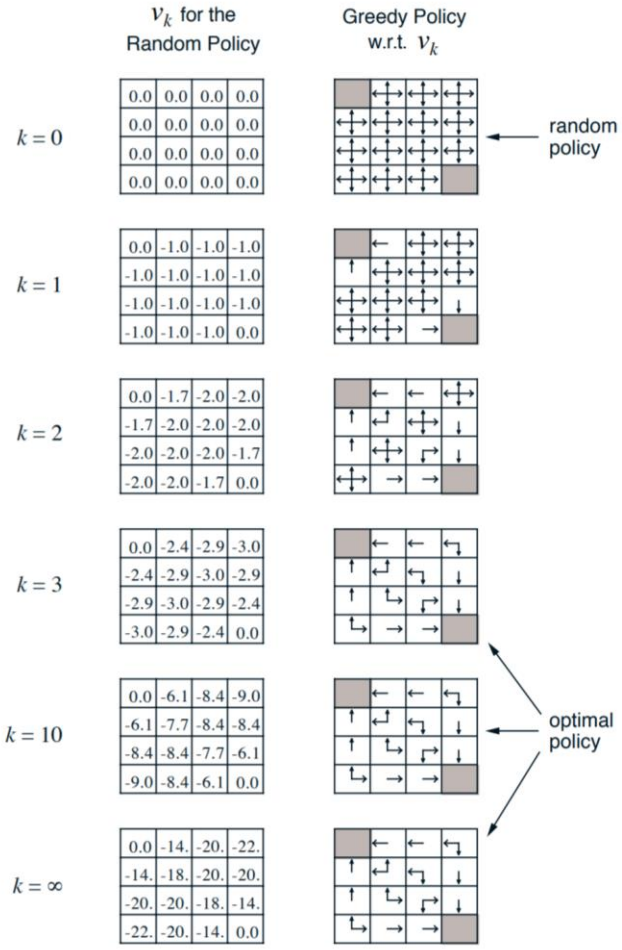


Figure 4.2: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equal). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum). The last policy is guaranteed only to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal.

Dynamic Programming: Policy Iteration

• Once a policy, π, has been improved using V^{π} to yield a better policy, π', we can then compute $V^{\pi'}$ and improve it again to yield an ever better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

Where E denotes a policy *evaluation* and I denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one.

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Dynamic Programming: Value Iteration

• The policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration.

• One importance special case is when policy evaluation is stopped after just one sweep (i.e. one backup of each state).

• This particular algorithm is called **value iteration**; it can be written as a particularly simple backup operation that combines the policy improvement and truncated policy evaluation steps:

$$V_{k+1}(s) = E_\pi\left[r_{t+1} + \gamma V_k(s+1) \mid s_t = s, a_t = a\right] = \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V_k(s')\right]$$

* This is equivalent to turning the Bellman optimality equation into an update rule:

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^*(s')\right]$$

Dynamic Programming: Value Iteration

$$V_{k+1}(s) = E_\pi\left[r_{t+1} + \gamma V_k(s+1) \mid s_t = s, a_t = a\right] = \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V_k(s')\right]$$

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

Figure 4.5: Value iteration.

Dynamic Programming: Practical Considerations

- A significant drawback to the DP methods discussed, is that they involve operations over the entire state set of the MDP, i.e., sweeps of the state set.

- If the state set is very large, then even a single sweep can be prohibitively expensive (e.g. backgammon has over 10^{20} states).

- **Asynchronous DP algorithms** are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms back up the values of states in any order whatsoever.

Dynamic Programming: General Policy Iteration

• Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (*policy evaluation*), and the other making the policy greedy with respect to the current value function (*policy improvement*).

• In policy iteration, these two processes alternate, each completing before the other begins, but this it not really necessary. In value iteration, for example, only a single iteration of policy evaluation is performed in between each policy improvement.

*<u>Almost all RL methods</u> can be described as *generalized policy iteration procedures (GPI)*.

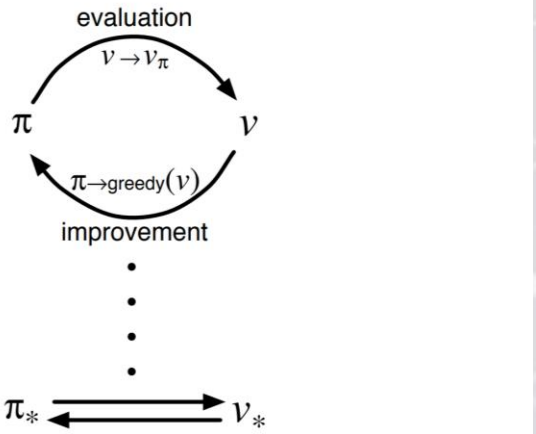
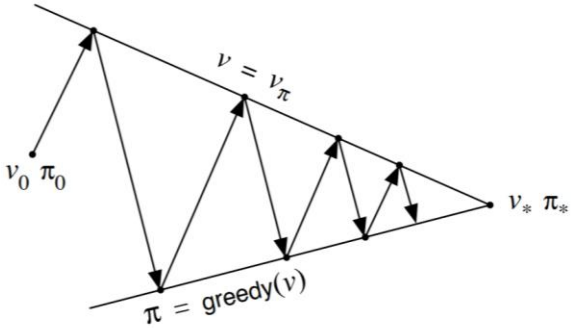


Figure 4.7: Generalized policy iteration: Value and policy functions interact until they are optimal and thus consistent with each other.



One can think of the interaction between the evaluation and improvement processes in GPI in terms of constraints. Each process drives the value function or policy toward one another; the goals accordingly interact.

Dynamic Programming: Efficiency

• DP may not be practical for large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient (remember that DP also requires an environment model).

• In the worst-case, DP methods find an optimal policy in polynomial time (wrt the number of states and actions).

• Linear programming methods can also be used to solve RL problems, but these methods become impractical at a much smaller number of states than DP methods.

Monte Carlo Methods

• Unlike Dynamic Programming methods, Monte Carlo methods (MCM) do not assume complete knowledge of the environment.

• MCM require only *experience* – sample sequence of states, actiona, and rewards from on-line or simulated interaction with an environment.

Learning from on-line experience is striking because it requires no prior knowledge of the environment's dynamics, yet can **still attain optimal behavior**.

- MCM are ways of solving the RL problem based on averaging sample returns.
- Despite their differences, the most important ideas from DP carry over to the MCM case. In particular, MCM attain optimality in essentially the same was as DP methods.

Monte Carlo Methods

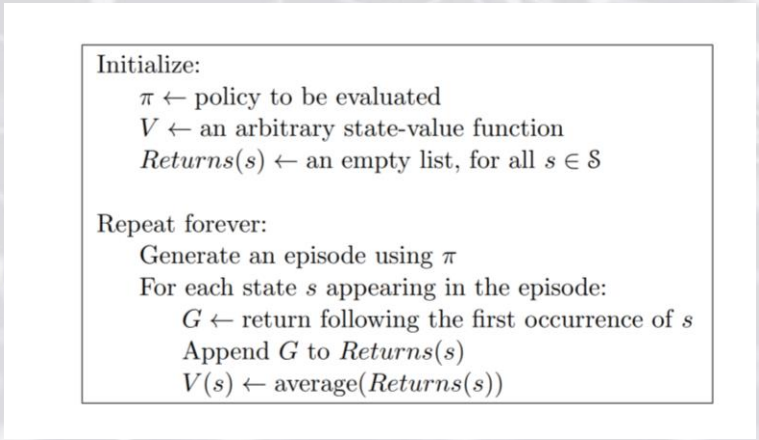
• Let's consider MCM for learning the state-value function for a given policy.

• Recall that the **value of a state** is the expected return – expected cumulative future discounted reward – starting from that state.

• An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value; this is the core idea underyling all MCM.

• One such method is called the **first-visit MCM**; this process just averages the returns following the first visits to *s*.

• By the *law of large numbers*, the first-visit MCM converges to $V^{\pi}(s)$ as the number of first visits to *s* goes to infinity.



Monte Carlo Methods: Blackjack

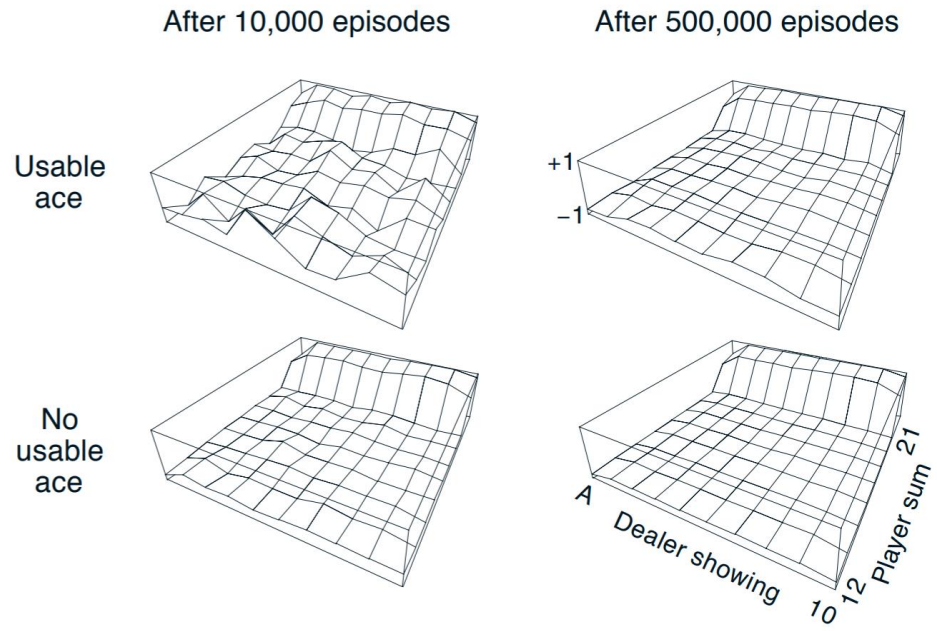


Figure 5.1: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation. ■

• Blackjack as an MDP: rewards of +1, -1, 0 are given for winning, losing and drawing respectively; no discount applied; cards drawn with replacement; policy considered: stick of player's sum is 20 or 21; state-value function approximated using MCM (DP would be difficult to apply here, since we require transition probabilities and associated rewards for all states).

