Learning in games I

Jonathan Shapiro

School of Computer Science
University of Manchester

Announcements

- This ends the content from Andrea Schalk's Notes on Games (The Theory of Games).
- Recommended readings in Blackboard.
- New set of notes "Learning in Games". First two chapters on Blackboard.
- Four lectures on Learning in Games.
- Friday is last day to sign up for a project group.

Why use learning?

From the Notes on Games, page 82,

> *"When the first game playing programs were written it was envisaged that machine-based learning would be the most important aspect in developing them. This faith in Artificial Intelligence has not proved appropriate in practice. All world-class game-playing programs use other principles foremost."*

- This was largely true in 2012. The big successes used variants of Mini-Max search, including almost all solved games: chess[1], checkers/draughts, Connect-4, Kalah (6,6), and many others.
- It is not true today.

[1] Not solved, but computers are much better than humans

Some highlights in AI for game playing

1952 – 1962: $\star$ Arthur Samuel (IBM) produces a computer program which plays checkers (draughts), using pruned search and learning.

1992: $\star$ Gerard Tesauro produces TD-Gammon, a computer program which uses TD-lambda reinforcement learning to play backgammon at to a world-class standard.

1997: IBM's Deep Blue chess program beats then world champion Garry Kasparov.

2011: $\star$ IBM's Watson beats top human players at the TV game show Jeopardy.

2015: $\star$ Head's-up, limit Texas Hold'em is solved by a group from the University of Alberta.

2015 – 2017: $\star$ Google's DeepMind AlphGo beats champion players at Go including the world number 1.

Those marked with $\star$ used learning!

Heads-up No-limit Poker results

There are two competing groups.

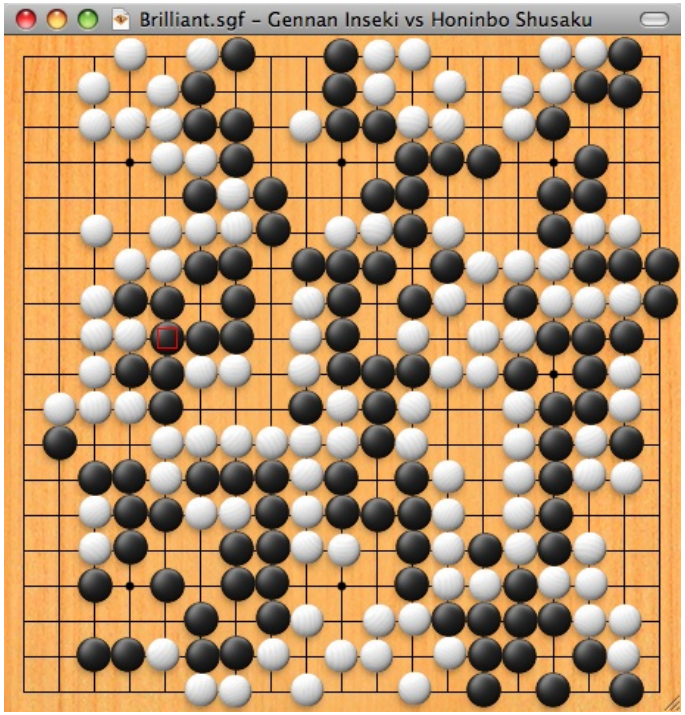
- 2016: **University of Alberta's Deepstack** defeated 11 professional poker players "with only one outside the margin of statistical significance".
- 2017: **Carnegie Mellon University's Libratus** played $120,000$ hands against four "top-class" human poker players over 20 days. The computer won a total of $\$1,766,087$.

Learning in two-player, zero-sum imperfect information games required a breakthrough, which we *cannot* cover in this course.

When is learning preferred over Mini-Max search

- When good heuristics cannot be found.
- When the branching factor is very high.
- When equilibria are hard to compute:
  - Games of incomplete information.
  - General-sum games.
  - Games with many players.

