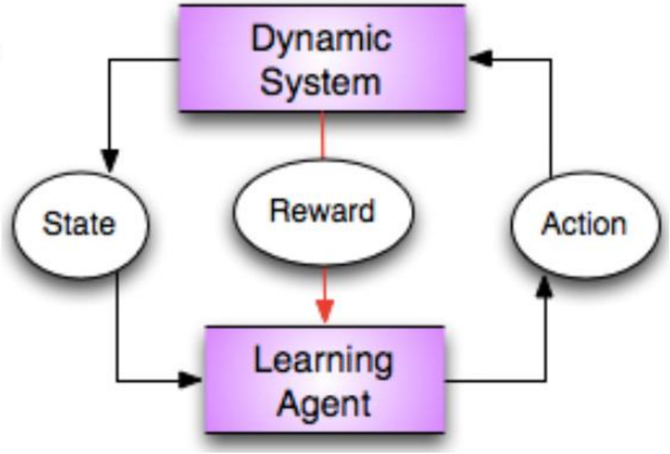
Reinforcement Learning

From basic concepts to deep Q-networks

Reinforcement learning

1. Learning agent tries a **sequence of actions** (a_t).

2. Observes outcomes (state s_{t+1}, rewards r_t) of those actions.

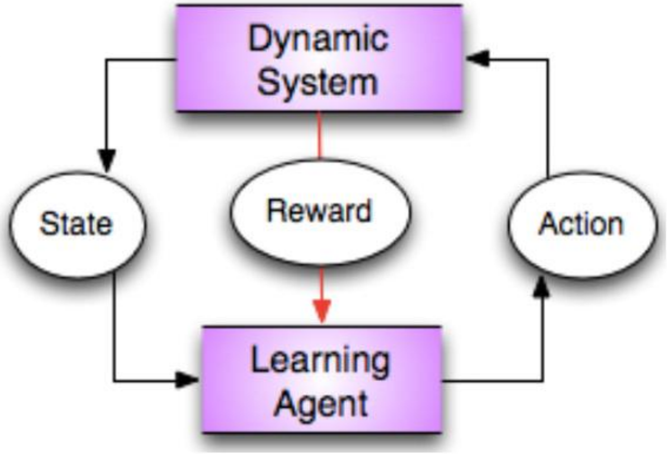


Reinforcement learning

1. Learning agent tries a **sequence of actions** (a_t).

2. Observes outcomes (state s_{t+1}, rewards r_t) of those actions.

3. Statistically estimates relationship between action choice and outcomes, $Pr(s_t|s_{t-1},a_t)$.

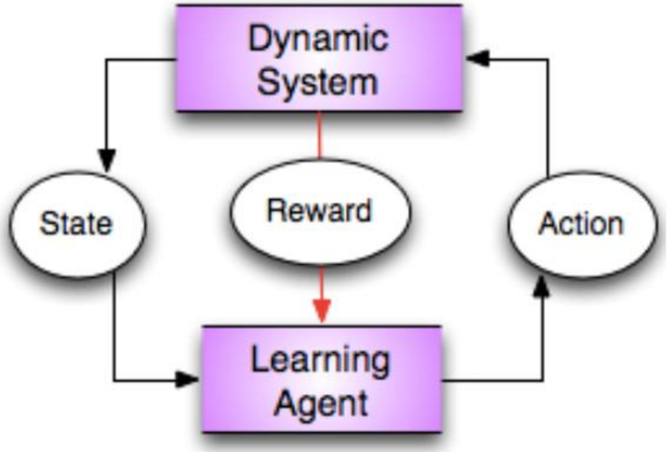


Reinforcement learning

1. Learning agent tries a **sequence of actions** (a_t).

2. Observes outcomes (state s_{t+1}, rewards r_t) of those actions.

3. Statistically estimates relationship between action choice and outcomes, $Pr(s_t|s_{t-1},a_t)$.



After some time… learns action selection policy, $\pi(s)$, that optimizes selected outcomes.

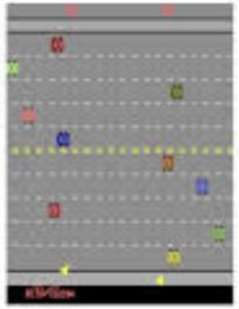
$$argmax_\pi E_\pi [r_0 + r_1 + \ldots + r_T \mid s_0]$$

http://en.wikipedia.org/wiki/Animal_training

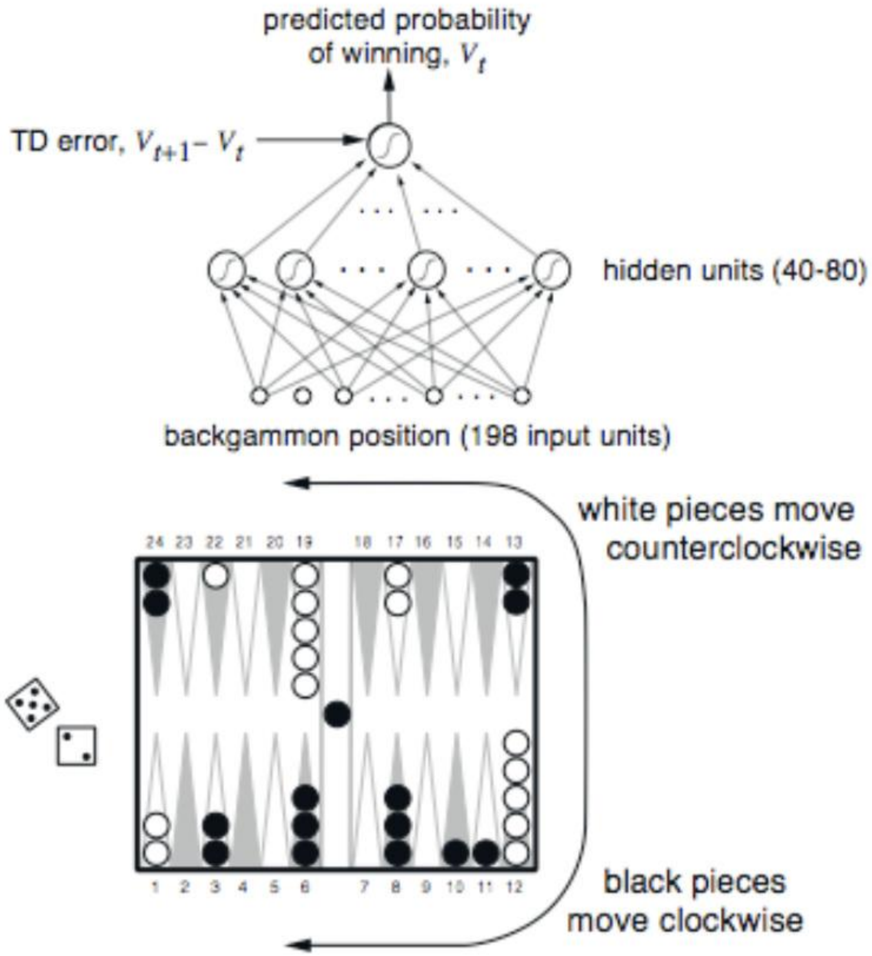
[Bellman, 1957; Sutton, 1988; Sutton&Barto, 1998.]

Many applications of RL

- Robotics
- Medicine
- Advertisement
- Resource management
- Game playing …



RL system circa 1990's: TD-Gammon



Reward function:

+100 if win

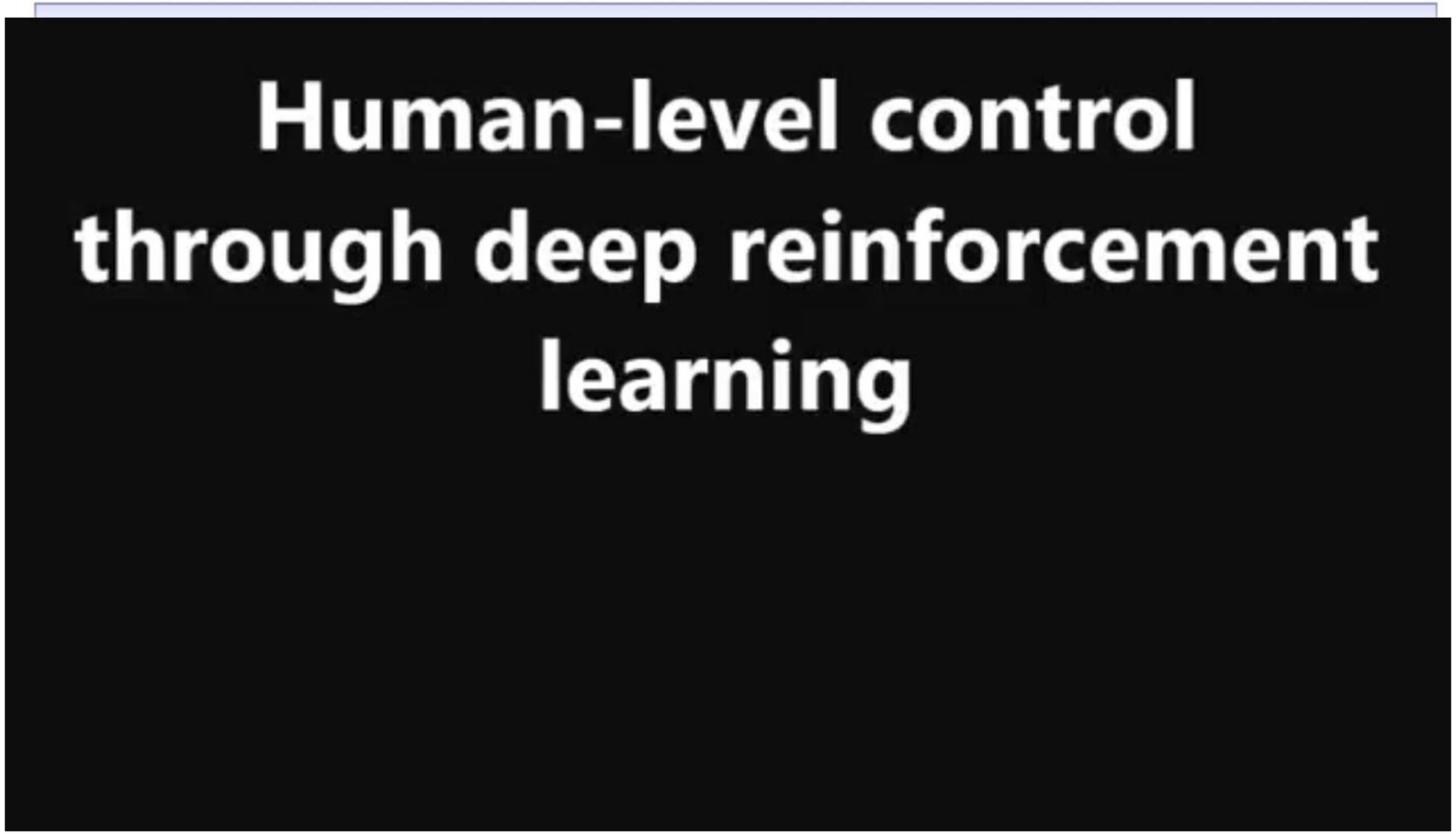
- 100 if lose

0 for all other states

Trained by playing $1.5x10^6$ million games against itself.

Enough to beat the best human player.

Human-level Atari agent (2015)



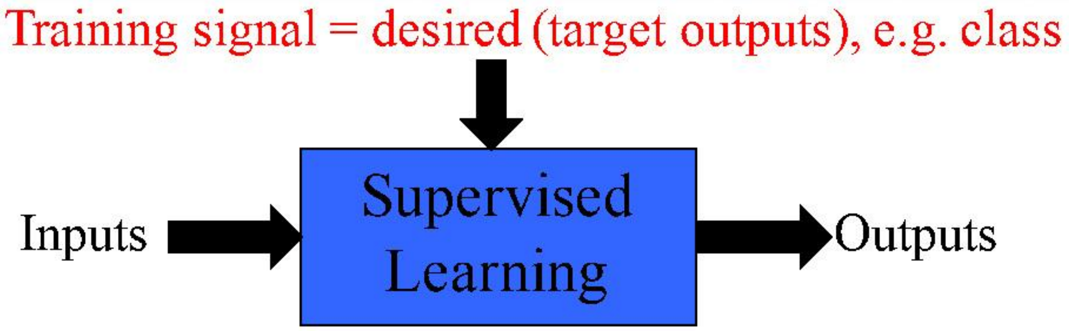
Human-level control through deep reinforcement learning

DeepMind’s AlphaGo (2016)

When to use RL?