Monte Carlo Methods

• If a model is not available, then it is particularly useful to estimate action values rather than state values. With a model, state values alone are sufficient to determine a policy; simply look ahead one step and choose whoever action leads to the best combination of reward and next state.

• **Without a model**, however, state values are insufficient. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Thus we should estimate Q*.

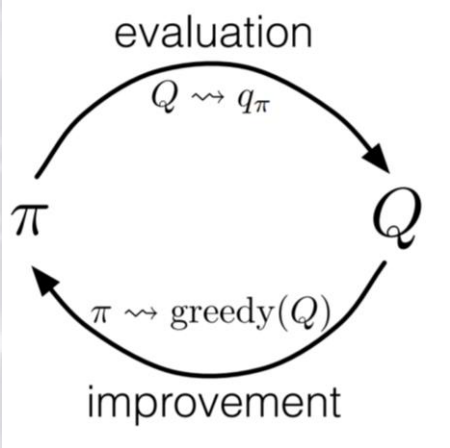
• The first-visit MC method averages the returns following the first time in each episode that the state was visited and the action was selected; these methods converge quadratically to the true expected values as the number of visits to each state-action pair approaches infinity.

* The only complication here is that many relevant state-action pairs may never be visited; one common remedy is to consider only policies that are stochastic with a nonzero probability of selecting all actions.

Monte Carlo Control

• How is MCM used to approximate optimal policies?

• The general pattern is to proceed as we did with regard to DP; we maintain both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function:



• MCM version of classical policy iteration entails performing alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal action-value function:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

Monte Carlo Control

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

• Policy evaluation is done exactly as described previously; many episodes are experienced, with the approximate action-value function approaching the true function asymptotically.

- Under some basic assumptions (e.g., infinite number of episodes), the MCM will compute $Q^{\pi k}$ exactly, for arbitrary π_k.

- Policy improvement is achieved by making the policy greedy with respect to the current value function. In this case, we have an action-value function, and therefore no model is needed to construct the greedy policy.

- For any action-value function Q, the corresponding **greedy policy** is the one that deterministically chooses:

$$\pi(s) = \arg\max_a Q(s,a)$$

- Policy improvement then can be done by constructing each π_{k+1} as the greedy policy wrt $Q^{\pi k}$.

Monte Carlo Control

• Define **Monte Carlo ES** as the MC algorithm that alternates between evaluation and improvement on an episode-by-episode basis:

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$\pi(s) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list

Repeat forever:
Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
Generate an episode starting from S_0, A_0, following π
For each pair s,a appearing in the episode:
$G \leftarrow$ return following the first occurrence of s,a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
For each s in the episode:
$\pi(s) \leftarrow \arg\max_a Q(s,a)$

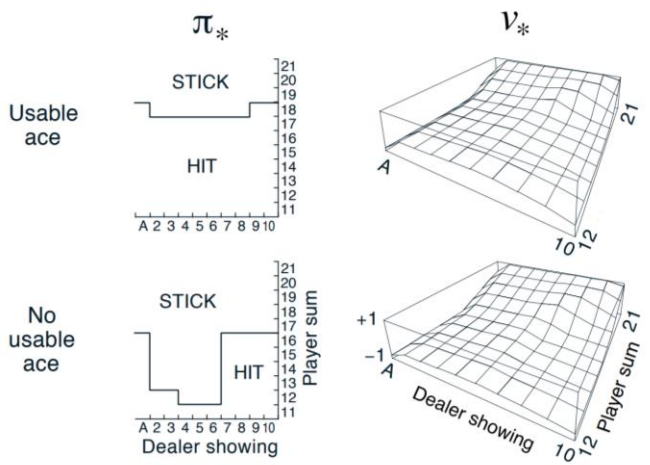


Figure 5.5: The optimal policy and state-value function for blackjack, found by Monte Carlo ES (Figure 5.4). The state-value function shown was computed from the action-value function found by Monte Carlo ES.

Monte Carlo Methods : On-Policy

• There are two general approaches to ensure that all actions are selected infinitely often: *on-policy* and *off-policy* methods.

• **On-policy methods** attempt to evaluate or improve the policy that is used to make decisions.

In on-policy control methods, the policy is generally soft, meaning that $\pi(s,a)>0$ for all $s \in S$ and all $a \in A(s)$.

• One common on-policy method uses the *epsilon-greedy approach*, meaning that most of the time they choose and action that has maximal estimated action value, but with probability epsilon they instead select an action at random.

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s,a appearing in the episode:
$G \leftarrow$ return following the first occurrence of s,a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
(c) For each s in the episode:
$a^* \leftarrow \arg\max_a Q(s,a)$
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

Figure 5.6: An on-policy first-visit MC control algorithm for ε-soft policies.

Monte Carlo Methods : Off-Policy

- On-policy methods estimate the value of a policy while using it for control.

- In **off-policy methods** these two functions are separated. The policy used to generate behavior, called the *behavior policy*, may in fact be unrelated to the policy that is evaluated and improved, called the *estimation policy*.

An advantage of this separation is that the estimation policy may be deterministic (e.g. greedy), while the behavior policy can continue to sample all possible actions.

- Off-policy MC control methods use the technique previously presented from estimating the value function for one policy while following another. They follow the behavior policy while learning about and improving the estimation policy (to explore all possibilities, we require that the behavior policy be soft).

Temporal-Difference Learning

• **Temporal-Difference** (TD) learning is a combination of Monte Carlo ideas and dynamic programming ideas.

• Like MC methods, TD methods can learn directly from raw experience without a model of the environment's dynamics.

• Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

* The relationship between TD, DP and MC methods is a recurring theme in RL.

Temporal-Difference Learning

• Both TD and MC methods use experience to solve the prediction problem.

• Given some experience following a policy π, both methods update their estimate V of V^{π}. If a nonterminal state s_t is visited at time *t*, then both methods update their estimate $V(s_t)$ based on what happens after that visit. Roughly speaking, MC methods wait until the return following the visit is known, then use that return as a target for $V(s_t)$.

A simple, every-visit MC method suitable for nonstationary environments is:

$$V(s_t) \leftarrow V(s_t) + \alpha\left[R_t - V(s_t)\right]$$

Where R_t is the actual return following time *t* and α is a constant step-size parameter. Call this method *constant-a MC*.

Temporal-Difference Learning

• Whereas MC methods must wait until the end of the episode to determine the increment to $V(s_t)$ (only R_t is known), TD methods need wait only until the next time step.

• At time t+1 they immediately form a target and make a useful update using the observed reward r_{t+1} and the estimate $V(s_{t+1})$. The simplest TD method, known as **TD(0)** is:

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

• In effect, the target for the MC update is R_t, whereas the target for the TD update is $r_{t+1}+\gamma V_t(s_{t+1})$. Because TD method bases its update in part on an existing estimate, we say that is a *bootstrapping method* (like DP).

Temporal-Difference Learning

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

• In effect, the target for the MC update is R_t, whereas the target for the TD update is $r_{t+1}+\gamma V_t(s_{t+1})$. Because TD method bases its update in part on an existing estimate, we say that is a *bootstrapping method* (like DP).

• The TD target is an estimate because it samples the expected value and it uses the current estimate V_t instead of the true V^{π}. Thus, TD methods combine the sampling of MC with the bootstrapping of DP.

Temporal-Difference Learning

Some advantages of TD learning:

• TD methods do not require a model of the environment (DP does)

• TD can be naturally implemented in an on-line, fully incremental fashion. With MC methods, one needs to wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step.

* Surprisingly, this turns out to be a critical consideration (NB: some applications have very long episodes).

* TD has been shown to converge to V^{π}, in the mean for a sufficiently small constant step-size parameter.

Q-Learning

• **Q-learning** is an off-policy TD control algorithm. In its simplest form, one-step Q-learning, it is defined by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_t) - (s_t, a_t) \right]$$

• In this case, the learned action-value function, Q, directly approximates Q*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs.

• The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated.

Q-Learning

How do we use Q-learning in practice?