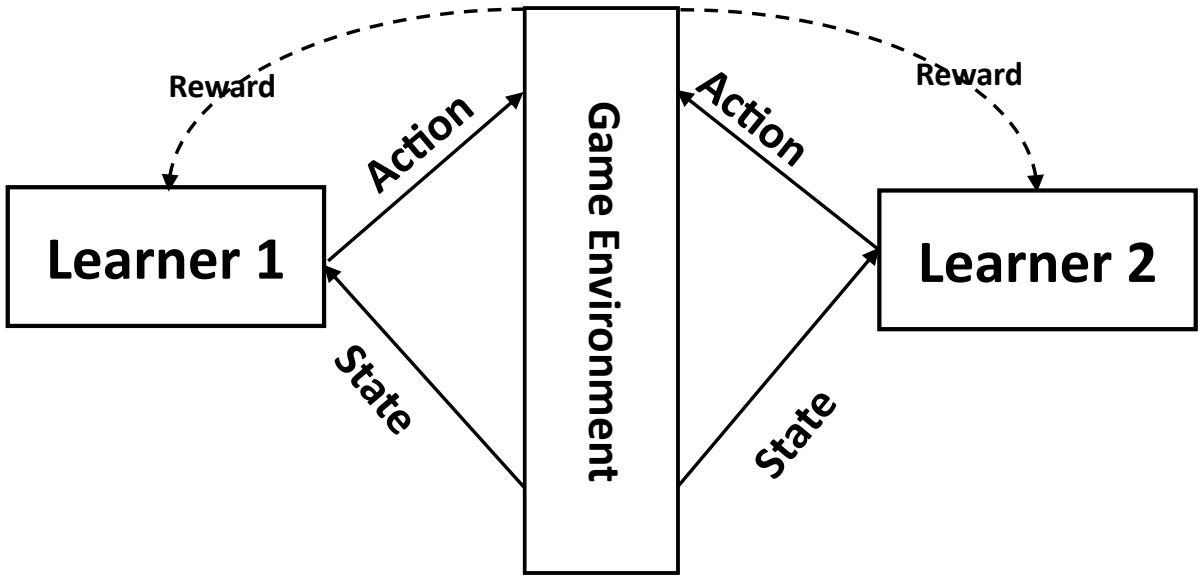
For example, Go



Go is a hard game for computers to play: it has a high branching factor, a deep tree, and lacks any known reliable heuristic value function for non-terminal board positions.

From “A survey of Monte Carlo Tree Search Methods”, Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton, IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 1, March 2012.

A key idea: Learning through “self play”



Two independent learning agents play games against each other.

Two independent learning agents play games against each other.

- Starting from naive states. (Typically)
- Incrementally improve.

Forming an “arms race”.

How this should work

Each player in turn:

1. Plays an action
2. Observes the new game position and any rewards (win/lose, etc)‘
3. Strengthens moves leading to wins; suppresses those leading to losses.
4. By playing many many games, learns to become a strong player (we hope).

But how is this possible?

Why is this problem hard?

Often,

- Learn while playing ("on-line" or "real-time" learning).
- Feedback from outcome of the game reveals you did something right or wrong, but not *what* you did right or wrong.
- A sequence of moves is required before the outcome is revealed. You don't know which move was crucial to the outcome.

These are characteristics of a form of machine learning called *reinforcement learning*.

What is **Reinforcement learning**?

Learning from rewards:

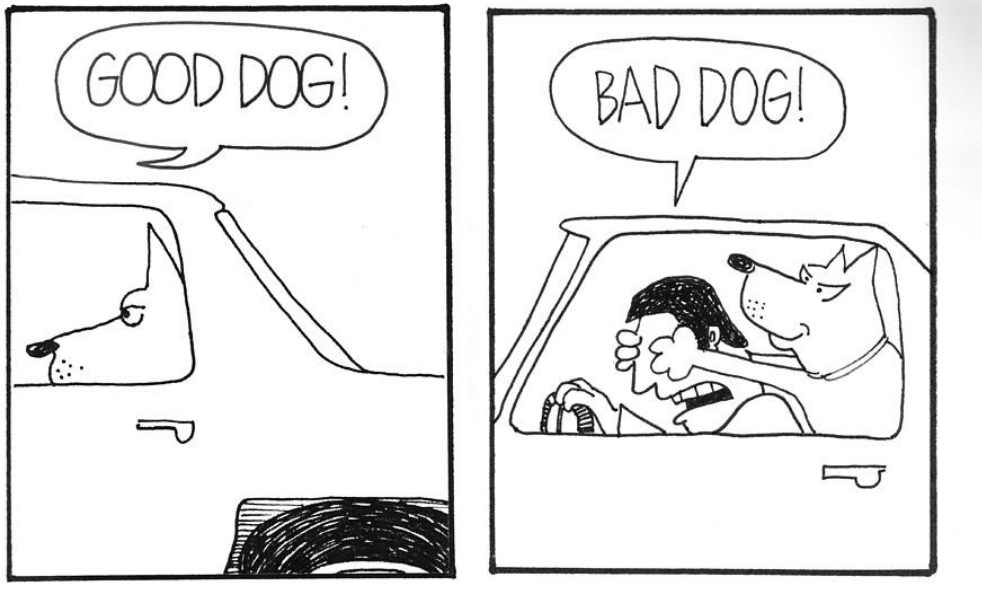
1. Observe the situation (the *state*);
2. Perform an *action*;
3. Receive a *reward*, positive or negative.

By doing this repeatedly — learn to

- maximize the positive rewards;
- minimize the negative rewards.

Well-know examples

Animal learning:



Drawing by Rhodes

Rumsey from `fineartamerica.com`

Human infant learning: How you learned to walk; and get mummy's attention.

Game playing: Need to learn, by playing many, many games, those moves which lead to wins.

Used when the best action/response is not known

There may be appropriate actions for every situation.

- No expert can say what it is.
- No labeled training data exists.

The best action needs to be discovered, by trial and error.

The reinforcement learning problem

- Learning where only the *quality* of the responses or actions can known (e.g. win/lose, good/bad, goal achieved/goal not achieved), not what the correct actions are.
- Alternatively, numerical rewards (profits/losses) can be observed.
- The reinforcement information may be available only after a sequence of actions has been taken.
- The environment may include opponents, who also might be learning.