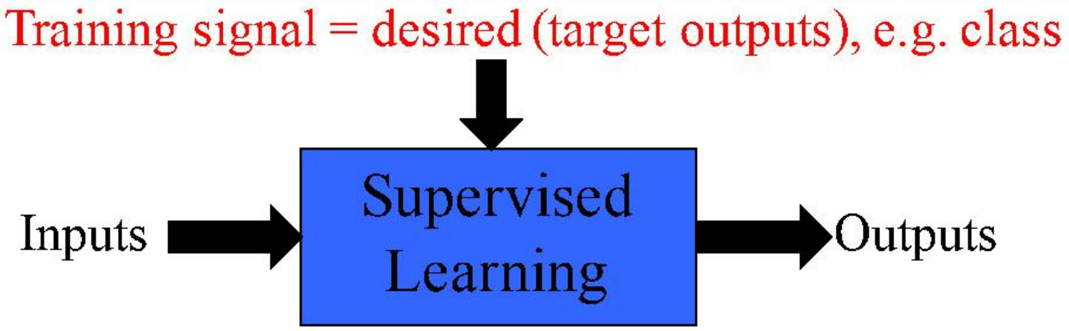
- Data in the form of trajectories.
- Need to make a sequence of (related) decisions.
- Observe (partial, noisy) feedback to state or choice of actions.
- There is a gain when optimizing action choice over a portion of the trajectory.

RL vs supervised learning



Training signal = desired (target outputs), e.g. class

RL vs supervised learning

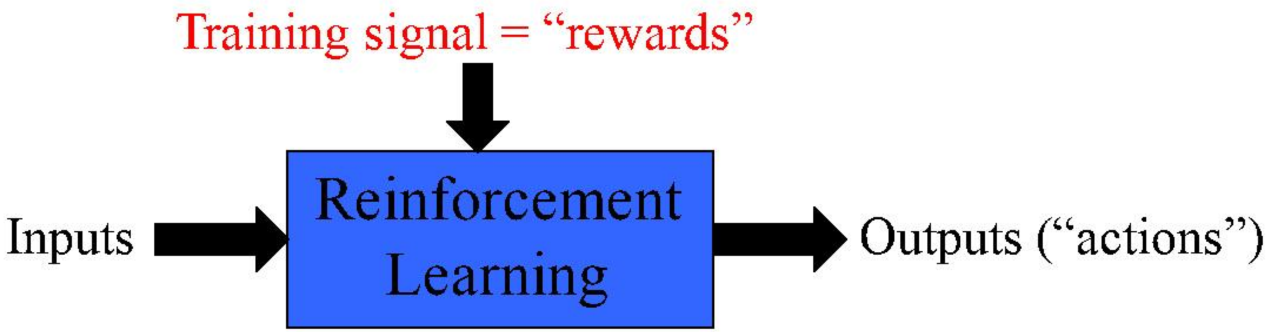


Training signal = desired (target outputs), e.g. class

Inputs

Supervised Learning

Outputs



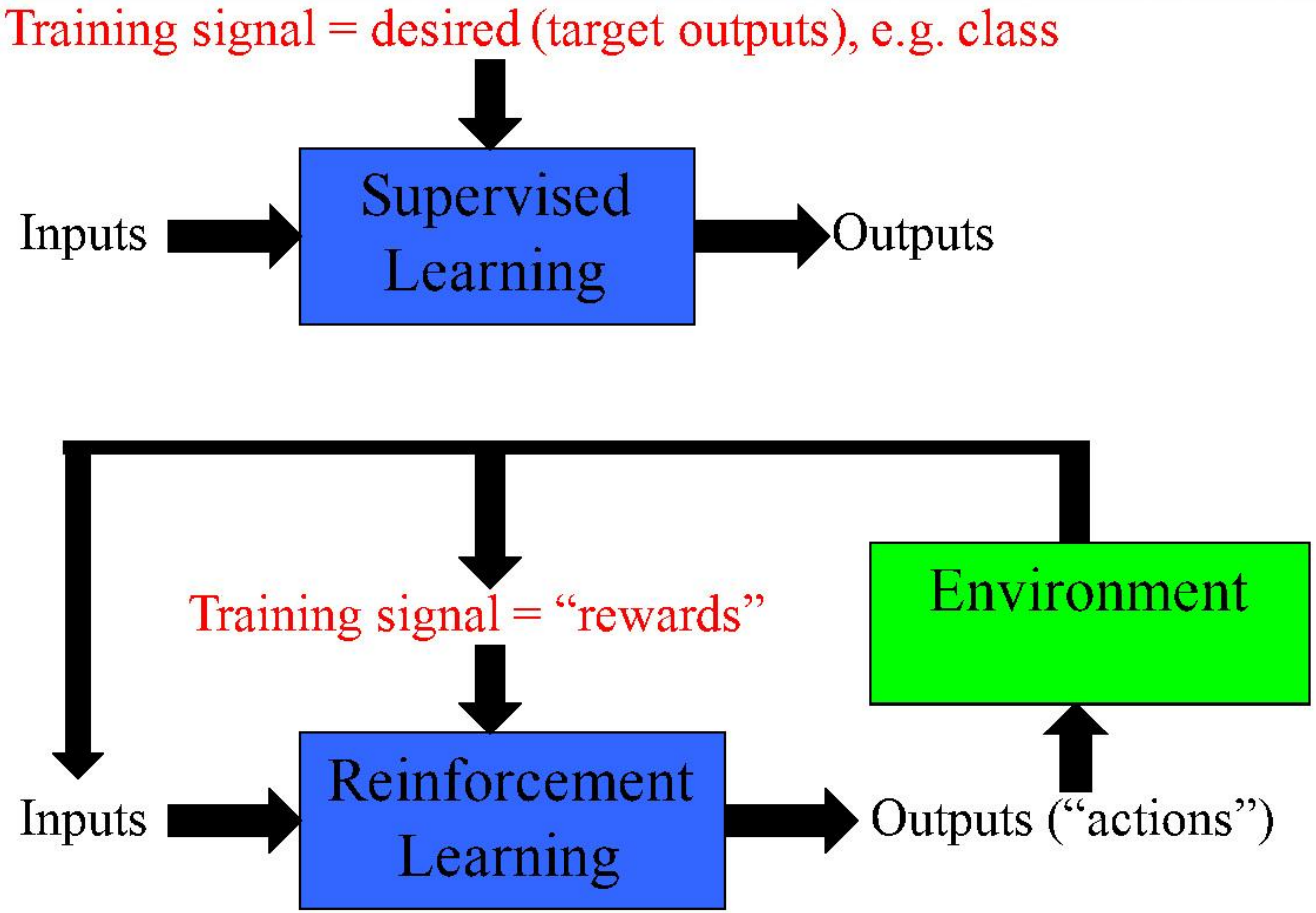
Training signal = “rewards”

Inputs

Reinforcement Learning

Outputs (“actions”)

RL vs supervised learning



Training signal = desired (target outputs), e.g. class

Inputs

Supervised Learning

Outputs

Training signal = "rewards"

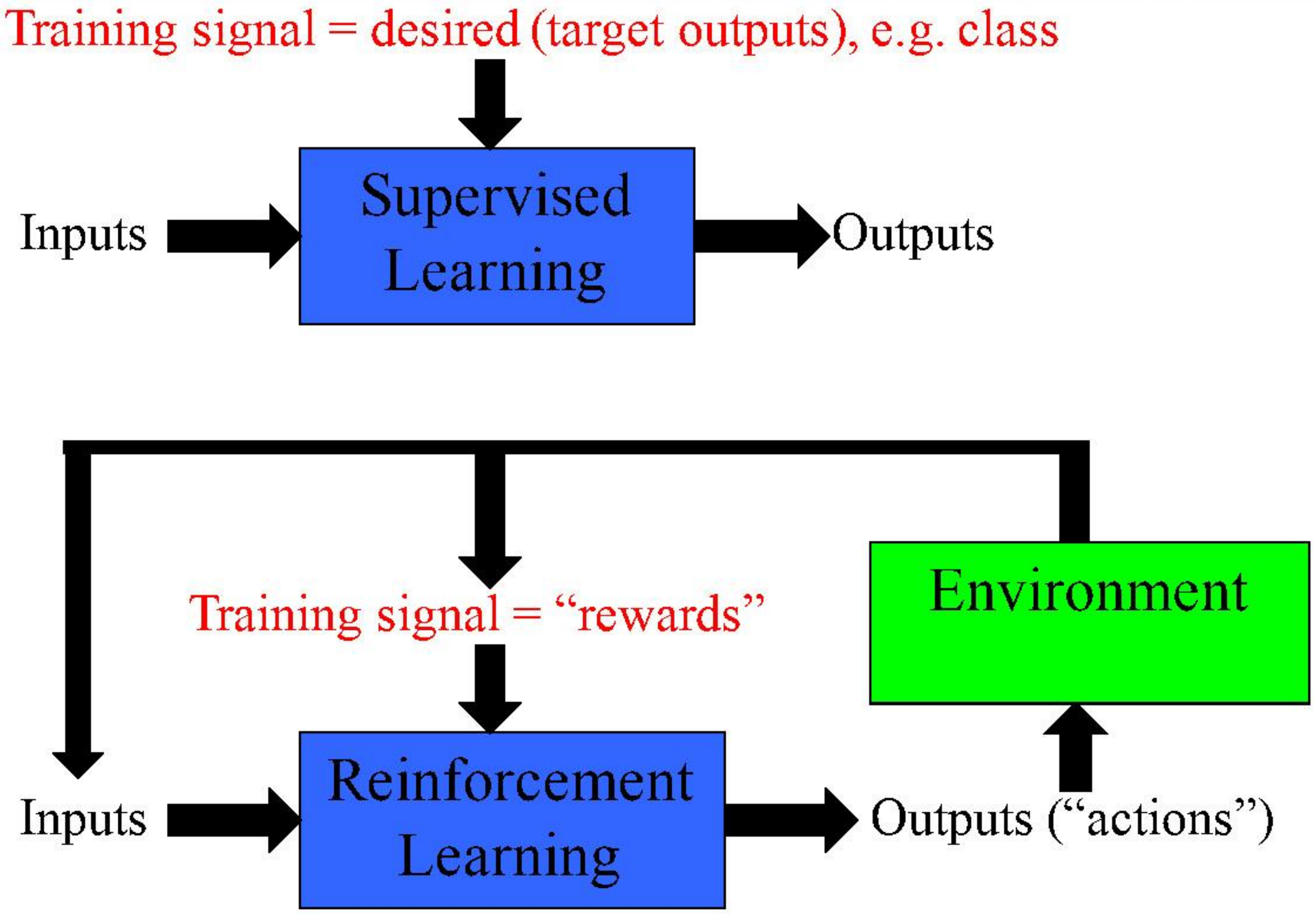
Environment

Inputs

Reinforcement Learning

Outputs ("actions")

RL vs supervised learning



Challenges:

Jointly learning AND planning from correlated samples.

Data distribution changes with action choice.

Need access to the environment.

Markov Decision Process (MDP)

Defined by:

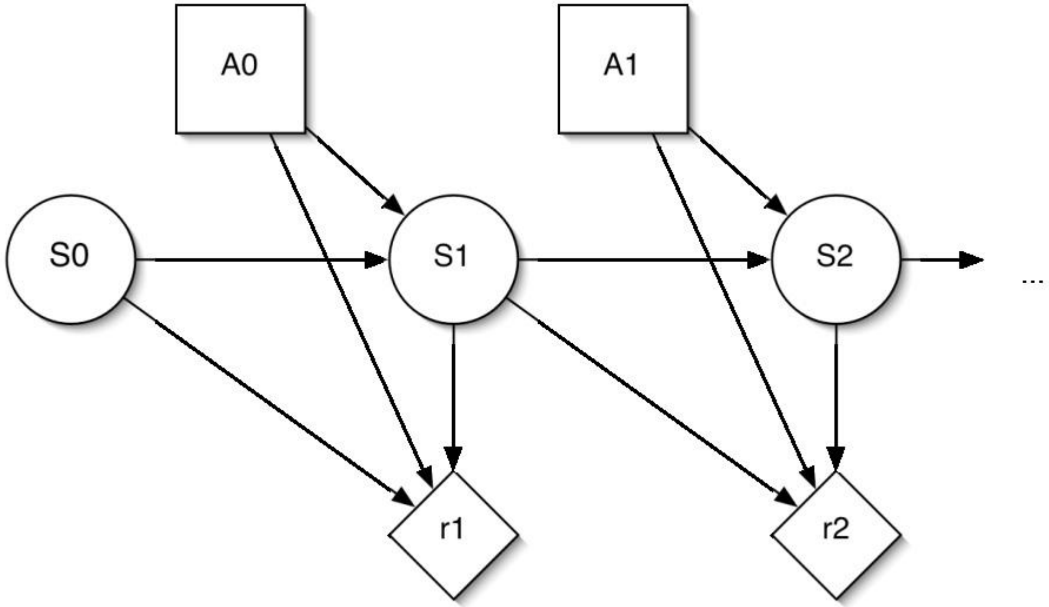
S: Set of states

A: Set of actions

$Pr(s_t|s_{t-1},a_t)$: Probabilistic effects

r_t : Reward function

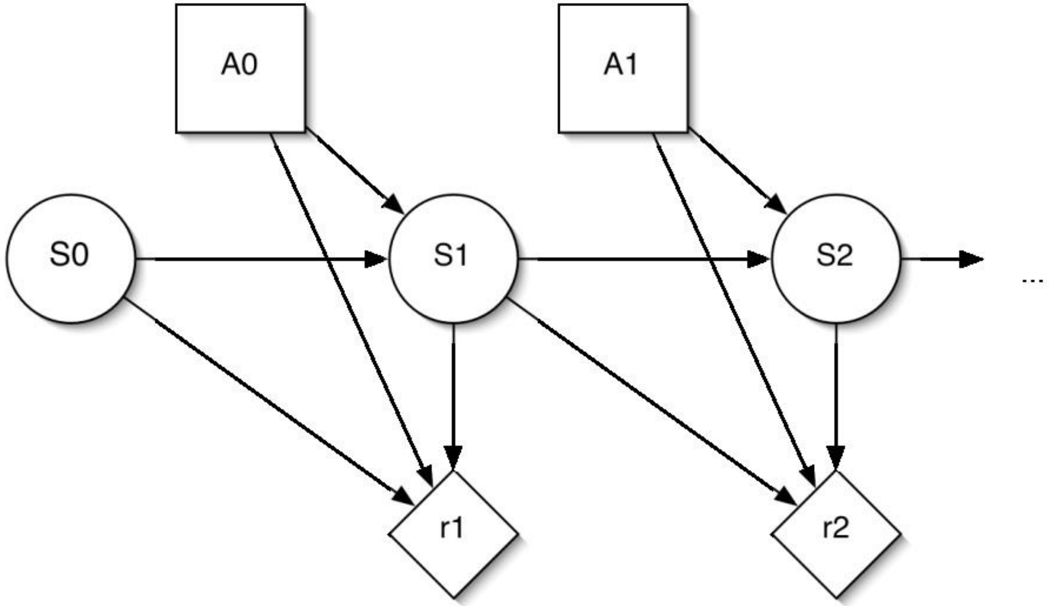
μ_t : Initial state distribution



The **Markov** property

The distribution over future states **depends only on the present state**, not on any previous events.

$$Pr(s_t \mid s_{t-1}, \ldots, s_0) = Pr(s_t \mid s_{t-1})$$



Maximizing utility

- Define: U_t , the utility for a trajectory, starting from step t.

- Episodic tasks (e.g. games, trips through a maze, etc.)

$$U_t = r_t + r_{t+1} + r_{t+2} + \dots + r_T$$

- Continuing tasks (e.g. tasks which may go on forever)

$$U_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \dots = \sum_{k=0:\infty} \gamma^k r_{t+k}$$

The discount factor, γ

- Discount factor, $\gamma \in [0, 1)$ (usually close to 1).

- Two interpretations:
  - At each time step, there is a $1-\gamma$ chance that the agent dies, and does not receive rewards afterwards.
  - Inflation rate: receiving an amount of money tomorrow, is worth less than today by a factor of γ.

The policy

A policy defines the action-selection strategy at every state:

$$\pi(s,a) = P(a_t=a \mid s_t=s)$$

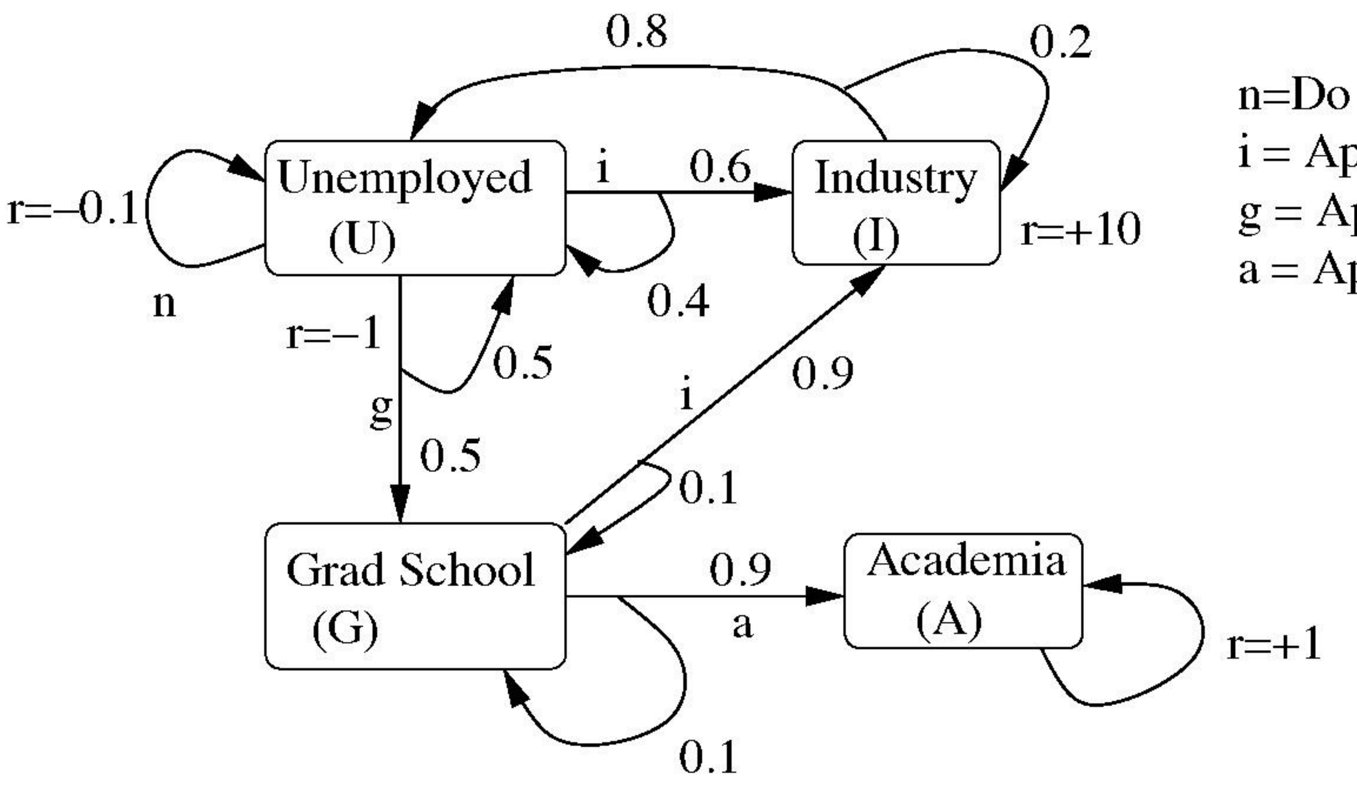
(Can be stochastic as above, or deterministic, $S \rightarrow A$.)

Goal: **Find the policy that maximizes expected total reward.**

(But there are many policies!)

$$\mathbf{argmax}_{\pi}\, E_{\pi} [\, r_0 + r_1 + \ldots + r_T \mid s_0 \,]$$

Example: Career Options



n=Do Nothing
i = Apply to industry
g = Apply to grad school
a = Apply to academia

What is the best policy?

Value functions

- If we want to find a policy that maximizes the expected return, it is useful to estimate the expected return.
- Then we can search through the space of policies for a good policy.
- **Value functions** represent the expected return, for every state, given a certain policy.

$$V^{\pi}(s) = \boldsymbol{E}_{\pi}\, [\boldsymbol{r_t} + \boldsymbol{r_{t+t}} + \ldots + \boldsymbol{r_T} \mid \boldsymbol{s_t} = \boldsymbol{s}\ ]$$

The value of a policy

$$V^{\pi}(s) = E_{\pi}\,[r_t + r_{t+1} + \ldots + r_T \mid s_t = s\,]$$

$$V^{\pi}(s) = E_{\pi}\,[r_t \mid s_t = s\,] \;+\; E_{\pi}\,[\,r_{t+1} + \ldots + r_T \mid s_t = s\,]$$

$$V^{\pi}(s) = \underbrace{\textstyle\sum_{a \in A} \pi(s,a) r(s,a)}_{\textit{Immediate reward}} + \underbrace{E_{\pi}\,[\,r_{t+1} + \ldots + r_T \mid s_t = s\,]}_{\textit{Future expected sum of rewards}}$$

The value of a policy

$$V^{\pi}(s) = E_{\pi}\,[r_t + r_{t+t} + \ldots + r_T \mid s_t = s\,]$$

$$V^{\pi}(s) = E_{\pi}\,[r_t \mid s_t = s\,] \; + \; E_{\pi}\,[\, r_{t+1} + \ldots + r_T \mid s_t = s\,]$$

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) r(s,a) \; + E_{\pi}\,[\, r_{t+1} + \ldots + r_T \mid s_t = s\,]$$

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) r(s,a) + \underbrace{\sum_{a \in A} \pi(s,a) \sum_{s' \in S} T(s,a,s')}_{\text{Expectation over 1-step transition}} E_{\pi}\,[r_{t+1} + \ldots + r_T \mid s_{t+1} = s'\,]$$

The value of a policy

$$V^{\pi}(s) = E_{\pi}\,[r_t + r_{t+t} + \ldots + r_T \mid s_t = s\,]$$

$$V^{\pi}(s) = E_{\pi}\,[r_t \mid s_t = s\,] + E_{\pi}\,[\,r_{t+1} + \ldots + r_T \mid s_t = s\,]$$

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) r(s,a) + E_{\pi}\,[\,r_{t+1} + \ldots + r_T \mid s_t = s\,]$$

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) r(s,a) + \sum_{a \in A} \pi(s,a) \sum_{s' \in S} T(s,a,s')\underbrace{E_{\pi}\,[r_{t+1} + \ldots + r_T \mid s_{t+1} = s'\,]}_{}$$

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) r(s,a) + \sum_{a \in A} \pi(s,a) \sum_{s' \in S} T(s,a,s')\ \underbrace{V^{\pi}(s')}_{\textit{By definition}}$$

This is a **dynamic programming** algorithm.

The value of a policy

State value function (for a **fixed** policy):

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) \, (\underbrace{r(s,a)}_{\textit{Immediate}} + \gamma \underbrace{\sum_{s' \in S} T(s,a,s') V^{\pi}(s') \,)}_{\textit{Future expected sum of rewards}}$$

State-action value function:

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^{\pi}(s',a')$$

These are (two forms of) **Bellman's equation**.

The value of a policy

State value function:

$$V^\pi(s) = \sum_{a\in A} \pi(s,a) \left(r(s,a) + \gamma \sum_{s'\in S} T(s,a,s')V^\pi(s') \right)$$

When S is a **finite set of states**, this is a **system of linear equations** (one per state) with a unique solution V^π.

Bellman's equation in matrix form: $V^\pi = R^\pi + \gamma\, T^\pi\, V^\pi$

Which can solved exactly: $V^\pi = (I - \gamma\, T^\pi)^{-1} R^\pi$

Iterative Policy Evaluation

Main idea: turn Bellman equations into update rules.

1. Start with some initial guess $V_0(s), \forall s$. (Can be 0, or $r(s,\cdot)$.)

Iterative Policy Evaluation

Main idea: turn Bellman equations into update rules.

1. Start with some initial guess $V_0(s), \forall s.$ (Can be 0, or $r(s,\cdot)$.)

2. During every iteration k, update the value function for all states:

$$V_{k+1}(s) \leftarrow \Big(R(s, \pi(s)) + \gamma \textstyle\sum_{s' \in S} T(s, \pi(s), s')V_k(s') \Big)$$

Iterative Policy Evaluation

Main idea: turn Bellman equations into update rules.

1. Start with some initial guess $V_0(s), \forall s.$ (Can be 0, or $r(s,\cdot)$.)

2. During every iteration k, update the value function for all states:

$$V_{k+1}(s) \leftarrow \Big(R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s')V_k(s') \Big)$$

3. Stop when the maximum changes between two iterations is smaller than a desired threshold (the values stop changing.)

Convergence of Iterative Policy Evaluation

- Consider the absolute error in our estimate $V_{k+1}(s)$:

$$
\begin{aligned}
|V_{k+1}(s) - V^{\pi}(s)| &= \left| \sum_a \pi(s,a)(R(s,a) + \gamma \sum_{s'} T(s,a,s')V_k(s')) \right. \\
&\quad \left. - \sum_a \pi(s,a)(R(s,a) + \gamma \sum_{s'} T(s,a,s')V^{\pi}(s')) \right| \\
&= \gamma \left| \sum_a \pi(s,a) \sum_{s'} T(s,a,s')(V_k(s') - V^{\pi}(s')) \right| \\
&\leq \gamma \sum_a \pi(s,a) \sum_{s'} T(s,a,s')|V_k(s') - V^{\pi}(s')|
\end{aligned}
$$

- As long as $\gamma<1$, the error **contracts** and eventually goes to 0.