Initialize Q(s,a) to all zeros

Initialize s

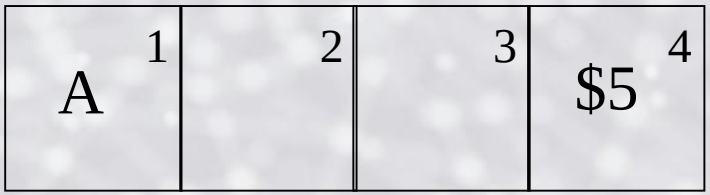
Repeat until stopping condition:

-- select action a
-- take action a and receive reward r
-- observe new state s'
-- update Q(s,a):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_t) - (s_t, a_t) \right]$$

-- update s ← s'

Example



A is our agent, who takes an action at each timestep.

Only action in square 1 is **F**orward.

Actions in squares 2 and 3 are (**F**orward, **B**ack)

Being in square 4 gives reward of $5

Only action in square 4 is **S**top

No other rewards or penalties.

Set $\gamma = .9$

Set $\eta = 1$

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 1
Current state *s* = 1

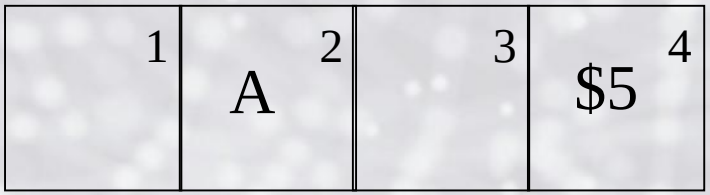
Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 1
Current state *s* = 1
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

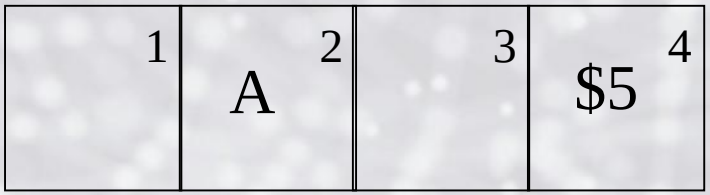
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 1
Current state *s* = 1
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

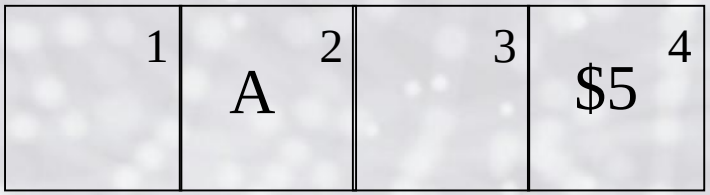
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 1
Current state *s* = 1
Action = F
r = 0
s' = 2

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

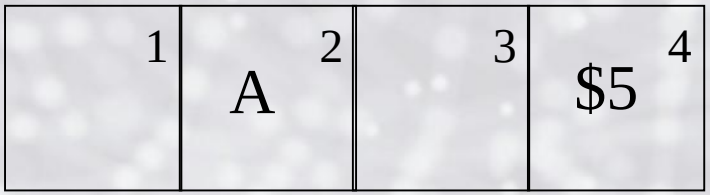


Episode 1
Current state *s* = 1
Action = F
r = 0
s' = 2

$$Q(1,F) = 0 + \max_{a'}\left[Q(2,a')\right] = 0$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

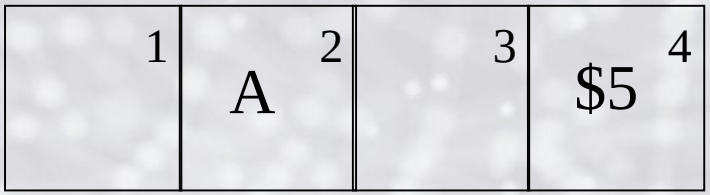
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 1
Current state *s* = 2

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

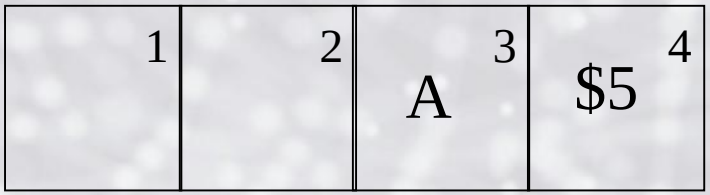
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 1
Current state *s* = 2
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

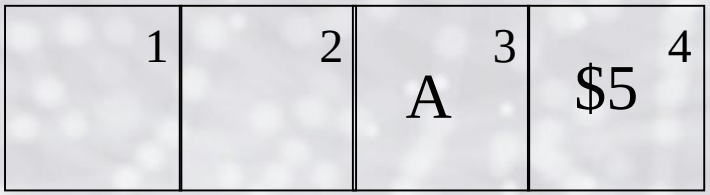
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 1
Current state *s* = 2
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

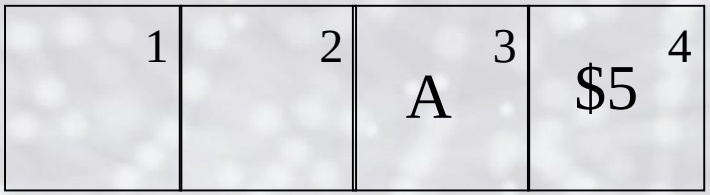
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 1
Current state *s* = 2
Action = F
r = 0
s’ = 3

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

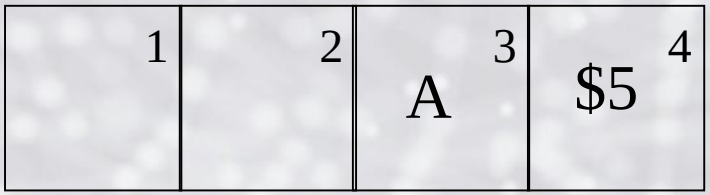


Episode 1
Current state *s* = 2
Action = F
r = 0
s' = 3

$$Q(2,F) = 0 + (0 + .9 \max_{a} Q(s',a') - Q(s,a)) = 0$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

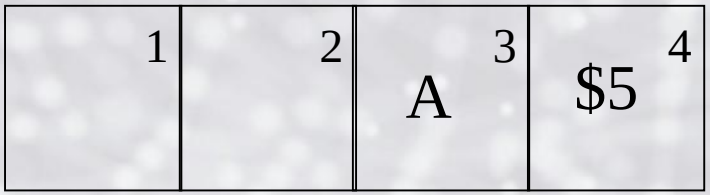
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 1
Current state *s* = 3

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 1
Current state *s* = 3
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 1
Current state *s* = 3
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 1
Current state *s* = 3
Action = F
r = $5
s' = 4

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 1
Current state *s* = 3
Action = F
r = $5
s' = 4

$$Q(3,F) = 0 + (\$5 + .9 \max_{a} Q(s',a') - Q(s,a)) = \$5$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	0	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 1
Current state *s* = 3
Action = F
r = $5
s' = 4

$$Q(3,F) = 0 + (\$5 + .9 \max_{a} Q(s',a') - Q(s,a)) = \$5$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 1
Current state *s* = 4

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 1
Current state *s* = 4
Action = Stop

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 2
Current state *s* = 1

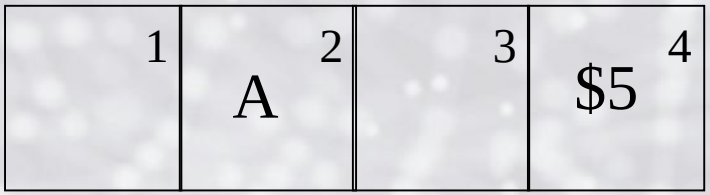
Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 2
Current state *s* = 1
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

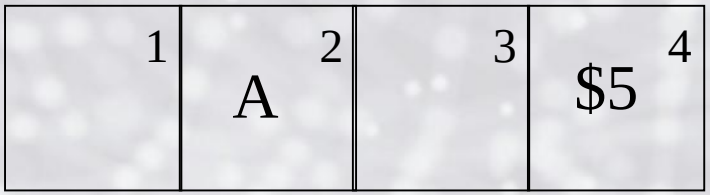
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 2
Current state *s* = 1
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

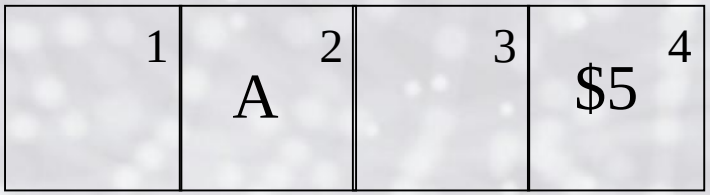
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 2
Current state *s* = 1
Action = F
r = 0
s' = 2

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

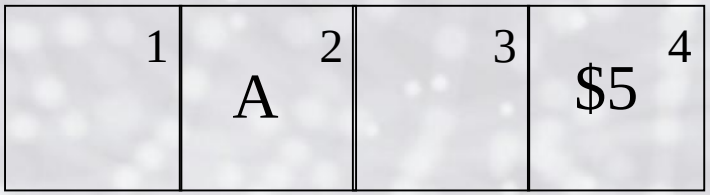


Episode 2
Current state *s* = 1
Action = F
r = 0
s' = 2

$$Q(1,F) = 0 + (0 + .9 \max_{a} Q(s',a') - Q(s,a)) = 0$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

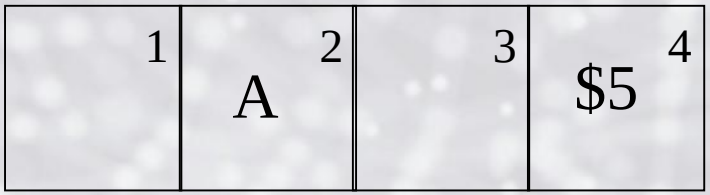
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 2
Current state *s* = 2