Reinforcement Learning Concepts

Different terminology than game theory

Rewards: Denoted r_t at time t. Can be positive or negative

- e.g. win game at time t ($r_t > 0$)
- e.g. no result at time t ($r_t = 0$ or slightly negative to encourage short games)
- lose game at time t ($r_t < 0$).

State: The current situation, board state, agent location, etc. Denoted S_t — the state at time t.

Policy: What to do in any situation. What action to take in each situation or state.

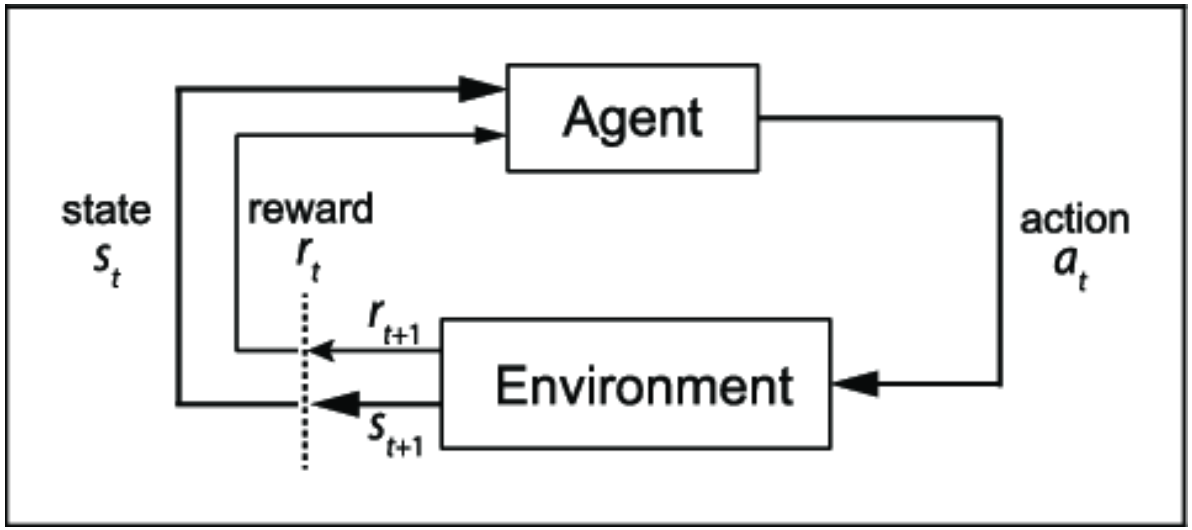
Reinforcement Learning Concepts

Reinforcement learning: Learn an effective policy simply by taking actions and observing rewards.

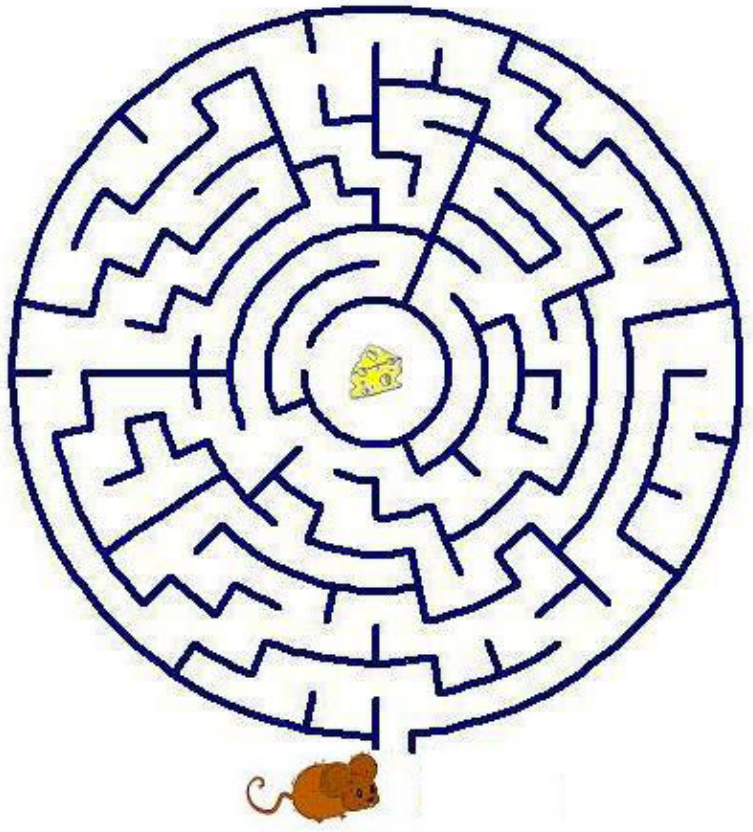
- learning by trial and error
- no teacher.

I will just show a few highlights of the application of reinforcement learning to games.

Reinforcement Learning Concepts



Reinforcement learning example — mouse in a maze



Possibilities:

States: Could be decision points.

Actions: Go left, right, straight.

A path

Another Key idea: The exploration — exploitation trade-off

There is a trade-off between

Exploration: find new states or new actions which may lead to higher rewards;

Exploitation: perform those actions which have worked in the past (use the knowledge already acquired).

We need learning algorithms which do both.

The ϵ-greedy policy

A simple approach to balance exploring and exploiting is the *epsilon-greedy* policy:

- Most of the time you take the best action (based on current knowledge);
- occasionally, take a random decision.