Optimal policies and optimal value functions

- The **optimal value function** V^* is defined as the best value that can be achieved at any state:

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

- Any policy that achieves the optimal value function is called an **optimal policy**, denoted π^*.

- There exists a unique optimal **value function** *(Bellman, 1957)*.
- The optimal policy is not necessarily unique.

Optimal policies and optimal value functions

- If we know V^* (and R, T, γ), then we can compute π^* easily:

$$\pi^*(s) \quad = \quad argmax_{a \in A} \left(r(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s') \right)$$

- If we know π^* (and R, T, γ), then we can compute V^* easily:

$$V^*(s) \quad = \sum_{a \in A} \pi^*(s,a) \left(r(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s') \right)$$

$$V^*(s) \quad = r(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s),s')V^*(s')$$

Finding a good policy: **Policy Iteration**

- Start with an initial policy π_0 (e.g. random)
- Repeat:
  - Compute V^π, using policy evaluation.
  - Compute a new policy π' that is greedy with respect to V^π
- Terminate when $\pi = \pi'$

Finding a good policy: **Value iteration**

Main idea: Turn the Bellman optimality equation into an iterative update rule (same as done in policy evaluation):

1. Start with an arbitrary initial approximation $V_0(s)$

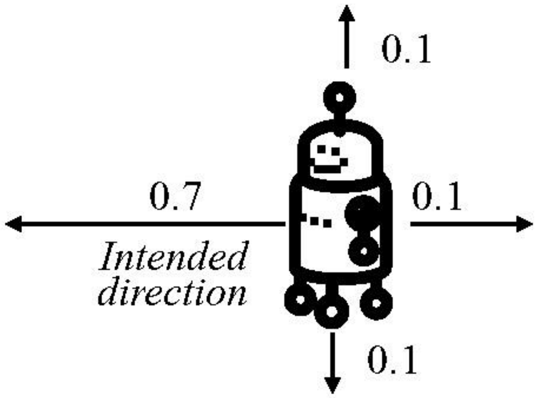
2. On each iteration, update the value function estimate:
   $V_k(s) = \max_{a \in A} (R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_{k-1}(s'))$

3. Stop when max value change between iterations is below threshold.

The algorithm converges (in the limit) to the true V^*.

A 4x3 gridworld example

- 11 discrete states, 4 motion actions (N, S, E, W) in each state.
- Transitions are mildly **stochastic**.
- Reward is +1 in top right state, -10 in state directly below, -0 elsewhere.
- Episode terminates when the agent reaches +1 or -10 state.
- Discount factor γ = *0.99*.



S			+1
	■		-10

Value Iteration (1)

0	0	0	+1
0	■	0	-10
0	0	0	0

Value Iteration (2)

0	0	0.69	+1
0	■	-0.99	-10
0	0	0	-0.99

Bellman residual: $|V_2(s) - V_1(s)| = 0.99$

Value Iteration (5)

0.48	0.70	0.76	+1
0.23	■	-0.55	-10
0	-0.20	-0.23	-1.40

Bellman residual: $|V_5(s) - V_4(s)| = 0.23$

Value Iteration (20)

0.78	0.80	0.81	+1
0.77		-0.44	-10
0.75	0.69	0.37	-0.92

Bellman residual: $|V_5(s) - V_4(s)| = 0.008$

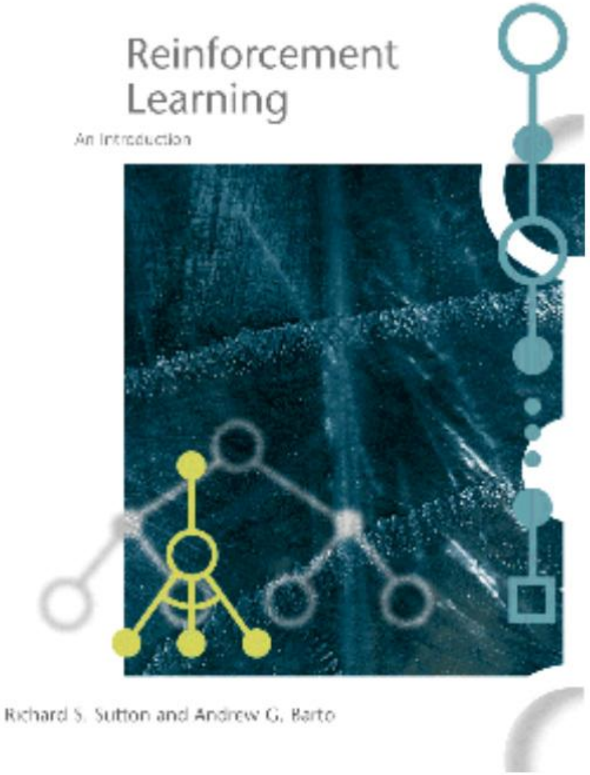
Asynchronous value iteration

- Instead of updating all states on every iteration, focus on *important states*.
  - E.g., board positions that occur on every game, rather than just once in 100 games.

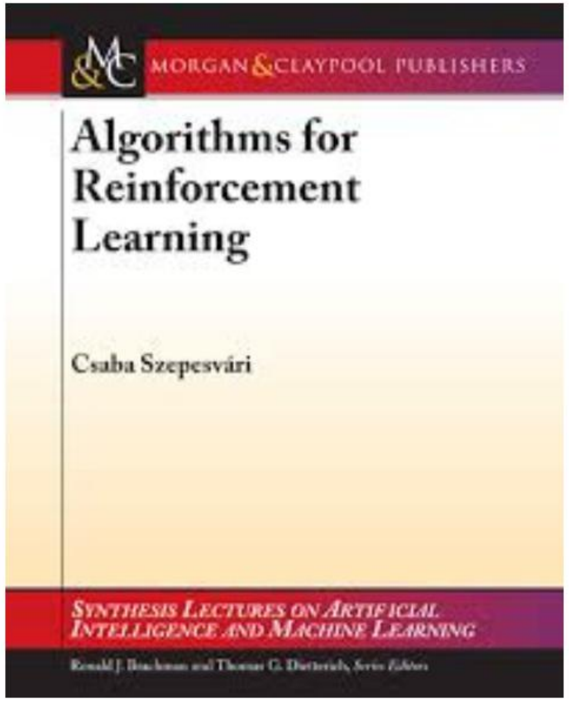
- Asynchronous dynamic programming algorithm:
  - Generate trajectories through the MDP.
  - Update states whenever they appear on such a trajectory.

- Focuses the updates on states that are actually possible.

Want to know more?



Sutton & Barto, 1998


M&C MORGAN & CLAYPOOL PUBLISHERS
Algorithms for Reinforcement Learning
Csaba Szepesvári
SYNTHESIS LECTURES ON ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING
Ronald J. Brachman and Thomas G. Dietterich, Series Editors

Szepesvari, 2010

Key challenges in RL

- Designing the problem domain
  - State representation
  - Action choice
  - Cost/reward signal
- Acquiring data for training
  - Exploration / exploitation
  - High cost actions
  - Time-delayed cost/reward signal
- Function approximation
- Validation / confidence measures

The RL lingo

- Episodic / Continuing task
- Tabular / Function approximation
- Batch / Online
- On-policy / Off-policy
- Exploration / Exploitation
- Model-based / Model-free
- Policy optimization / Value function methods

Episodic / Continuing

- Let U_t be the utility for a trajectory, starting from step t.

- **Episodic tasks:** e.g. games, trips through a maze, etc.

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

** Some subtleties about value iteration, e.g. need to keep $V_t(s)$, $t=0..T$*

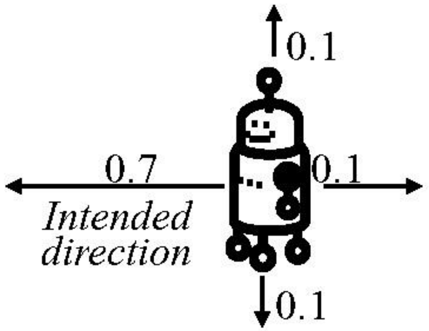
- **Continuing tasks:** e.g. tasks which may go on forever

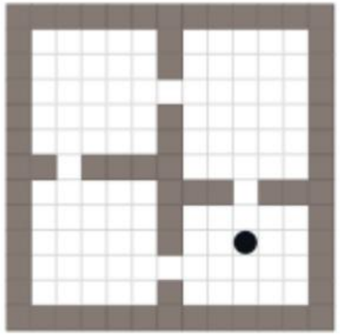
$$U_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \ldots = \sum_{k=0:\infty} \gamma^k r_{t+k}$$

** Need to use a discount factor. Interesting new ideas on how to set.*

Tabular / Function approximation

- **Tabular**: Can store in memory a <u>list of the states</u> and their value.





** Can prove many more **theoretical properties** in this case, about convergence, sample complexity.*

- **Function approximation**: Too many states, continuous state spaces.

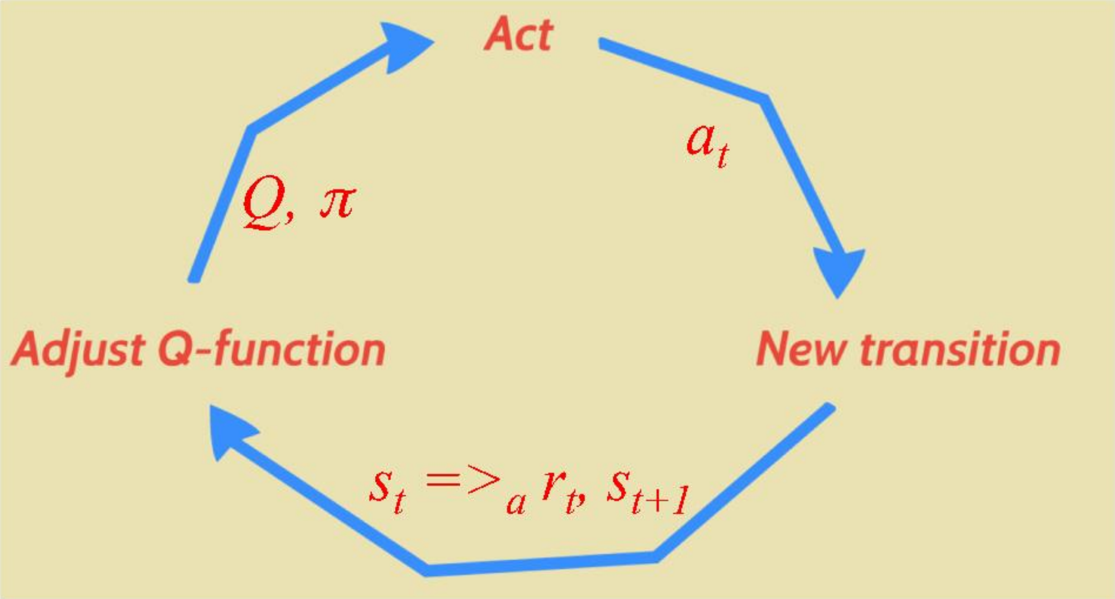
Batch / Online

- **Learning from a batch** (more on this later).

  ** Get all data at once, collected from a fixed (unknown?) policy.*

- **Learning online from repeated interactions**:

  ** Can vary the collection policy. Non-stationary data distribution.*



Online learning

- **Monte-Carlo** value estimate: Use the empirical return, *$U(s_t)$* as a target estimate for the actual value function:

$$V(s_t) \leftarrow V(s_t) + \alpha\big(U(s_t) - V(s_t)\big)$$

** Not a Bellman equation. More like a gradient equation.*

  - Here α is the learning rate (a parameter).
  - Need to wait until the end of the trajectory to compute *$U(s_t)$*.

- **Temporal difference** learning: Use an estimate of the return.

$$V(s_t) \leftarrow V(s_t) + \alpha\big(r_t + \gamma V(s_{t+1}) - V(s_t)\big)$$

Temporal-Difference with function approx.

- **Tabular TD(0)**:

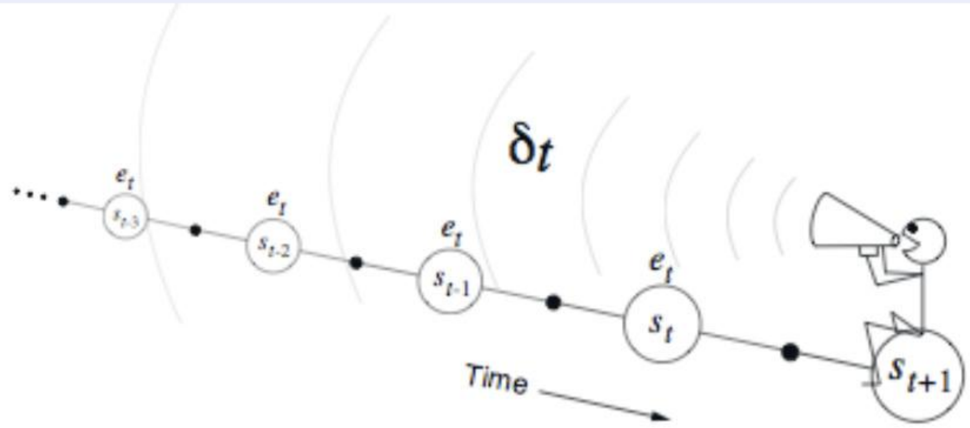
$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right) \forall t = 0, 1, 2, \ldots$$

- **Gradient-descent TD(0)**:

$$\theta \leftarrow \theta + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right) \nabla_\theta V(s_t), \forall t = 0, 1, 2, \ldots$$

Use the **TD-error**, instead of the “supervised” error.