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

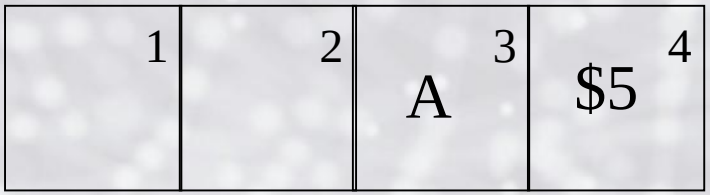
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 2
Current state *s* = 2
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

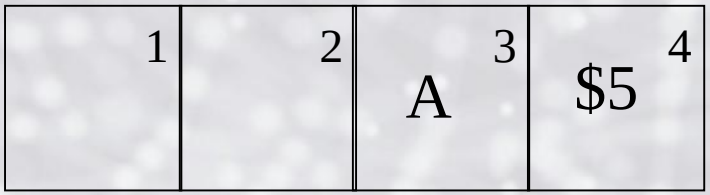
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 2
Current state *s* = 2
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

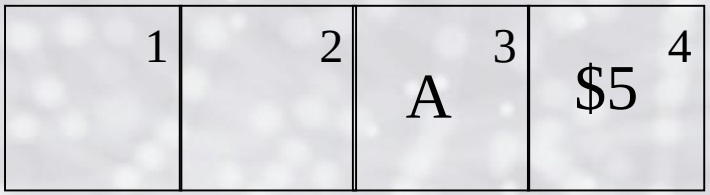
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 2
Current state *s* = 2
Action = F
r = 0
s' = 3

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

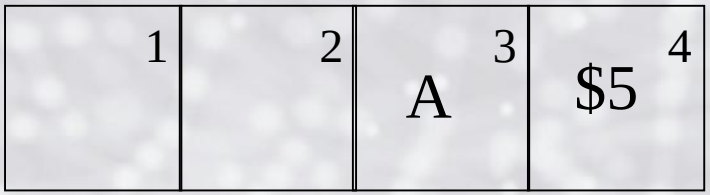


Episode 2
Current state *s* = 2
Action = F
r = 0
s' = 3

$$Q(2,F) = 0 + (0 + .9 \max_{a} Q(s',a') - Q(s,a))$$
$$= 0 + 0 + (.9)(\$5) = \$4.50$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	0	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

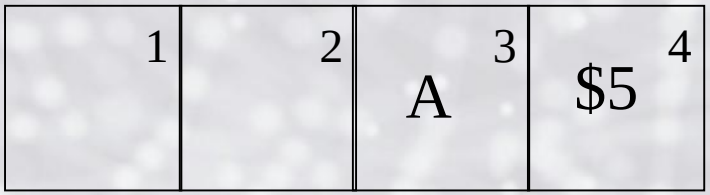


Episode 2
Current state *s* = 2
Action = F
r = 0
s' = 3

$$Q(2,F) = 0 + (0 + .9 \max_{a} Q(s',a') - Q(s,a))$$
$$= 0 + 0 + (.9)(\$5) = \$4.50$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

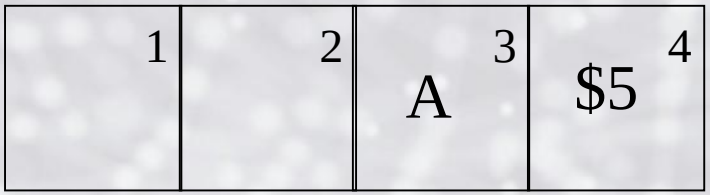
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 2
Current state *s* = 3

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 2
Current state *s* = 3
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 2
Current state *s* = 3
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 2
Current state *s* = 3
Action = F
r = $5
s' = 4

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 2
Current state *s* = 3
Action = F
r = $5
s' = 4

$$Q(3,F) = \$5 + (\$5 + .9 \max_{a} Q(s',a') - Q(s,a))$$
$$= \$5 + \$5 + 0 - \$5 = \$5$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 2
Current state *s* = 4

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 2
Current state *s* = 4
Action = Stop

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state *s* = 1

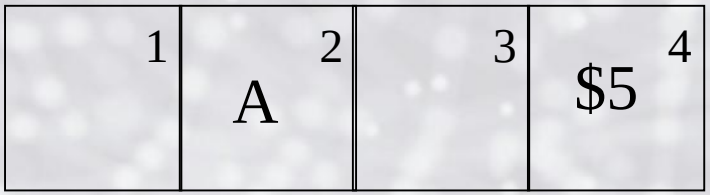
Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state *s* = 1
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

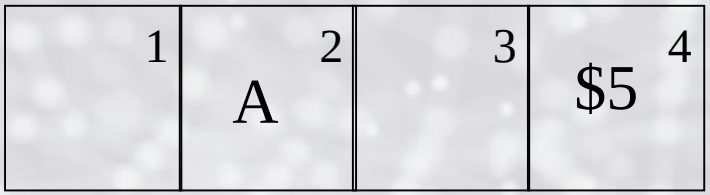
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 3
Current state *s* = 1
Action = F

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

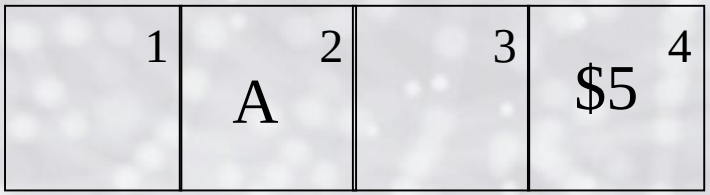
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 3
Current state *s* = 1
Action = F
r = 0
s' = 2

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

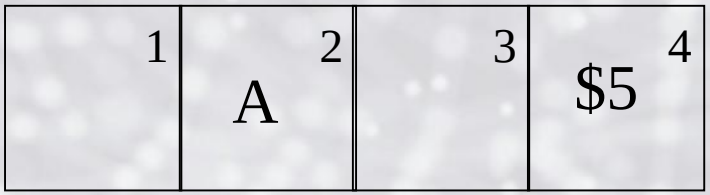


Episode 3
Current state *s* = 1
Action = F
r = 0
s' = 2

$$Q(1,F) = 0 + (0 + .9 \max_{a} Q(s',a') - Q(s,a))$$
$$= 0 + 0 + (.9)(\$4.50) - 0 = \$4.05$$

Q(s,a)	Forward	Back	Stop
1	0	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

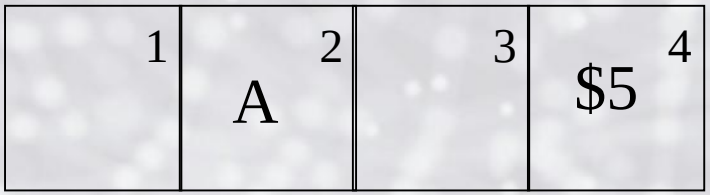


Episode 3
Current state s = 1
Action = F
r = 0
s' = 2

$$Q(1,F) = 0 + (0 + .9 \max_{a} Q(s',a') - Q(s,a))$$
$$= 0 + 0 + (.9)(\$4.50) - 0 = \$4.05$$

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

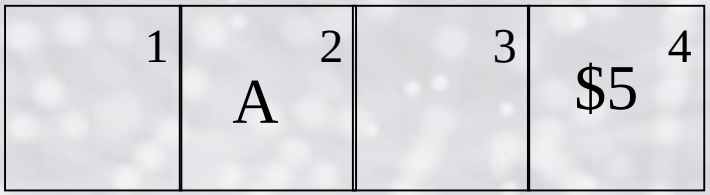
$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 3
Current state *s* = 2

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$



Episode 3
Current state *s* = 2
Action = B

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state *s* = 2
Action = B

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state *s* = 2
Action = B
r = 0
s' = 1

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

1	2	3	4
A			$5

Episode 3
Current state *s* = 2
Action = B
r = 0
s' = 1

$$Q(2,B) = 0 + (0 + .9 \max_{a} Q(s',a') - Q(s,a))$$

$$0 + 0 + (.9)(\$4.05) - 0 = \$3.65$$

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	0	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state s = 2
Action = B
r = 0
s' = 1

$$Q(2,B) = 0 + (0 + .9 \max_{a} Q(s',a') - Q(s,a))$$

$$0 + 0 + (.9)(\$4.05) - 0 = \$3.65$$

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	**$3.65**	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state *s* = 1

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	**$3.65**	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state *s* = 1
Action = F

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	**$3.65**	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state *s* = 1
Action = F
r = 0
s' = 2

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	**$3.65**	X
3	**$5**	0	X
4	X	X	0

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,(r + \gamma \max_{a'} Q(s',a') - Q(s, a))$$

Episode 3
Current state s = 1
Action = F
r = 0
s' = 2

$$Q(1,F) = \$4.05 + (0 + .9 \max_{a} Q(s',a') - \$4.05)$$

$$\$4.05 + 0 + (.9)(\$4.50) - \$4.05 = \$4.05$$

Q(s,a)	Forward	Back	Stop
1	**$4.05**	X	X
2	**$4.50**	**$3.65**	X
3	**$5**	0	X
4	X	X	0

Q-Learning

- Results: Q-learning converges to optimal policy – even if you're acting sub-optimally!