The ϵ-greedy policy

A simple approach to balance exploring and exploiting is the *epsilon-greedy* policy:

$$\text{selected action} = \begin{cases} \text{believed best action}; & \text{with probability } 1-\epsilon \\ \text{random action}; & \text{with probability } \epsilon. \end{cases}$$

where ϵ is a small number between 0 and 1.

Decreasing ϵ over time

- As the agent learns and improves, it can be beneficial to decrease ϵ (less exploration) over time t.
- Theoretical reasons suggest a decrease $O(1/t)$.
- E.g.

$$\epsilon_t = \epsilon_0/t;\ \epsilon_0 \in (0,1)$$
$$\epsilon_t = \min(1, T_0/t);\ T_0 \text{ large}.$$

Application to immediate reward RL

- The agent can observe the current state s_t of the world at time t.
- It chooses an action a_t to perform.
- It receives rewards r_t at time t, which is a function (possibly stochastic) of the states and the actions.
- A “one-shot game”, it does not have to play more actions to receive the reward.

A simple example

A room full of slot machines.

- Cost of £1 to play.
- One machine pays off at positive rate (state law).
- The rest pay off at a negative rate.
- The positive machine changes hourly.

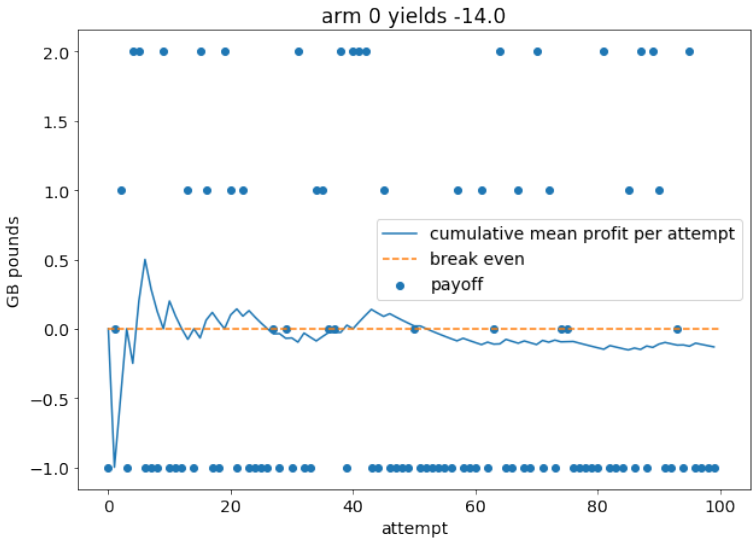
Task: Find the machine with the positive payoff, by playing different slot machines.

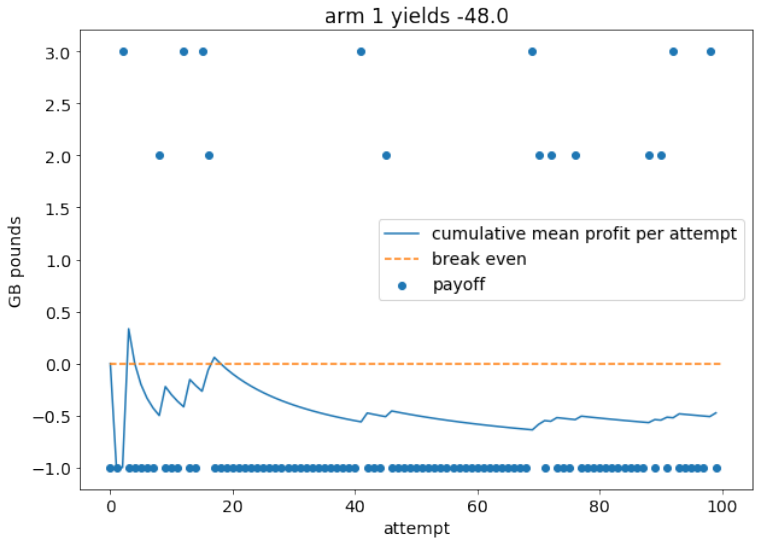
One approach

Play each machine 100 times to find the best machine. What is wrong with this?