Online learning with eligibility: TD(λ)

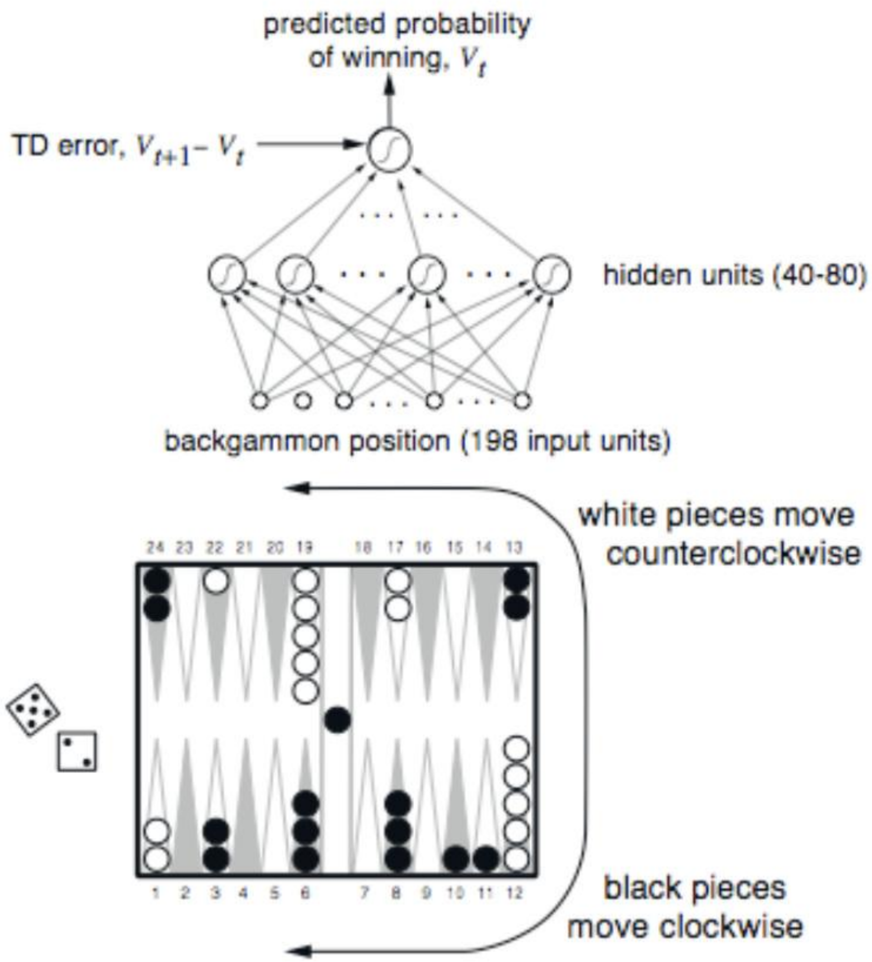


- On every time step *t*, we compute the TD error:

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

- Update all states $V(s_t) \leftarrow V(s_t) + \alpha\, \delta_t\, e(s_t)$
- Decrease eligibility $e(s_t) \leftarrow \gamma\lambda\, e(s)$, where $\lambda \in [0,\ 1]$ is a parameter.

TD-Gammon (Tesauro, 1992)



Reward function:

- +100 if win
- - 100 if lose
- 0 for all other states

Trained by playing 1.5x10^6 million games against itself.

Enough to beat the best human player.

The RL lingo

- Episodic / Continuing task
- Tabular / Function approximation
- Batch / Online
- **On-policy / Off-policy**
- Exploration / Exploitation
- Model-based / Model-free
- Policy optimization / Value function methods

On-policy / Off-policy

- Policy induces a distribution over the states (data).
  - Data distribution **changes** every time you change the policy!

On-policy / Off-policy

- Policy induces a distribution over the states (data).
  - Data distribution **changes** every time you change the policy!

- Evaluating several policies with the same batch:
  - Need very big batch!
  - Need policy to adequately cover all *(s,a)* pairs.

On-policy / Off-policy

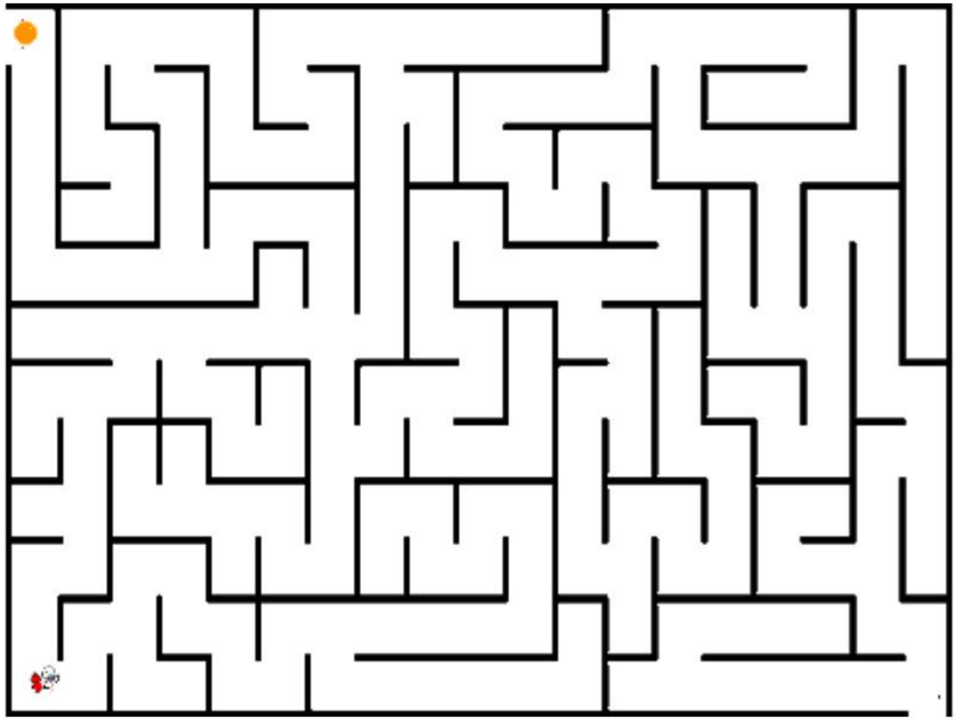
- Policy induces a distribution over the states (data).
  - Data distribution **changes** every time you change the policy!

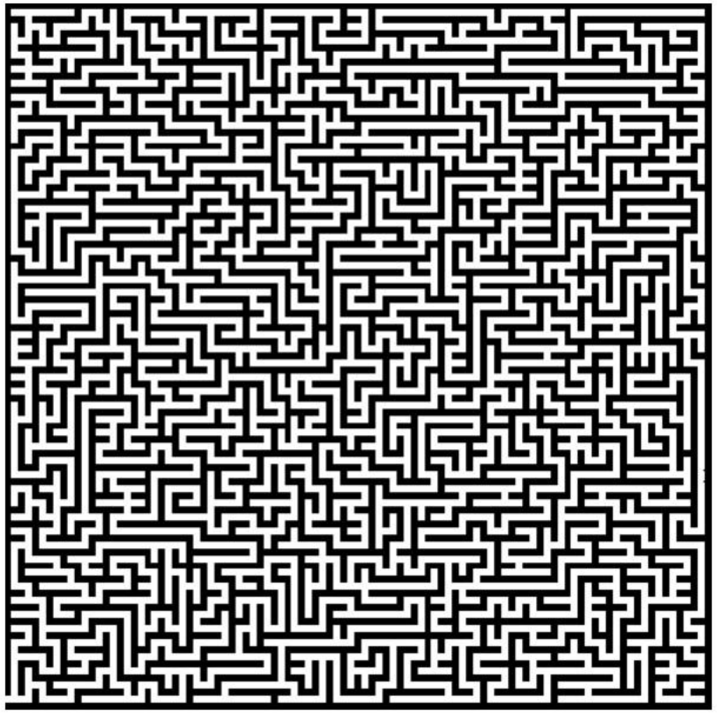
- Evaluating several policies with the same batch:
  - Need very big batch!
  - Need policy to adequately cover all *(s,a)* pairs.

- Use importance sampling to reweigh data samples to compute unbiased estimates of a new policy.

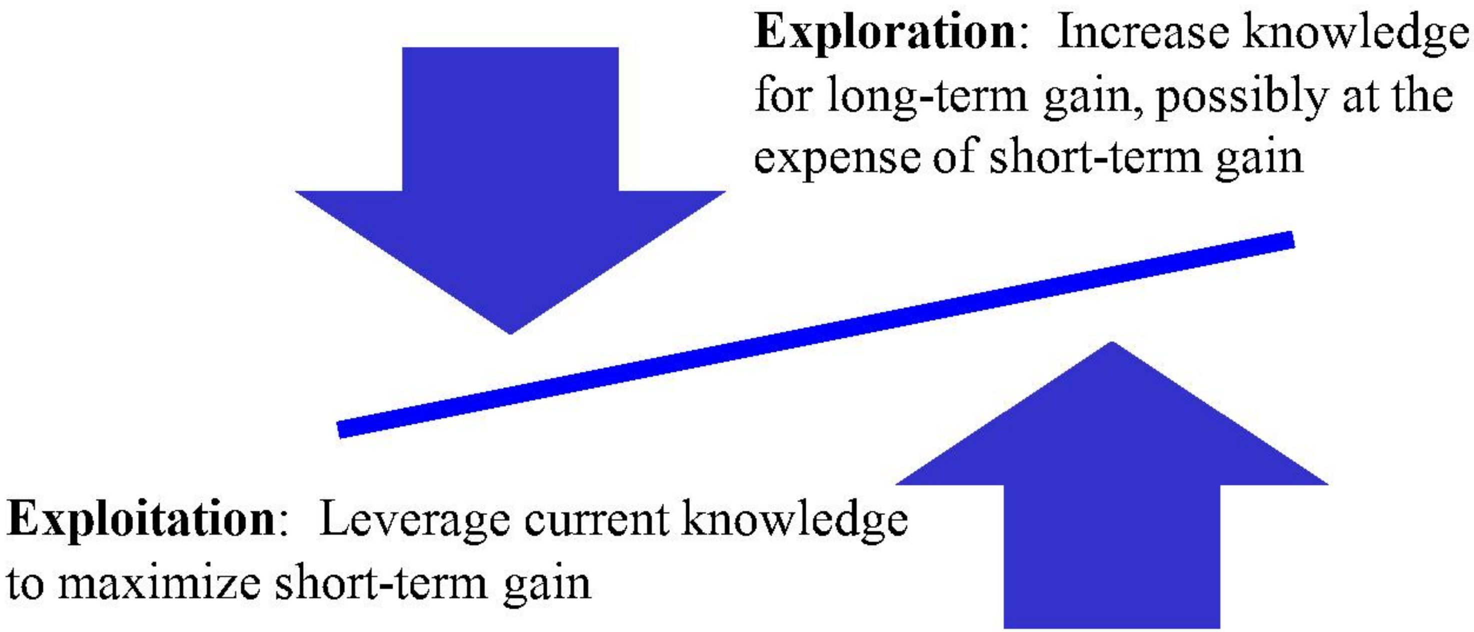
$$\rho_t = \frac{\pi(s_t, a_t)}{b(s_t, a_t)}$$

Exploration / Exploitation





Exploration / Exploitation



Exploration: Increase knowledge for long-term gain, possibly at the expense of short-term gain

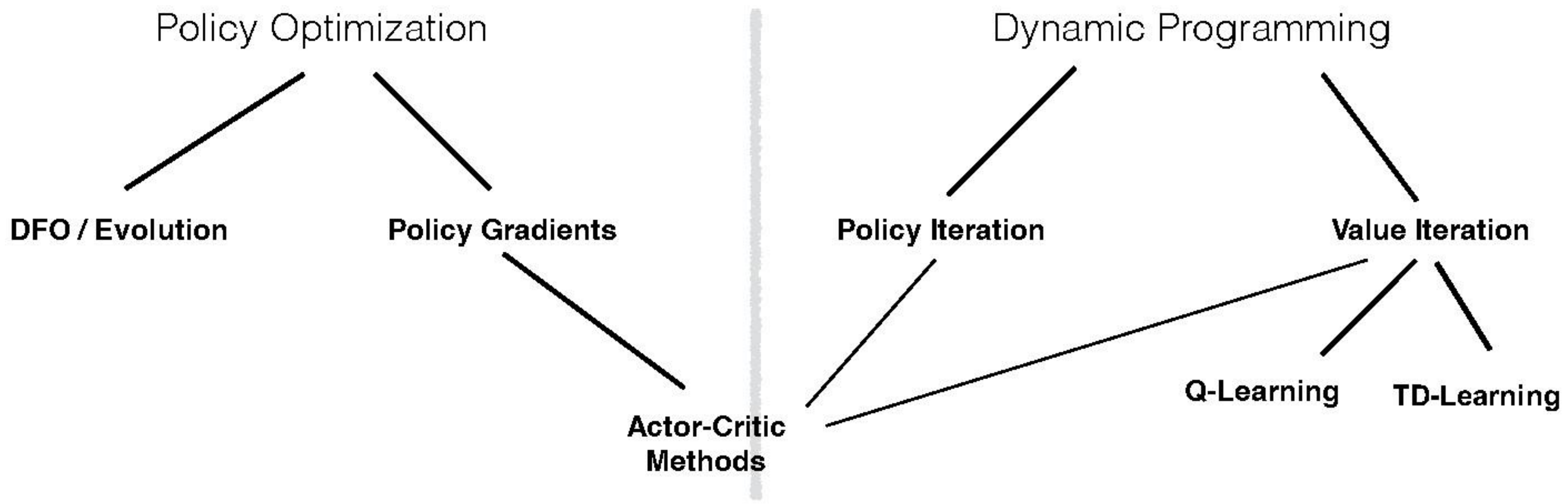
Exploitation: Leverage current knowledge to maximize short-term gain

Model-based vs Model-free RL

- **Option #1**: Collect large amounts of observed trajectories. Learn an approximate model of the dynamics (e.g. with supervised learning). Pretend the model is correct and apply value iteration.