- This is called **off-policy learning**.

Caveats:

- You have to explore sufficiently.
- You have to make learning rate small enough (but also not decrease it too quickly).

Q-Learning

- Note that in all of the previous discussion, Q(*s*, *a*) was assumed to be a look-up table, with a distinct table entry for each distinct (*s*,*a*) pair.

- More commonly, Q(*s*, *a*) is represented as a function (e.g., a neural network), and the function is estimated (e.g., through back-propagation).

Summary of RL

- In addition to the agent and environment, there are (4) key ingredients to RL:
- (1) A **policy** (usually denoted π) is a mapping, $\pi: S \rightarrow A$. The policy is the decision-making function for the agent.
- (2) A **reward function** maps each perceived state (or state-action pair) of the environment to its corresponding reward value: $r: S \times A \rightarrow \mathbb{R}; r(s_t, a_t) = r_t$. Most often the reward function is unknown to the agent.
- (3) A **value function** specifies what is 'good' in the long-run for the agent. In this sense, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

  Formally, $V^{\pi}(s) = E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \middle| s_t = s\right]$, with 'discount' $0 \leq \gamma \leq 1$.
- (4) A **model** of the environment, consisting of the triple: $(S, A, \delta: S \times A \rightarrow S)$. Where S is the 'state space' (assumed finite), A is the 'action space' (normatively, $|A| \ll |S|$) and δ is the transition function inherent to the environment.
- Put simply, the goal of the learning task is to learn an optimal policy, π^*, that maximizes $V^{\pi}(s)$ for all states.

Summary of RL

- At first blush, it seems as though the best strategy for learning the optimal policy, π^*, might be to directly learn $V^*(s)$.
- To do so, we could solve the recurrence relation:

$$V^{\pi}(s) = E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \,\middle|\, s_t = s\right] = \ldots = \sum_a \underbrace{\pi(s,a)}_{\substack{\text{prob. of taking} \\ \text{action } a \text{ in state } s}} \sum_{s'} \underbrace{P^a_{ss'}}_{\substack{\text{transition} \\ \text{probability}}} \left[R^a_{ss'} + V^{\pi}(s')\right]$$

- This equation is known as the **Bellman Equation for V^{π}**. The Bellman equation averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.
- When the agent has **complete knowledge** of its environment, then, in theory, we could directly solve (or even approximate) for the value function.
- In this fashion, the optimal policy would be defined as follows: $\pi^*(s) = \arg\max_a \left[r(s,a) + \gamma V^*(\delta(s,a))\right]$, so that action a is chosen in such a way so as to maximize the sum of the immediate reward and the discounted expected reward for the successor state.
- The problem, however, with this approach is that it requires full knowledge of the transition and reward functions which are unknown to the agent.
- Alternatively, a common approach for learning an optimal policy in RL with incomplete environmental knowledge is to use **temporal-difference learning** (TD), including **Q learning**.

Summary of RL

- Define the *value* of the state-action pair as the mapping: $Q: S \times A \to \mathbb{R}$; let $Q^{\pi}(s,a) = E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \middle| s_t = s, a_t = a\right]$.
- Q^{π} denotes the action-value for policy π, which is to say, the *expected return* starting from state s, taking action a and following policy π henceforth.
- One-step Q learning is defined as the following iterative update:

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(r_{t+1} + \gamma \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\substack{\text{estimate of optimal} \\ \text{future value}}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right)$$

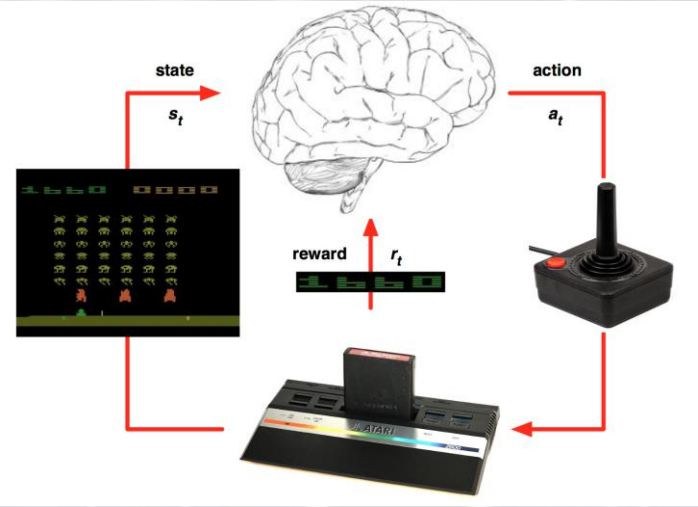
- Here the learned action-value function, Q, directly approximates Q*, the optimal action-value function, independent of the policy being followed (we say the method is 'off-policy').
- Q learning algorithm schematic:

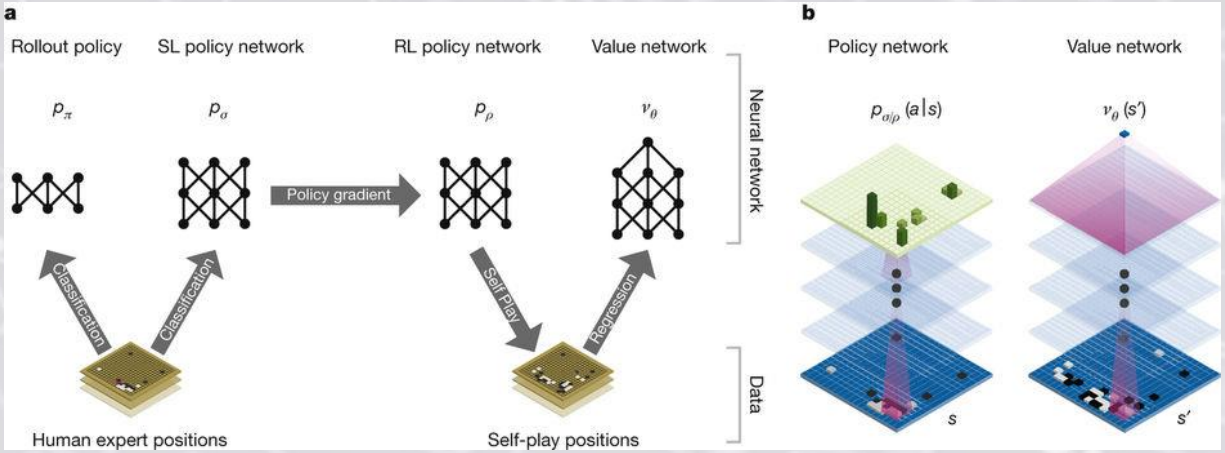
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

Figure 6.12: Q-learning: An off-policy TD control algorithm.

- Famously, Tesauro (early 1990s) developed **TD-Gammon** using Q-learning (computed with NN equipped with 50 hidden units; trained originally on 300,00 games against itself); by 1995 TD-Gammon was competitive with best human players in the world.

Deep Reinforcement Learning





Deep Reinforcement Learning

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**
Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

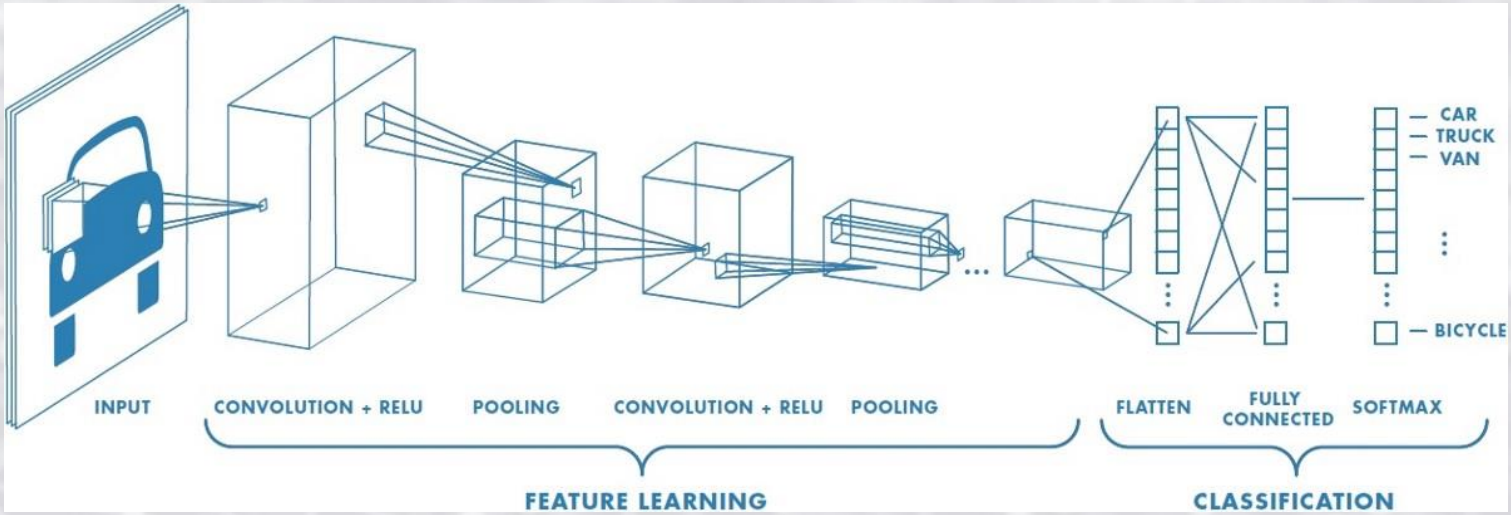
1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [11, 22, 16] and speech recognition [6, 7]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have ex-

Demis Hassabis (co-founder DeepMind)

https://youtu.be/rbsqaJwpu6A



- **Deep learning:**

  Requires large amount of hand-labeled data

  Assumes data samples are iid, with stationary distribution

- **Reinforcement learning:**

  Must learn from sparse, noisy, delayed reward function

  "Samples" are not independent

  Data distribution can change as system learns "online"

- Uses convolutional neural network (CNN):
  - Input is raw pixels of video frames (~ the "state")
- Output is estimated Q(*s*,*a*) for each possible action

Deep Reinforcement Learning

- System learns to play Atari 2600 games;
- 210x160 RGB video at 60 Hz.
- Designed to be difficult for human players
- "Our goal is to create a single neural network agent that is able to successfully learn to play as many of the games as possible."
- No game-specific info provided. No hand-designed visual features.
- Learns exclusively from video input, the reward, and terminal signals. Network architecture and hyperparameters kept constant across all games.