A four slot-machine example

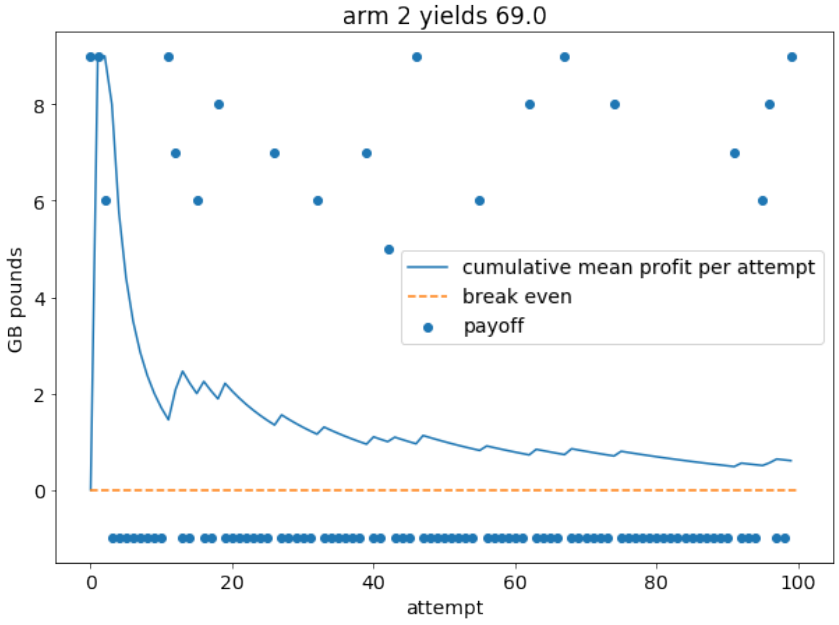
Properties of the machines

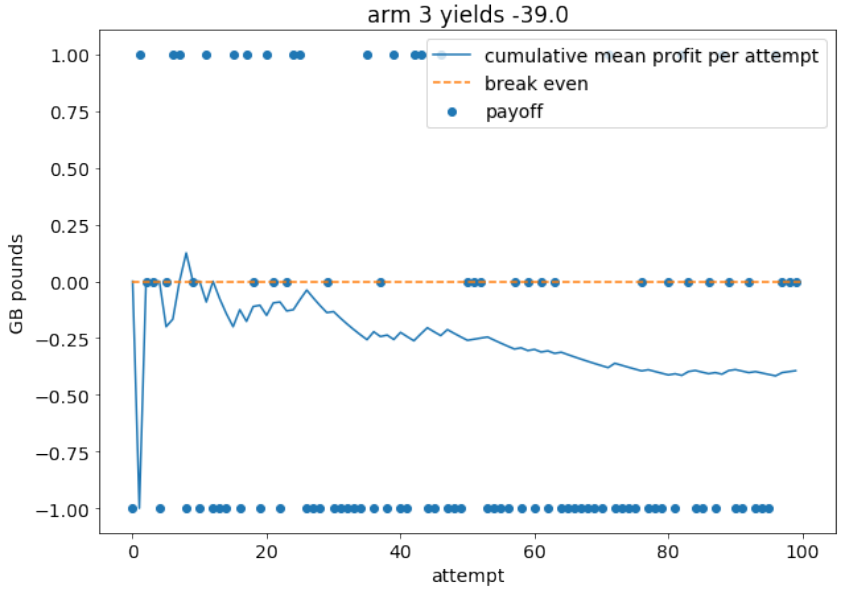




A four slot-machine example

Properties of the machines





Cost £32 to learn the positive machine this way.

Tabular RL approach

Maintain a table, $V(i)$, where

- $V(i)$ — approximation to the average payoff when slot machine i is played.
- The values of $V(\cdot)$ will be learned by playing various machines.

Choosing the machine to play

Use epsilon greedy

With probability $(1 - \epsilon)$: choose machine $a = \text{argmax}_i \, V(i)$

With probability ϵ: choose random machine $a =$ random .

How to update the value table

- Assumption: At time t, machine i was played and reward r_t received.
- Let α be a small, positive learning rate.

$$V[i] \leftarrow V[i] + \alpha(r_t - V[i]).$$

- Only the value associated with the machine which was played is updated.