- **Option #2**: Use data to directly learn the value function or optimal policy.

Policy Optimization / Value Function



Policy Optimization

DFO / Evolution

Policy Gradients

Actor-Critic Methods

Dynamic Programming

Policy Iteration

Value Iteration

Q-Learning

TD-Learning

The RL lingo – done!

- Episodic / Continuing task
- Tabular / Function approximation
- Batch / Online
- On-policy / Off-policy
- Exploration / Exploitation
- Model-based / Model-free
- Policy optimization / Value function methods

Action Selection Policies

As mentioned above, there are three common policies used for action selection. The aim of these policies is to balance the trade-off between exploitation and exploration, by not always exploiting what has been learnt so far.

ε-greedy - most of the time the action with the highest estimated reward is chosen, called the greediest action. Every once in a while, say with a small probability ε, an action is selected at random. The action is selected uniformly, independent of the action-value estimates. This method ensures that if enough trials are done, each action will be tried an infinite number of times, thus ensuring optimal actions are discovered.

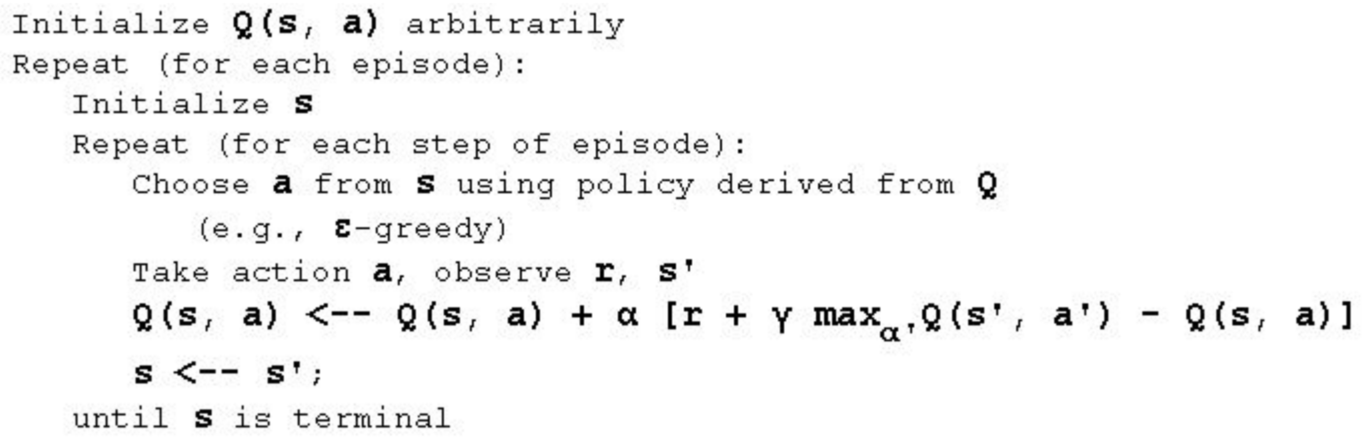
ε-soft - very similar to ε-greedy. The best action is selected with probability 1 - ε and the rest of the time a random action is chosen uniformly.

softmax - one drawback of ε-greedy and ε-soft is that they select random actions uniformly. The worst possible action is just as likely to be selected as the second best. Softmax remedies this by assigning a rank or weight to each of the actions, according to their action-value estimate. A random action is selected with regards to the weight associated with each action, meaning the worst actions are unlikely to be chosen. This is a good approach to take where the worst actions are very unfavourable.

It is not clear which of these policies produces the best results overall. The nature of the task will have some bearing on how well each policy influences learning. If the problem we are trying to solve is of a game playing nature, against a human opponent, human factors may also be influential.

Q-Learning

Q-Learning is an Off-Policy algorithm for Temporal Difference learning. It can be proven that given sufficient training under any ε-soft policy, the algorithm converges with probability 1 to a close approximation of the action-value function for an arbitrary target policy. Q-Learning learns the optimal policy even when actions are selected according to a more exploratory or even random policy. The procedural form of the algorithm is:



```
Initialize Q(s, a) arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q
            (e.g., $\varepsilon$-greedy)
        Take action a, observe r, s'
        $Q(s, a)$ <-- $Q(s, a) + \alpha [r + \gamma \max_{\alpha'} Q(s', a') - Q(s, a)]$
        s <-- s';
    until s is terminal
```

The parameters used in the Q-value update process are:

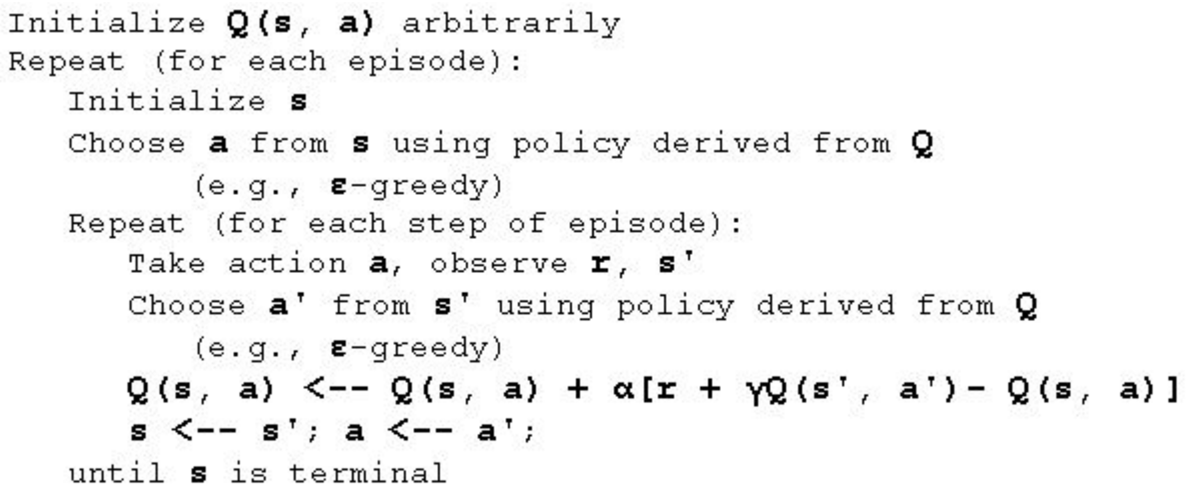
α - the learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value such as 0.9 means that learning can occur quickly.
γ - discount factor, also set between 0 and 1. This models the fact that future rewards are worth less than immediate rewards. Mathematically, the discount factor needs to be set less than 0 for the algorithm to converge.
$\max_{\alpha}$- the maximum reward that is attainable in the state following the current one. i.e the reward for taking the optimal action thereafter.

This procedural approach can be translated into plain english steps as follows:

1. Initialize the Q-values table, **Q(s, a)**.
2. Observe the current state, **s**.
3. Choose an action, **a**, for that state based on one of the action selection policies explained on the previous slide (ε-soft, ε-greedy or softmax).
4. Take the action, and observe the reward, **r**, as well as the new state, **s'**.
5. Update the Q-value for the state using the observed reward and the maximum reward possible for the next state. The updating is done according to the formulla and parameters described above.
6. Set the state to the new state, and repeat the process until a terminal state is reached.

Sarsa

The Sarsa algorithm is an On-Policy algorithm for TD-Learning. The major difference between it and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. The name Sarsa actually comes from the fact that the updates are done using the quintuple **Q(s, a, r, s', a').** Where: **s, a** are the original state and action, **r** is the reward observed in the following state and **s', a'** are the new state-action pair. The procedural form of Sarsa algorithm is comparable to that of Q-Learning:



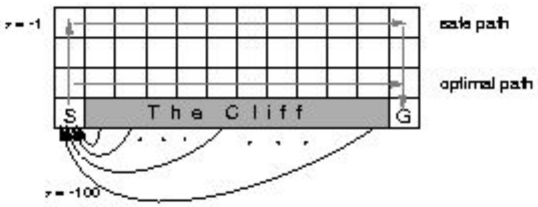
```
Initialize Q(s, a) arbitrarily
Repeat (for each episode):
    Initialize s
    Choose a from s using policy derived from Q
            (e.g., ε-greedy)
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q
            (e.g., ε-greedy)
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
        s <-- s'; a <-- a';
    until s is terminal
```

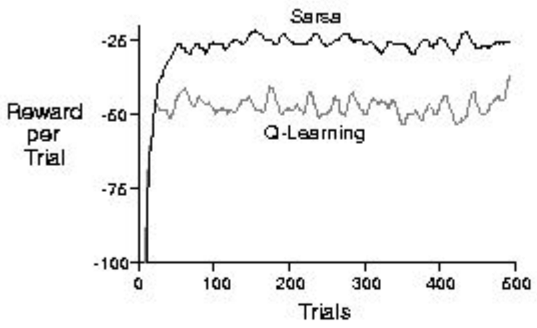
As you can see, there are two action selection steps needed, for determining the next state-action pair along with the first. The parameters γ and α have the same meaning as they do in Q-Learning.

Q-Learning vs SARSA

The reason that Q-learning is off-policy is that it updates its Q-values using the Q-value of the next state s′ and the greedy action a′. In other words, it estimates the return (total discounted future reward) for state-action pairs assuming a greedy policy were followed despite the fact that it's not necessary following a greedy policy.

The reason that SARSA is on-policy is that it updates its Q-values using the Q-value of the next state s′ and the current policy's action a″. It estimates the return for state-action pairs assuming the current policy continues to be followed.





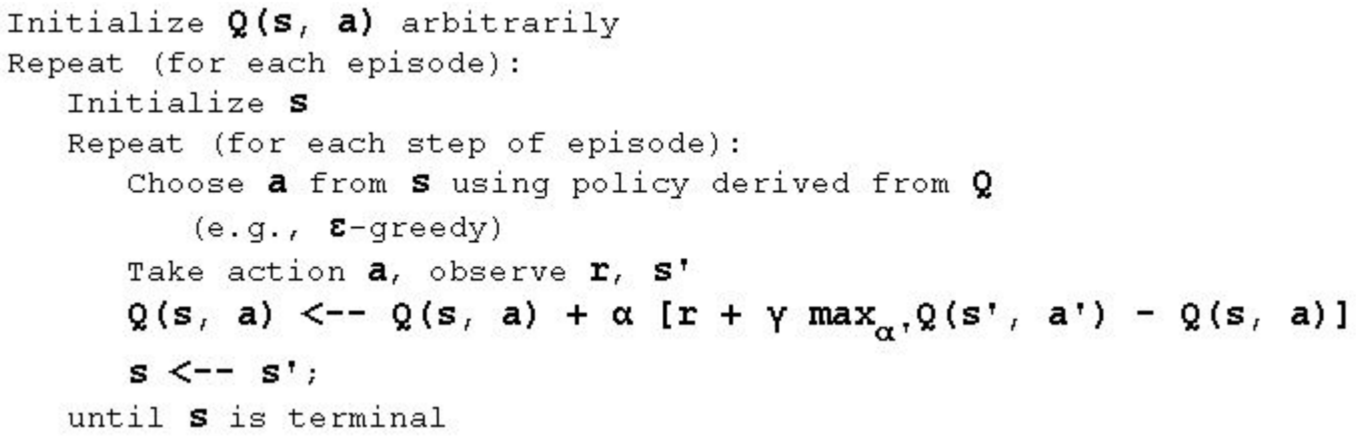
The world consists of a small grid. The goal-state of the world is the square marked **G** on the lower right-hand corner, and the start is the **S** square in the lower left-hand corner. There is a reward of negative 100 associated with moving off the cliff and negative 1 when in the top row of the world. Q-Learning correctly learns the optimal path along the edge of the cliff, but falls off every now and then due to the ε-greedy action selection. Sarsa learns the safe path, along the top row of the grid because it takes the action selection method into account when learning. Because Sarsa learns the safe path, it actually receives a higher average reward per trial than Q-Learning even though it does not walk the optimal path.