Deep Reinforcement Learning

Methodological details

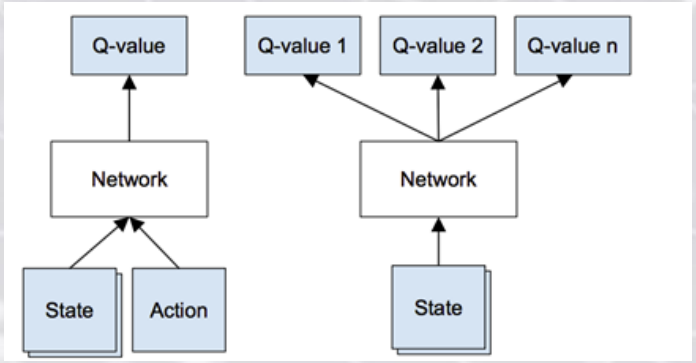
- **Model Architecture**: The researchers developed a novel agent, a **deep Q-network** (*DQN*) which is able to combine RL with 'deep' NNs (that is to say they have many layers).
- Hassabis *et al.* use a deep CNN, which employs hierarchical layers of tiled convolutional filters to mimic the effects of 'receptive fields'.
- The goal of the game-playing agent is to select actions in a fashion that maximizes cumulative feature rewards. Formally, the deep CNN is used to approximate the optimal action-value, which is the maximum sum of discounted rewards achievable by a behavior policy $\pi = P(a \mid s)$:

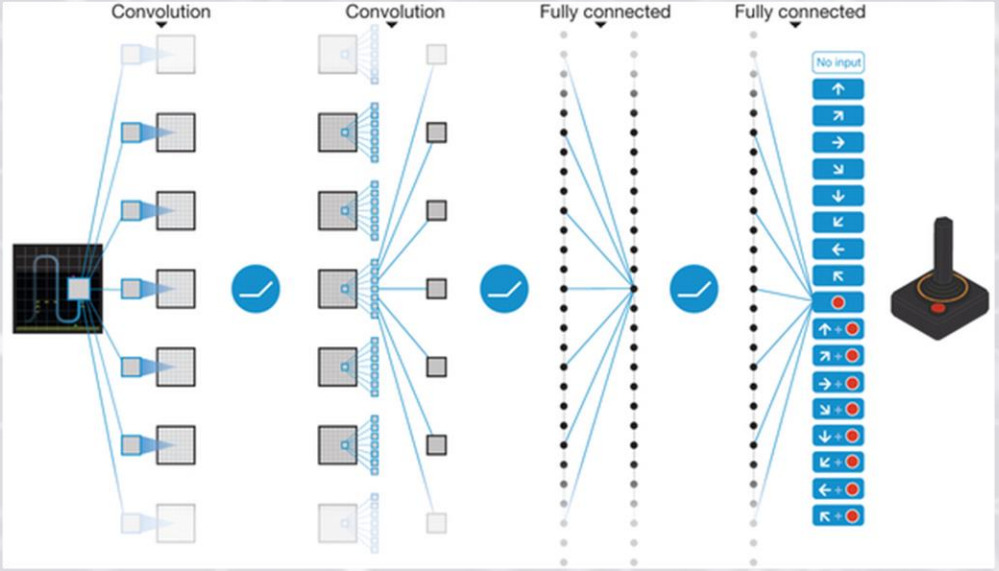
$$Q^*(s,a) = \max_{\pi} E\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \middle| s_t = s, a_t = a, \pi \right]$$

- RL is known to be **numerically unstable** when NNs are used to approximate Q functions; this is largely due to (2) issues: (1) many data sequences of state action pairs are highly correlated; (2) minute updates in Q-value approximations can significantly impact the behavior of an optimal policy.
- To get around these potential shortcomings, the authors propose: (1) the use of '**experience replay**' that randomizes over the data and thus removes many data correlations; and (2) the use of iterative updates to Q-values that are only periodically updated.

Deep Reinforcement Learning

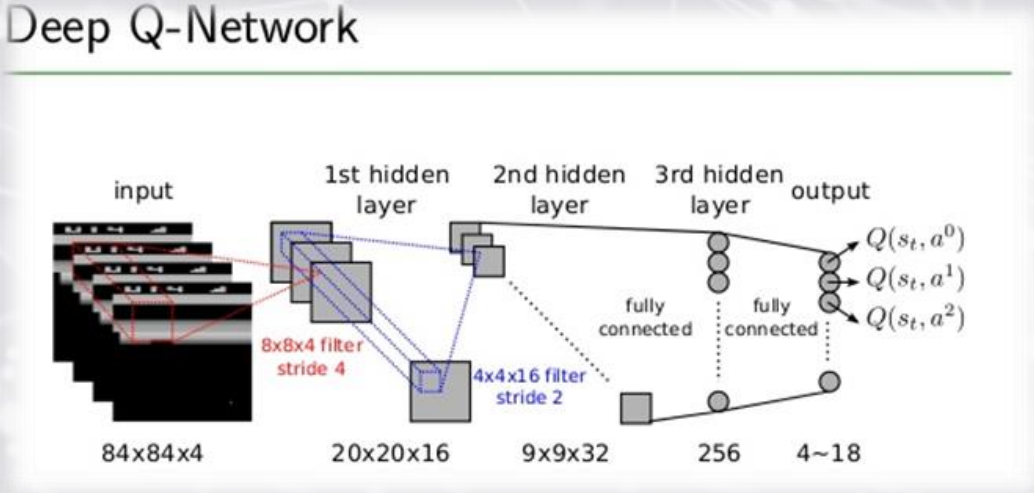
- **Model Architecture (cont'd)**:
- In previous approaches, researchers applied NNs to approximate Q-values using histories + actions as inputs to the NN. This scheme presents a significant drawback, however, since a separate forward pass is required to compute the Q-value for each individual action.
- Instead, in the current method, the outputs correspond to the predicted Q-values of the individual actions for the input state. This presents a significant computational advantage over previous methods; Q-values are accordingly computed for **all possible actions** in a given state with only a single forward pass through the network.





Deep Reinforcement Learning

- **Model Architecture (cont'd)**:
- The input to the NN consists of an 84x84x4 image produced by the preprocessing map.
- The first hidden layer convolves 32 filters of size 8x8 with stride 4 and applies a RELU.
- The second hidden layer convolves 64 filters of size 4x4 with stride 2, again followed by a RELU.
- The third hidden layer convolves 64 filters of size 3x3 with stride 1, with RELU.
- The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected layer with an output for each action. The number of valid actions varies between 4 and 18 in the games considered.



Deep Reinforcement Learning

- **Algorithm Details:**
- Sequences of actions and observations, $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$, are input to the algorithm, which then learns game strategies depending upon these sequences.
- This formalism gives rise to a large (but finite) Markov decision process (*MDP*).
- The optimal action-value function in this setting obeys the aforementioned Bellman equation. Normally, where possible, one would use the Bellman equation as an iterative update for the action-value approximation.
- In practice for large sequence MDPs, this approach is impractical because it requires estimating the action-value function for each sequence separately, without any generalization.
- Alternatively, the authors use a NN, *viz.*, a **Q-Network** (with parameter set θ) for the approximation: . $Q(s,a;\theta) \approx Q^*(s,a)$
- Note that without an efficient state-action value approximation, the number of action pair values is astronomically large ($\sim 10^{67970}$)!
- The Q-Network is trained by adjusting the parameters θ_i at each iteration to reduce the MSE in the Bellman equation, this yields the loss function:

$$L(\theta_i) = E\left[\left(y - Q(s,a;\theta_i)\right)^2\right] + E\left[V[y]\right]^2, \; with: y = r + \gamma \max_{a'} Q^*(s',a';\theta_i^-)$$

- Differentiating this loss function wrt the weights yields a gradient used in stochastic gradient descent. Note that state-action sequences are generated off-policy; the behavior distribution is ε-greedy.