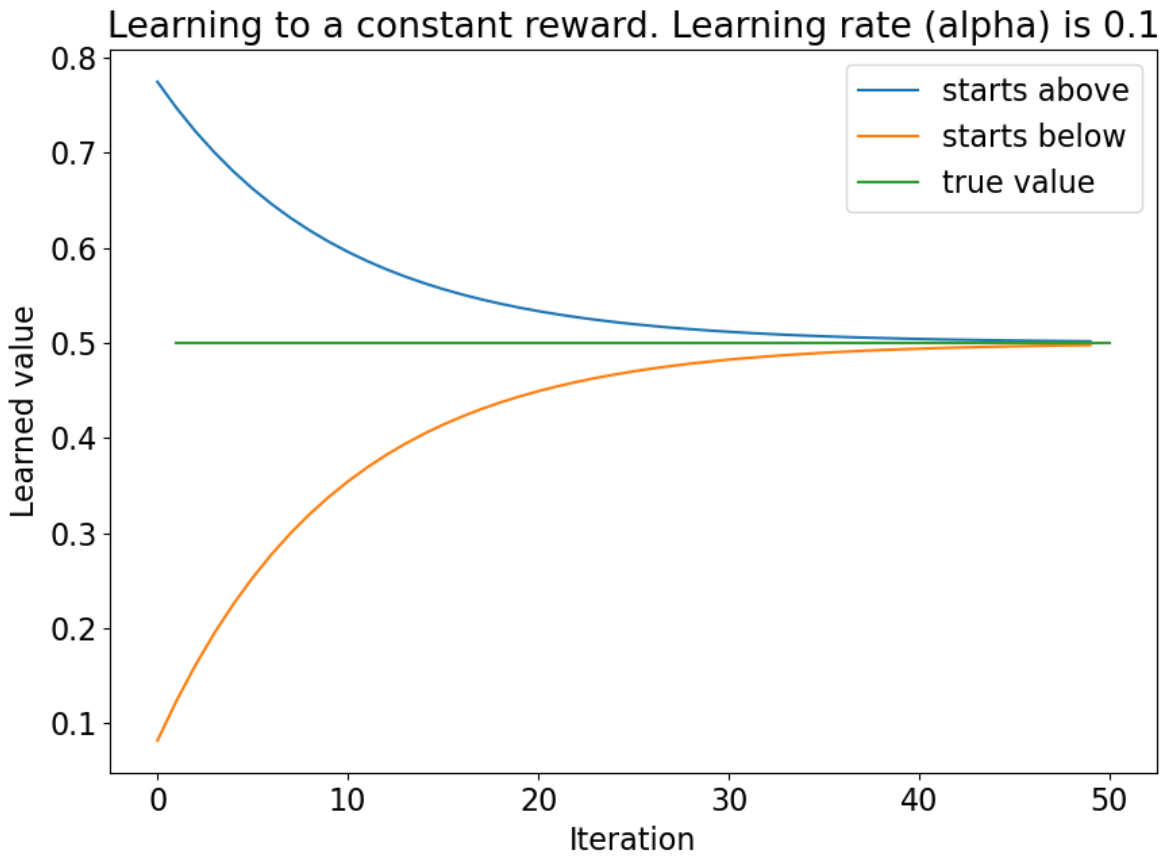
How to update the value table 2

- Assumption: At time t, machine i was played and reward r_t received.
- Let t_i be the count of the number of times machine i was played up to the current time.

$$V[i] \leftarrow V[i] + (r_t - V[i])/t_i.$$

- Equivalent to a time-dependent learning rate, $\alpha(t) = 1/t$.
- Maintains $V[i]$ as the mean of the rewards when i was played.

The learning dynamics



Exercise

problem

- *Let $x_1, \ldots, x_n, \ldots$ be a stream of data.*
- *Let M be a variable, which is initially $M = 0$. As each value of x_n comes in, M is updated via the following equation,*

$$M \leftarrow \frac{n-1}{n} M + \frac{x_n}{n}.$$

Show that after the nth point comes in, that

$$M = \frac{1}{n} \sum_{i=1}^{n} x_i,$$

The arithmetic mean of the data.

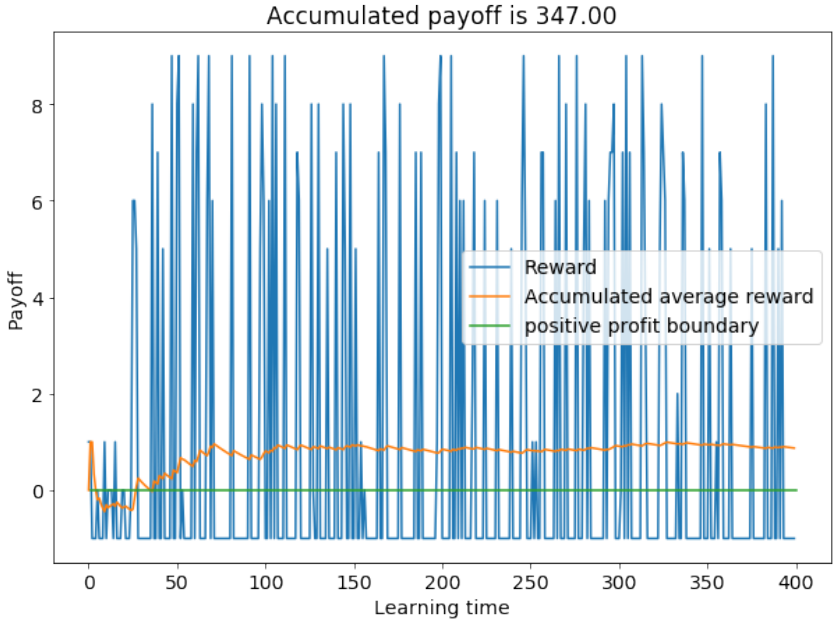
Time-dependent or constant learning rate?

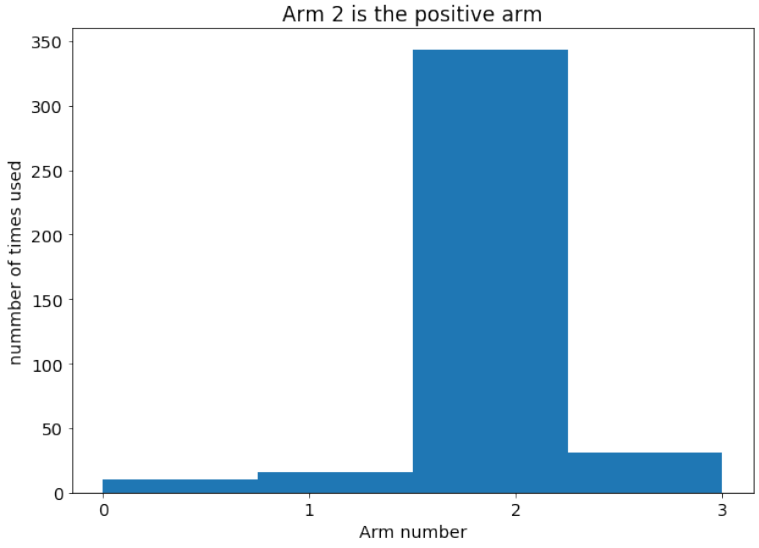
Constant learning rate: Use in a changing in environment. A decreasing learning rate loses ability to respond to changes.