Incremental Average Computation

The terms of the average are arranged in a way to have both A(n+1) and A(n).

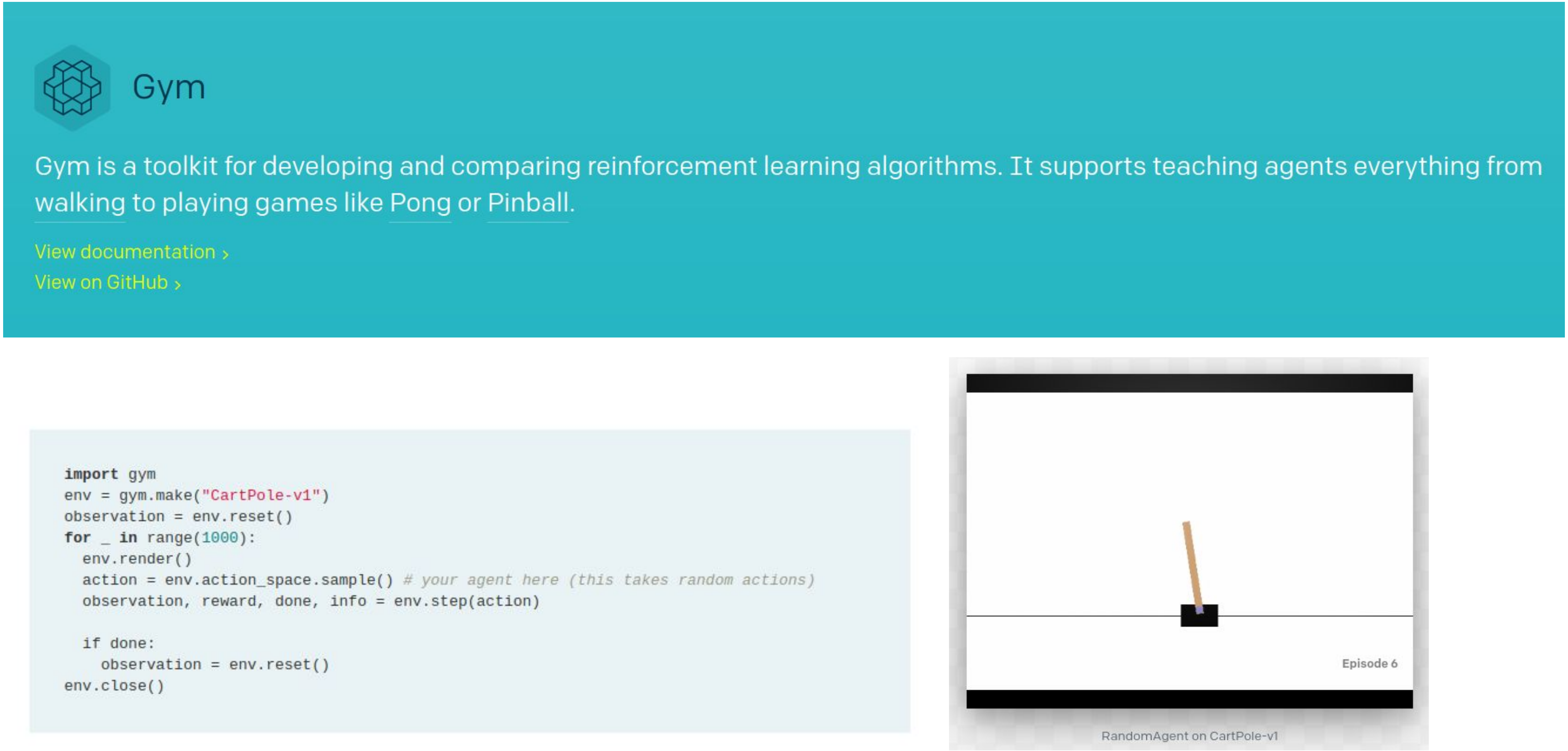
Notice that 1/(n+1) represents the term alpha in the State-Value and Action-Value functions.

$$\begin{aligned}
A_{n+1} &= \frac{\sum_{i=1}^{n+1} v_i}{n+1} \\
&= \frac{\sum_{i=1}^{n} v_i + v_{n+1}}{n+1} \\
&= \frac{nA_n + v_{n+1}}{n+1} \\
&= \frac{v_{n+1} + nA_n + A_n - A_n}{n+1} \\
&= \frac{v_{n+1} + (n+1)A_n - A_n}{n+1} \\
&= A_n + \frac{v_{n+1} - A_n}{n+1}
\end{aligned}$$



```
Initialize Q(s, a) arbitrarily
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) <-- Q(s, a) + $\alpha$ [r + $\gamma$ max$_{\alpha'}$Q(s', a') - Q(s, a)]
        s <-- s';
    until s is terminal
```

[OpenAI Gym]



Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.

View documentation ›

View on GitHub ›

```
import gym
env = gym.make("CartPole-v1")
observation = env.reset()
for _ in range(1000):
  env.render()
  action = env.action_space.sample() # your agent here (this takes random actions)
  observation, reward, done, info = env.step(action)

  if done:
    observation = env.reset()
env.close()
```

RandomAgent on CartPole-v1

In large state spaces: Need approximation

Challenge: finding good features

$$\hat{Q}^{\pi}(s,a) = \sum_{i=1}^{d} \theta_i \underbrace{\phi_i(s,a)}_{\text{feature vector}}$$

Fitted Q-iteration

- Use **supervised learning** to estimate the **Q-function** from a batch of training data.
  - Input: $x_i := <s_i, a_i>, \ i=1..N$
  - Output: $y_i := r_i + \gamma \ max_a Q_\theta(s_i',a)$
  - Loss: $\sum_i || r_i + \gamma \ max_a Q_\theta(s_i',a) - Q_\theta(s_i,a_i) ||^2$
- Regression with linear function, neural network, etc.

  (Can use other functions, e.g. random forests.)

Fitted Q-iteration

- Use **supervised learning** to estimate the **Q-function** from a batch of training data.
  - Input: $x_i := \langle s_i, a_i \rangle, \; i=1..N$
  - Output: $y_i := r_i + \gamma \max_a Q_\theta(s_i',a)$
  - Loss: $\sum_i || r_i + \gamma \max_a Q_\theta(s_i',a) - Q_\theta(s_i,a_i) ||^2$

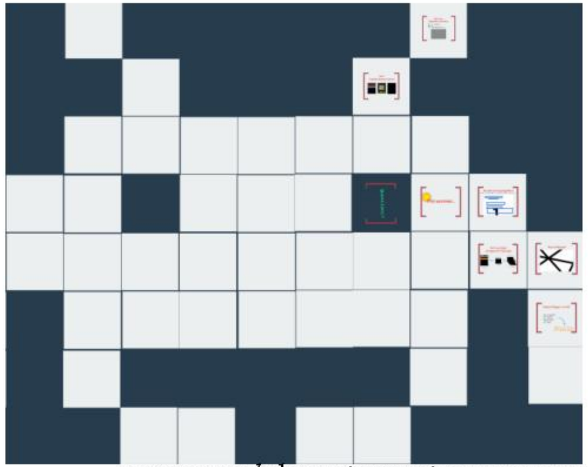
- Regression with linear function, neural network, etc.

  (Can use other functions, e.g. random forests.)

- **Important note**: Q_θ appears twice in the loss => Hard to learn!
  - And in addition, r can be very sparse.

The Arcade Learning Environment

- Several Atari 2600 Games
- States:
  - 210x160 colour video at 60Hz
- Actions:
  - Discrete, small set



www.arcadelearningenvironment.org

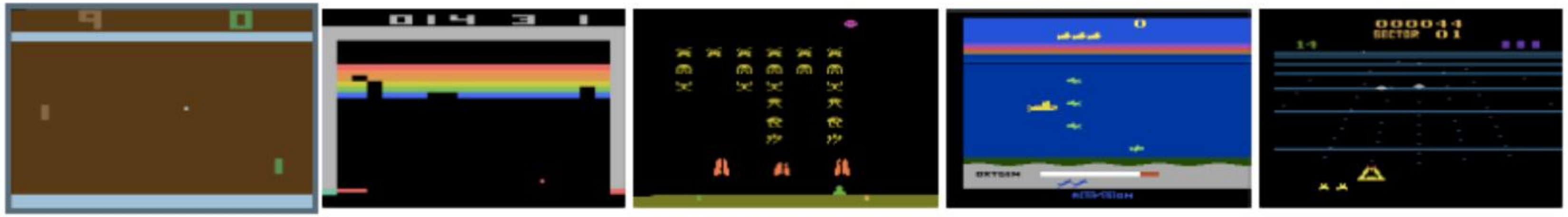
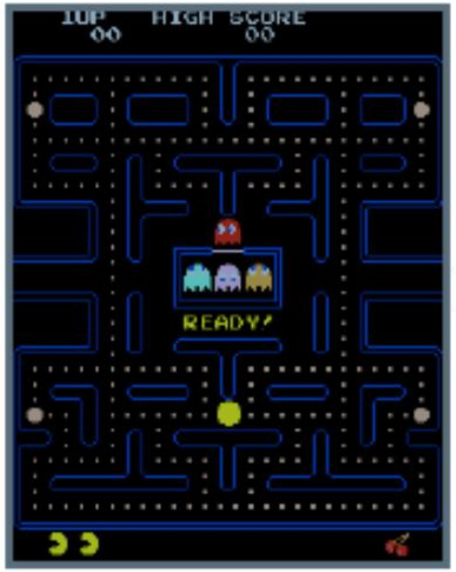


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

Learning representations for RL

s

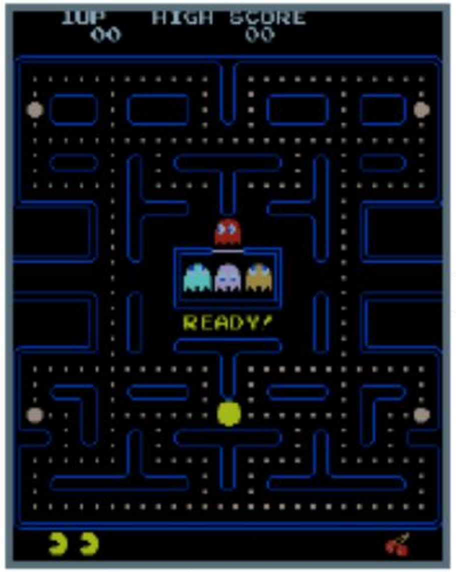


Original state

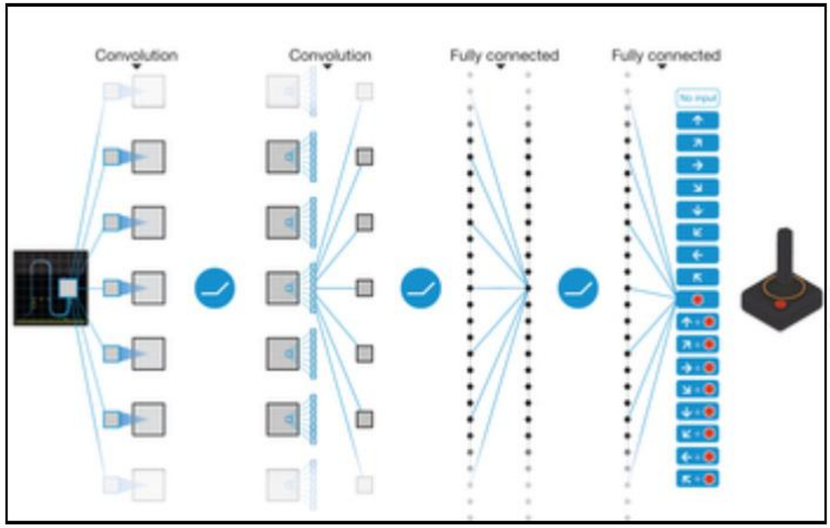
$\rightarrow$ $\phi(s)?$ $\rightarrow$ $Q_\theta(s,a)$