Deep Reinforcement Learning

- **Putting it all together…**
- The agent selects and executes actions according to an ε–greedy policy based on Q. The Q-function works on fixed length representations of histories produced by the pre-processing function ϕ.
- The algorithm modifies standard online Q-learning in (2) ways to make it suitable for training a large NN.
- (1) The authors employ a technique called 'experience replay', in which the agent's experiences at each time step $e_t=(s_t,a_t,r_t,s_{t+1})$ are stored in a data set $D_t=\{e_1,\ldots,e_t\}$ pooled over many episodes.
- During the inner loop of the algorithm the authors apply Q-learning updates to samples of experience, (s,a,r,s')~U(D), drawn at random from the pool of stored samples. (this improves data efficiency and reduces correlations between samples and the presence of feedback loops in the in training process).
- By using experience replay, the behavior distribution is averaged over many of its previous states, thereby smoothing out learning and avoiding oscillations or avoidance in the parameters.
- Note that the uniform sampling gives equal importance to all transitions in the replay memory (a possible improvement would be to apply a more sophisticated sampling strategy similar to prioritized sweeping).
- (2) To further improve stability, a separate network for generating the targets (y_i's) in the Q-learning update. More precisely, every C updates the authors cloned the network Q to obtain a target network that is used for generating Q-learning targets for the following C updates to Q.

Deep Reinforcement Learning

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

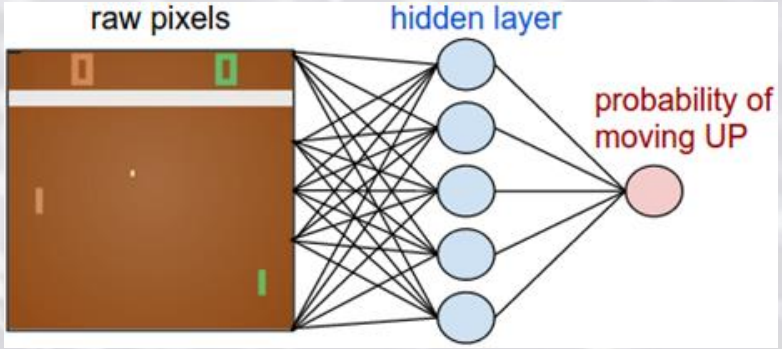
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