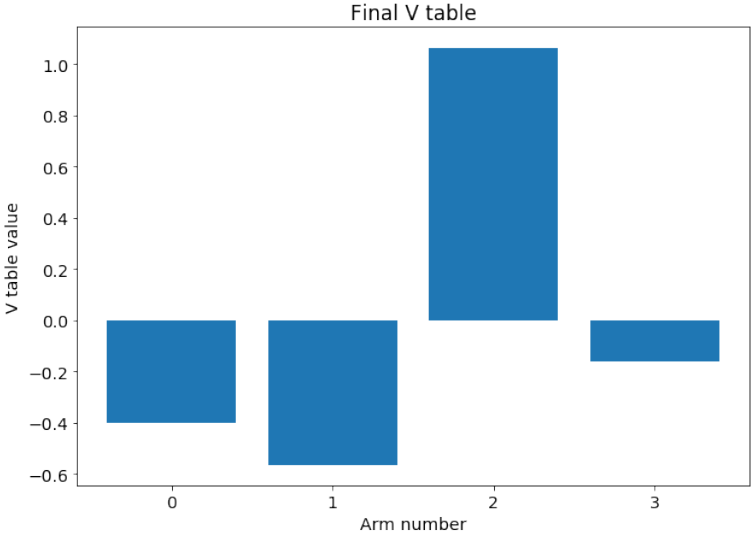
Decreasing learning rate: Use in an unchanging environment. Produces a more precise estimation of the average reward.

Of course, we often do not know whether the environment is changing or not.

Result of simple learning in simple example







Summary

- By exploring we find the best machine;
- By exploiting we get a good payoff.

This example had no state. Only actions.

The multi-arm bandit problem

This is an example of the so-called multi-arm bandit problem (MAB)

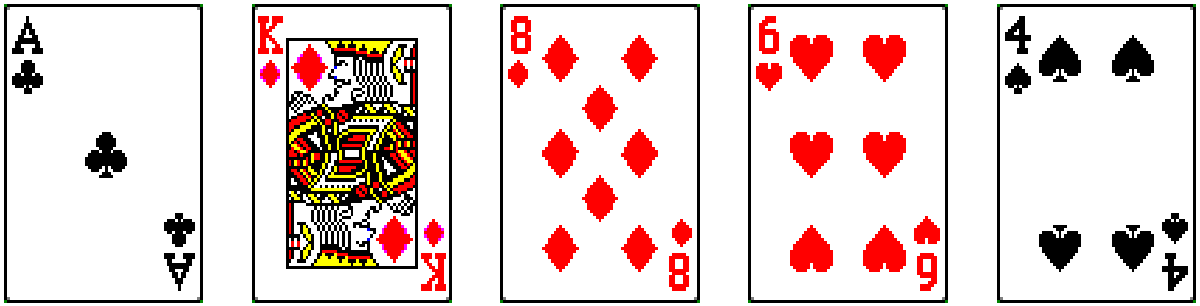
(See Chapter 2 of “Reinforcement Learning An Introduction 2nd edition”, 2017, Sutton and Barto. Link on Blackboard under Week 5 material)

Consider *video poker*

- a video slot machine (not real poker!)
- the player is dealt 5 cards,
- has one chance to change any or all of the dealt cards,
- goal is to make as good a poker hand as possible.
- Like a “fruit machine” found in pubs.

A stochastic immediate reward problem


THE ONLY WAY IS
EPIC
1000



Hand	Reward
Royal Flush	249
Straight Flush	49
Four of a kind	24
Full House	8
Flush	5
Straight	3
Three of a Kind	2
Two Pair	1
Jacks or Better	0
Nothing	-1
Theoretical Return	-0.0195