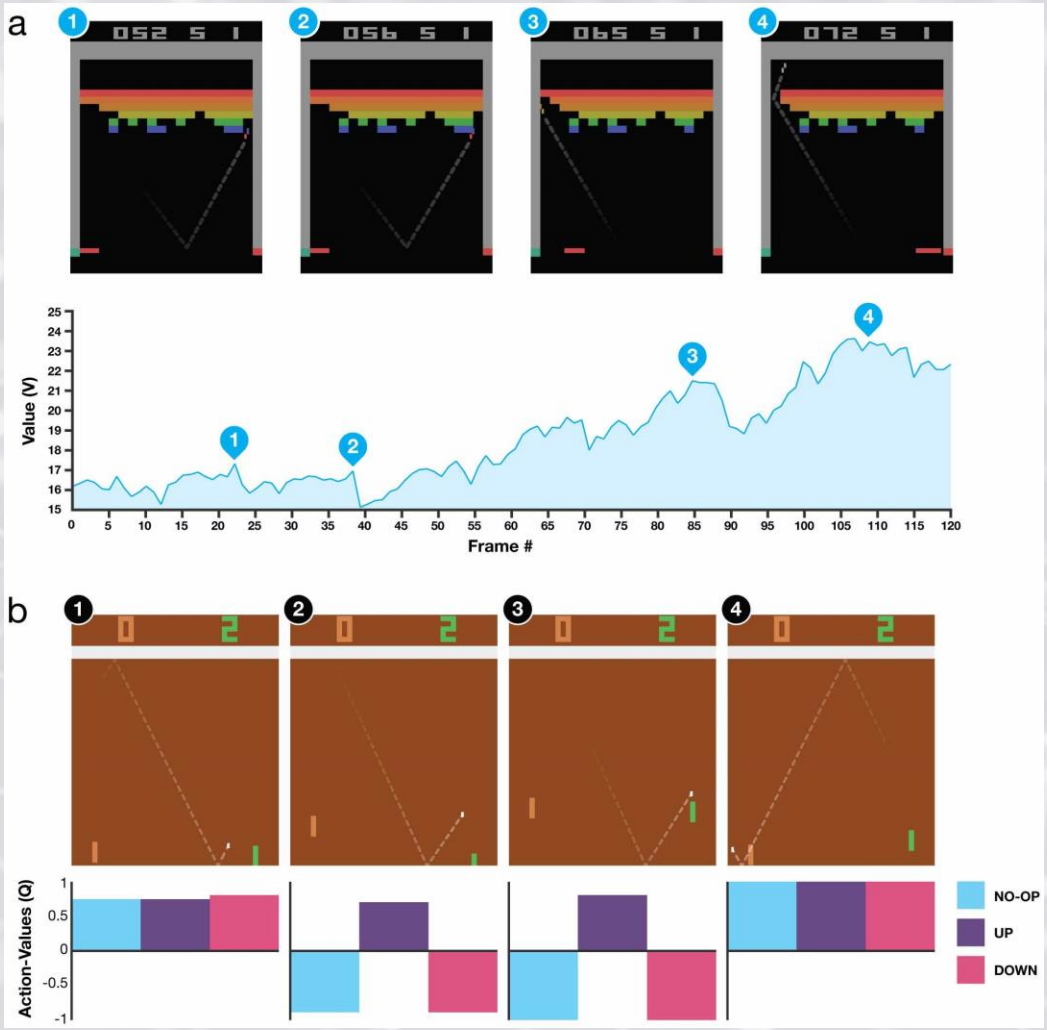
    Every C steps reset $\hat{Q} = Q$

  End For

End For



Deep Reinforcement Learning

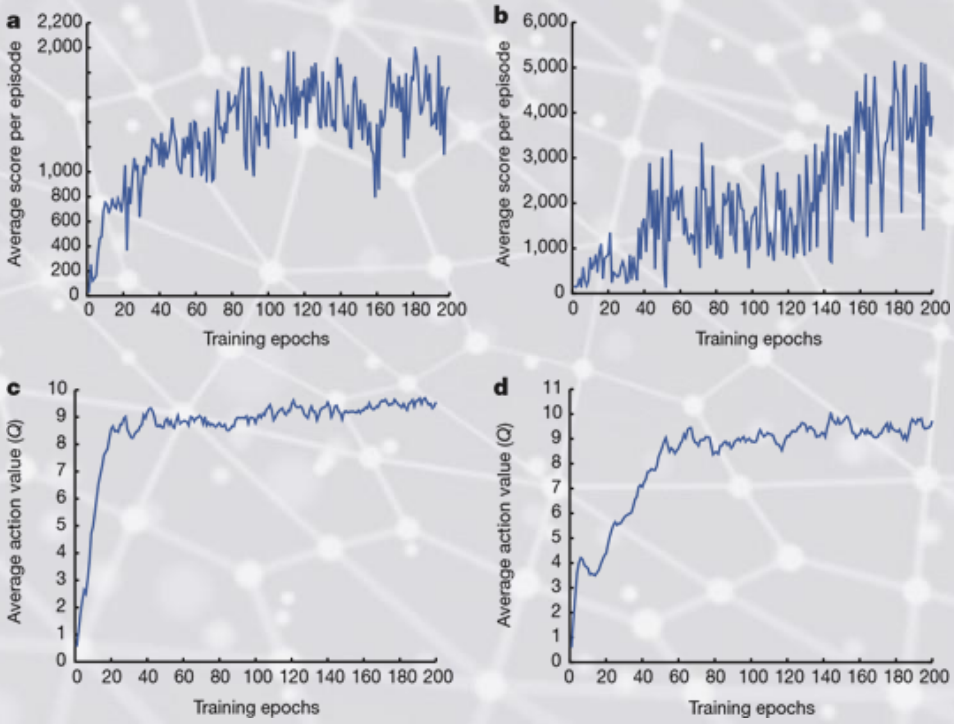


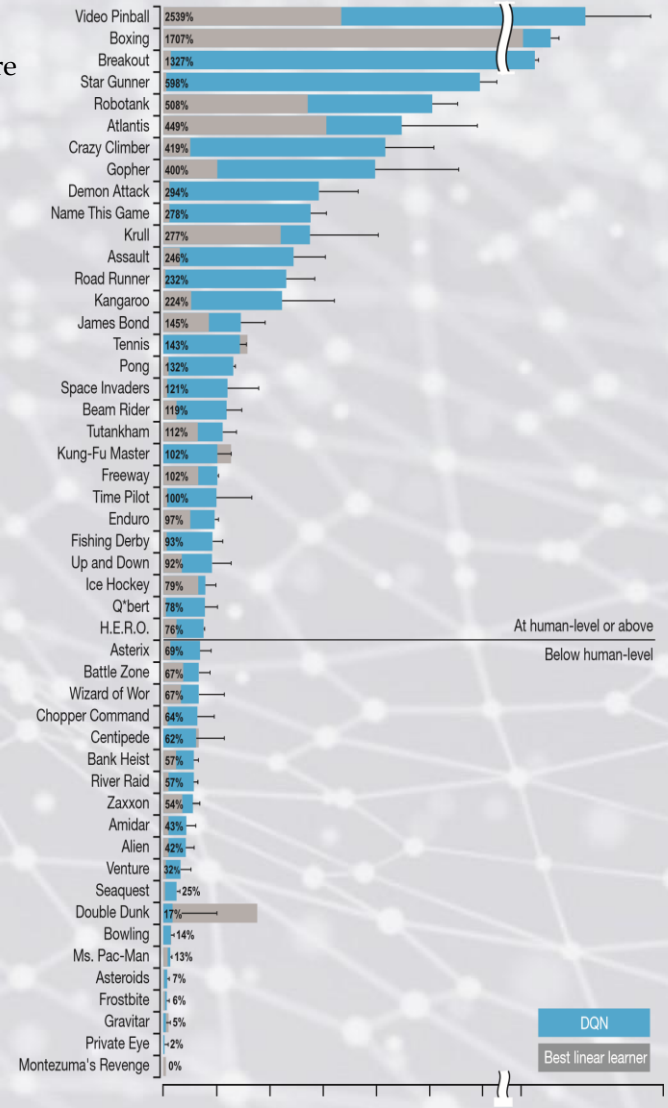
Deep Reinforcement Learning

- **Results**

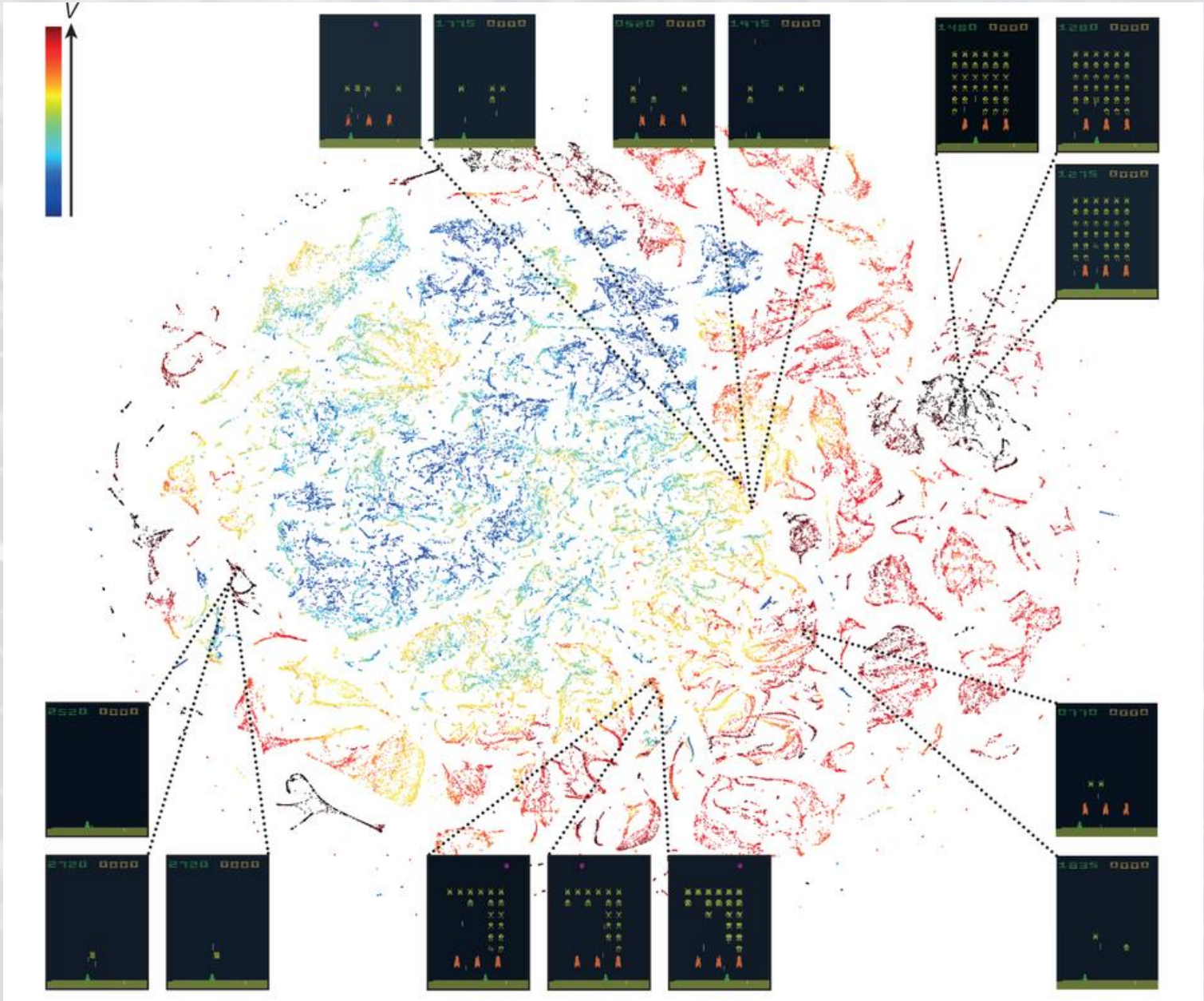
- The DQN agent performed at a level comparable to that of a professional human games test across the set of 49 games, achieving more than 75% of the human score on more than half the games.

- The authors' method was able to train large NNs using RL with stochastic gradient descent in a stable manner – illustrated by the temporal evolution of two indices of learning (the agent's average score-per-episode and average predicted Q-values).





Deep Reinforcement Learning



ARTICLE

doi:10.1038/nature24270

Mastering the game of Go without human knowledge

David Silver[1*], Julian Schrittwieser[1*], Karen Simonyan[1*], Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here we introduce an algorithm based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game rules. AlphaGo becomes its own teacher: a neural network is trained to predict AlphaGo's own move selections and also the winner of AlphaGo's games. This neural network improves the strength of the tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program AlphaGo Zero achieved superhuman performance, winning 100–0 against the previously published, champion-defeating AlphaGo.

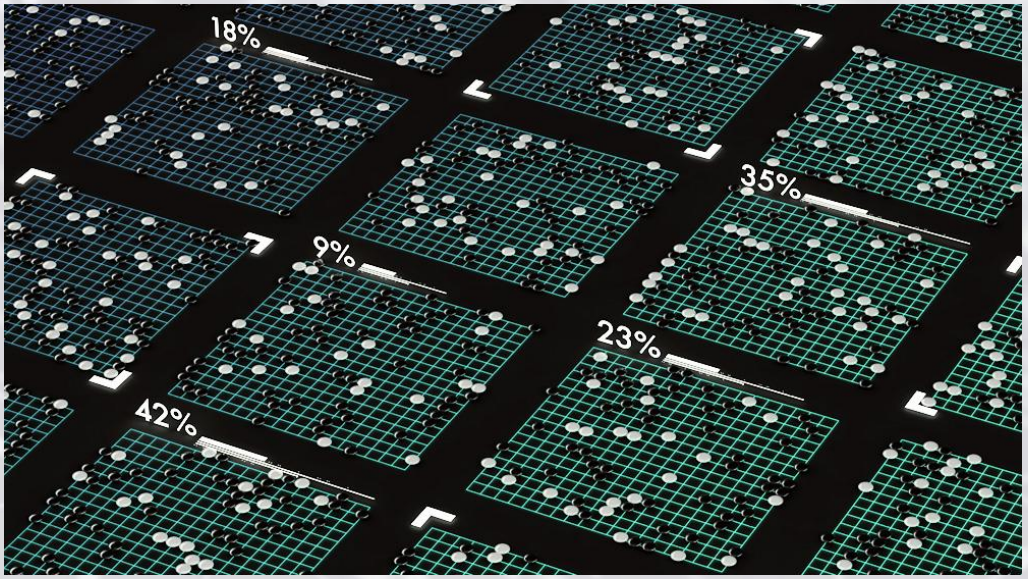
Much progress towards artificial intelligence has been made using supervised learning systems that are trained to replicate the decisions of human experts[1–4]. However, expert data sets are often expensive, unreliable or simply unavailable. Even when reliable data sets are available, they may impose a ceiling on the performance of systems trained in this manner[5]. By contrast, reinforcement learning systems are trained from their own experience, in principle allowing them to exceed human capabilities, and to operate in domains where human expertise is lacking. Recently, there has been rapid progress towards this goal, using deep neural networks trained by reinforcement learning. These systems have outperformed humans in computer games, such as Atari[6,7] and 3D virtual environments[8–10]. However, the most challenging domains in terms of human intellect—such as the game of Go, widely viewed as a grand challenge for artificial intelligence [11]—require a precise and sophisticated lookahead in vast search spaces. Fully general methods have not previously achieved human-level performance in these domains.

AlphaGo was the first program to achieve superhuman performance in Go. The published version[12], which we refer to as AlphaGo Fan, defeated the European champion Fan Hui in October 2015. AlphaGo Fan used two deep neural networks: a policy network that outputs

trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it uses only the black and white stones from the board as input features. Third, it uses a single neural network, rather than separate policy and value networks. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts. To achieve these results, we introduce a new reinforcement learning algorithm that incorporates lookahead search inside the training loop, resulting in rapid improvement and precise and stable learning. Further technical differences in the search algorithm, training procedure and network architecture are described in Methods.

Reinforcement learning in AlphaGo Zero

Our new method uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\boldsymbol{p}, v) = f_\theta(s)$. The vector of move probabilities $\boldsymbol{p}$ represents the probability of selecting each move a (including pass), $p_a = \Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s. This neural network combines the roles



https://www.youtube.com/watch?v=53YLZBSS0cc

Fin