Return

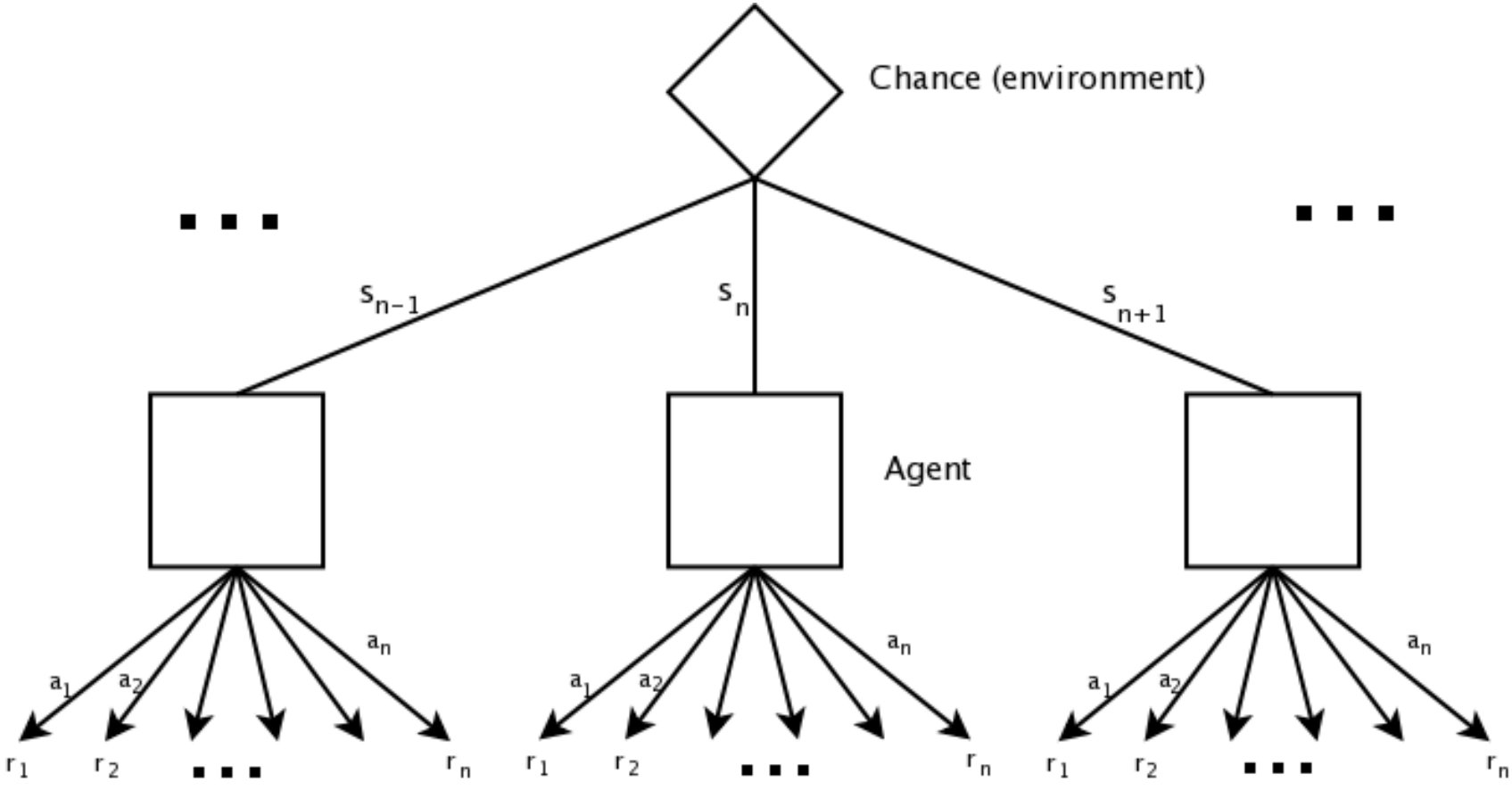


Figure: A ”game tree” for video poker. Environment (chance) chooses state; agent has to choose action to maximize rewards, which are stochastic.

Tabular learning

The agent is represented by a big table of:

State-action pairs:

- $Q(s_t, a_t)$ — expected reward for taking action a_t from state s_t at time t. Called Q-learning.[2]

State values:

- $V(s_t)$ — The expected reward of a state s_t assuming a policy for taking the action from that state. Called state-value learning.

[2] Not really Q-learning yet

Tabular learning for video poker

States: A representation of the 5 cards dealt

Actions: Those cards which are to be discarded and redrawn. 32 possible actions.

Value: The *expected* pay-off (i.e. long-term average).

Learning model: A big table $Q(s, a)$, expected reward for playing action a from state s over and over.

A learning algorithm

1. Select ϵ value. Possibly α value
2. Initialize $Q(s, a)$ for all s and a; $t \leftarrow 1$.
3. For each state-action pair t_{sa} counts the number of times (s, a) was visited.
4. Repeat
   - 4.1 Observe state s.
   - 4.2 Using epsilon greedy, choose either the best action according to the current $Q(s, a)$ or a random action.
   - 4.3 Observe reward r_t
   - 4.4 Update $Q(s, a)$ only for the action a taken from state s.

$$Q(s, a) \leftarrow Q(s, a) + \frac{1}{t_{sa}} [r_t - Q(s, a)] \tag{1}$$

$$\text{or} \tag{2}$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r_t - Q(s, a)] \tag{3}$$

5. until there is no time left

Question: How many states; how many actions?

How many actions: $2^5 = 32$.

How many states: $\binom{52}{5} = 2,598,960$

- Each state is equally likely.

The need for Q-function approximation

Problem: There are a lot of states. It will take too long to learn.

Solutions:

- A (hand-coded) representation of poker hands which uses a single representation for equivalent hands.
- Use a supervised learning function approximation

  Input: is a representation of the hand and the action.

  Desired output: is the expected reward. A regression problem.

  - E.g. using MLP regressor, SVM regressor, linear or polynomial regression, etc.

Back to video poker

Summary

1. Learning in games requires reinforcement learning.
2. Reinforcement learning requires a balance of exploitation and exploitation.
3. Learning in games involves ‘learning via‘self-play”.
4. Tabular learning uses a table to represent the value of states and actions.
5. Too many states or too many actions requires function approximation.