Deep Q-network (DQN)

s



Original state



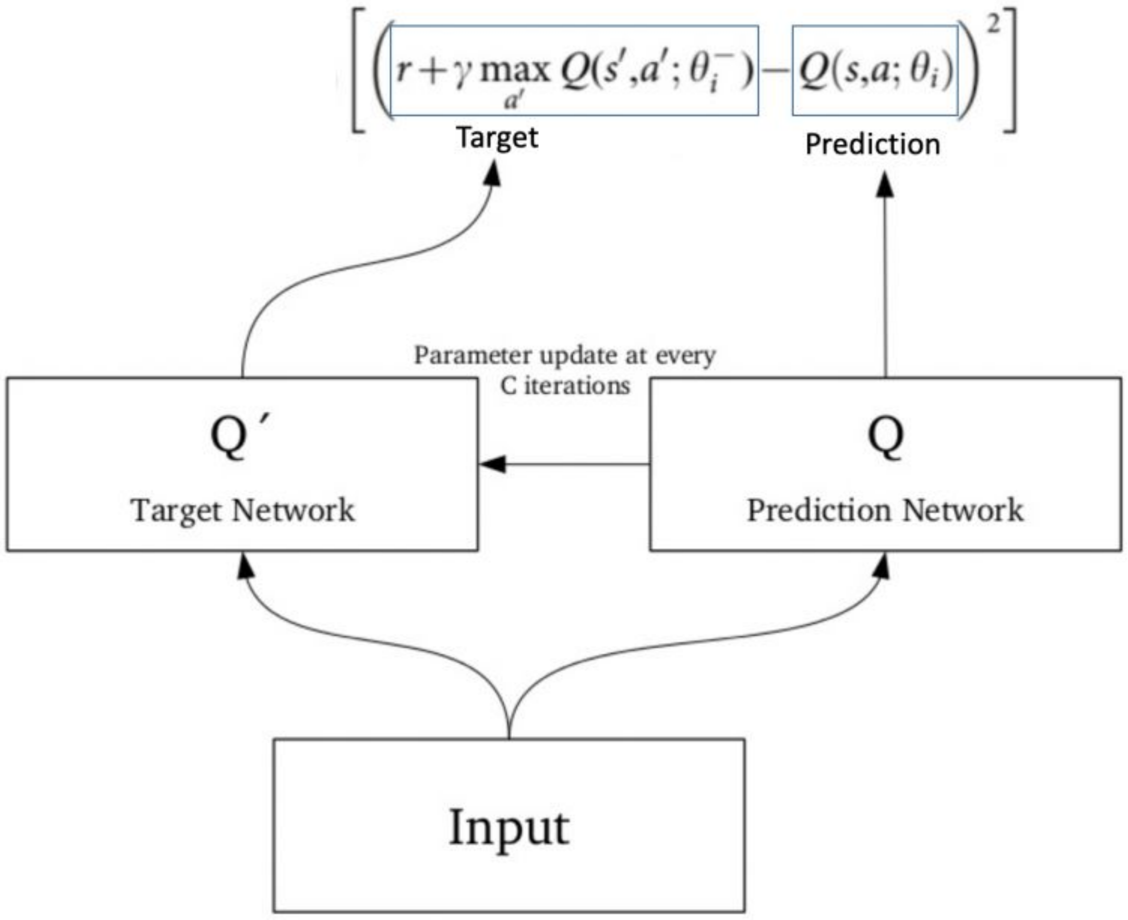
Convolutional Neural Net

$Q_\theta(s,a)$

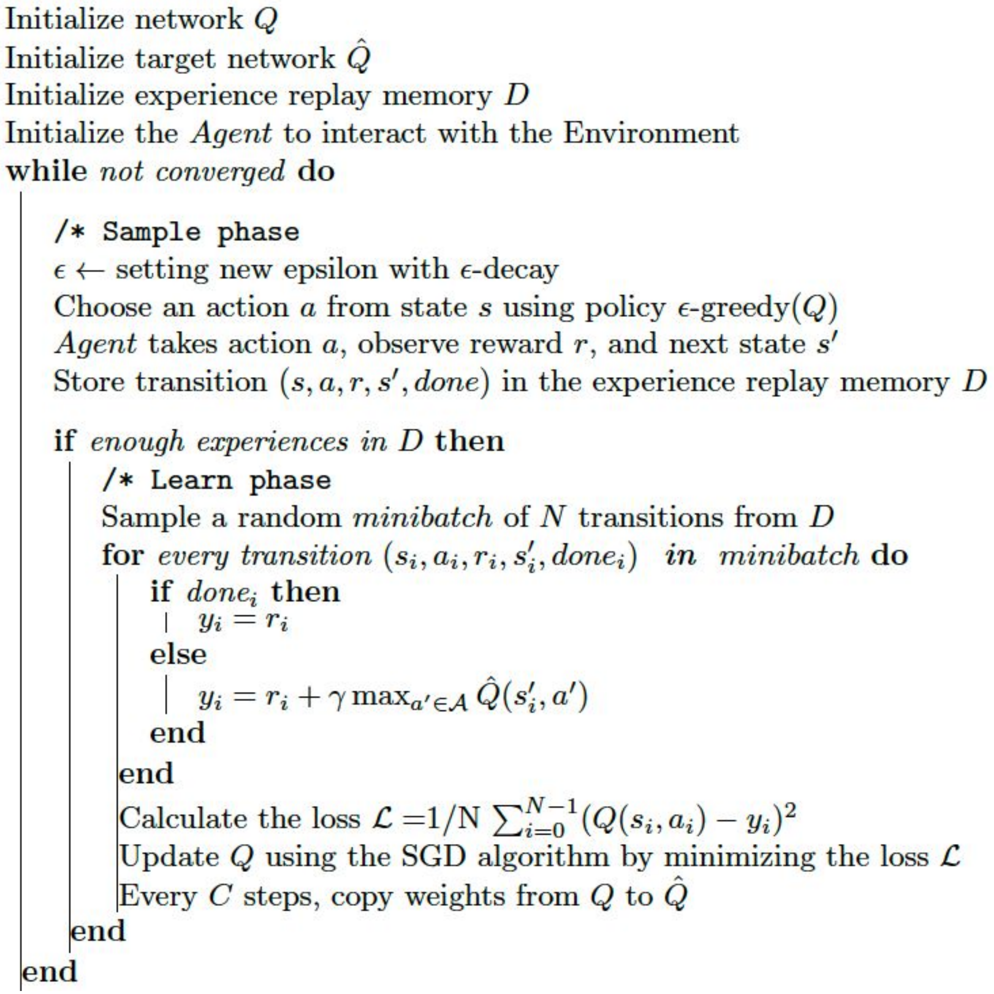
Trained with stochastic gradient descent.

[DeepMind: Mnih et al., 2015].

Deep Q-network (DQN)



$\left[\left(\boxed{r+\gamma \max_{a'} Q(s',a';\theta_i^-)} - \boxed{Q(s,a;\theta_i)}\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input



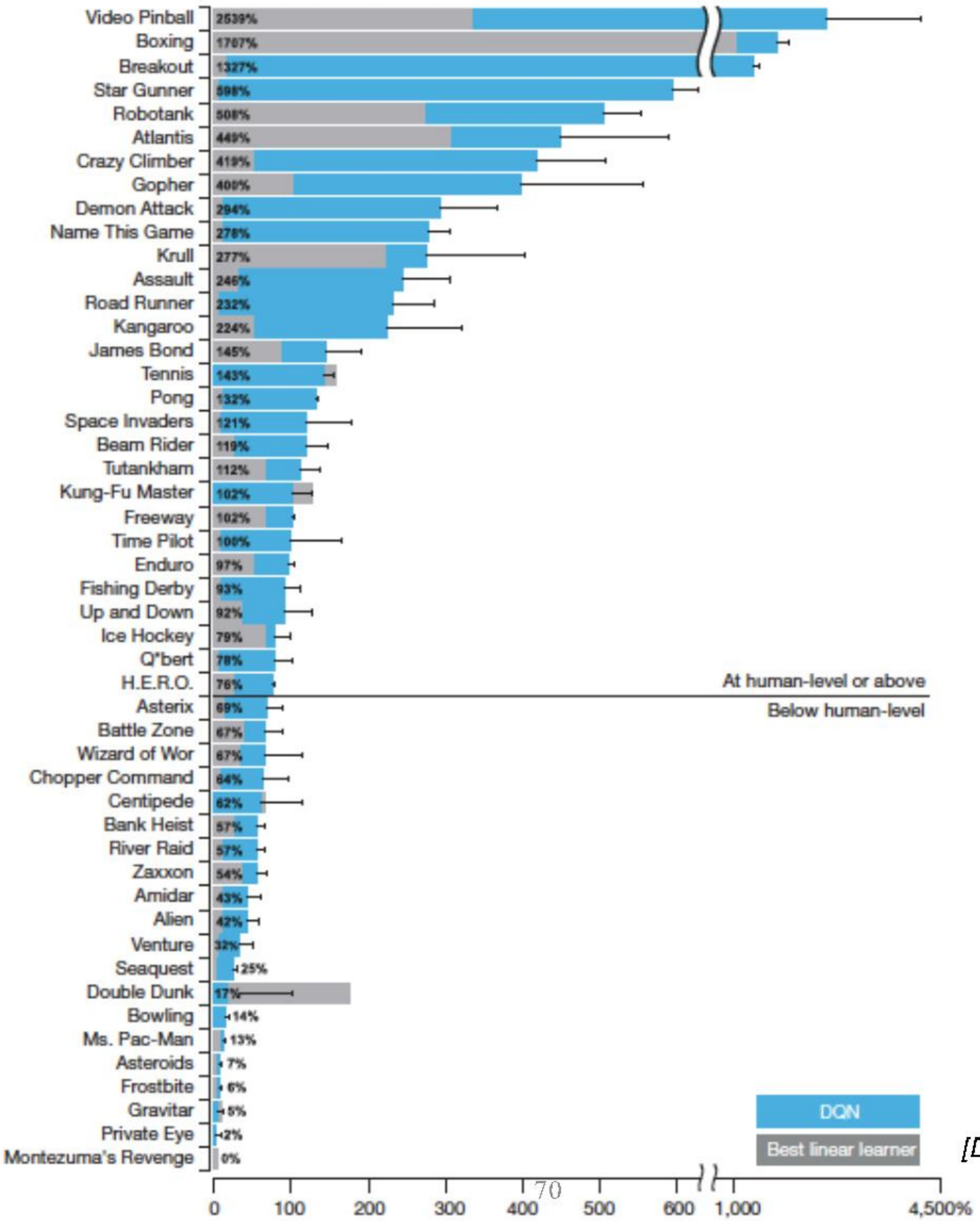
```
Initialize network $Q$
Initialize target network $\hat{Q}$
Initialize experience replay memory $D$
Initialize the Agent to interact with the Environment
while not converged do
    /* Sample phase
    $\epsilon \leftarrow$ setting new epsilon with $\epsilon$-decay
    Choose an action $a$ from state $s$ using policy $\epsilon$-greedy($Q$)
    Agent takes action $a$, observe reward $r$, and next state $s'$
    Store transition $(s, a, r, s', done)$ in the experience replay memory $D$

    if enough experiences in $D$ then
        /* Learn phase
        Sample a random minibatch of $N$ transitions from $D$
        for every transition $(s_i, a_i, r_i, s'_i, done_i)$ in minibatch do
            if $done_i$ then
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
            end
        end
        Calculate the loss $\mathcal{L}$ =1/N $\sum_{i=0}^{N-1}(Q(s_i, a_i) - y_i)^2$
        Update $Q$ using the SGD algorithm by minimizing the loss $\mathcal{L}$
        Every $C$ steps, copy weights from $Q$ to $\hat{Q}$
    end
end
```

Train / Test protocol for RL

- Choose an exploration policy. Run it. Get a batch of data.
- Train your Q-function. (Stop training, fix *Q()*.)
- Use your learned Q-function to generate new trajectories. Measure the utility on these new trajectories.
- Repeat.

(Never report results for a hold-out test set.)



[DeepMind: Mnih et al., 2015].

DQN: Useful tips for stability

- **Experience replay** *[Mnih et al., 2015]*
  - Store large batch of observed experiences: <s_t, a_t, r_t, s_{t+1}>.
  - Update Q-function by randomly drawing mini-batch of experiences.

- **Prioritized experience replay** *[Schaul et al., 2016]*
  - Replay important transitions more frequently.
  - Higher TD error => higher probability of being sampled.

DQN: Useful tips for stability

- **Periodic updates to target value** *[Mnih et al., 2015]*
  - Use a fixed target network $Q_{\theta}^{-}()$ to calculate the error.
  - Apply updates to a separate network $Q_{\theta}^{+}()$.
  - Every k iterations substitute $Q_{\theta}^{-}() <= Q_{\theta}^{+}()$.

- **Gradient clipping**

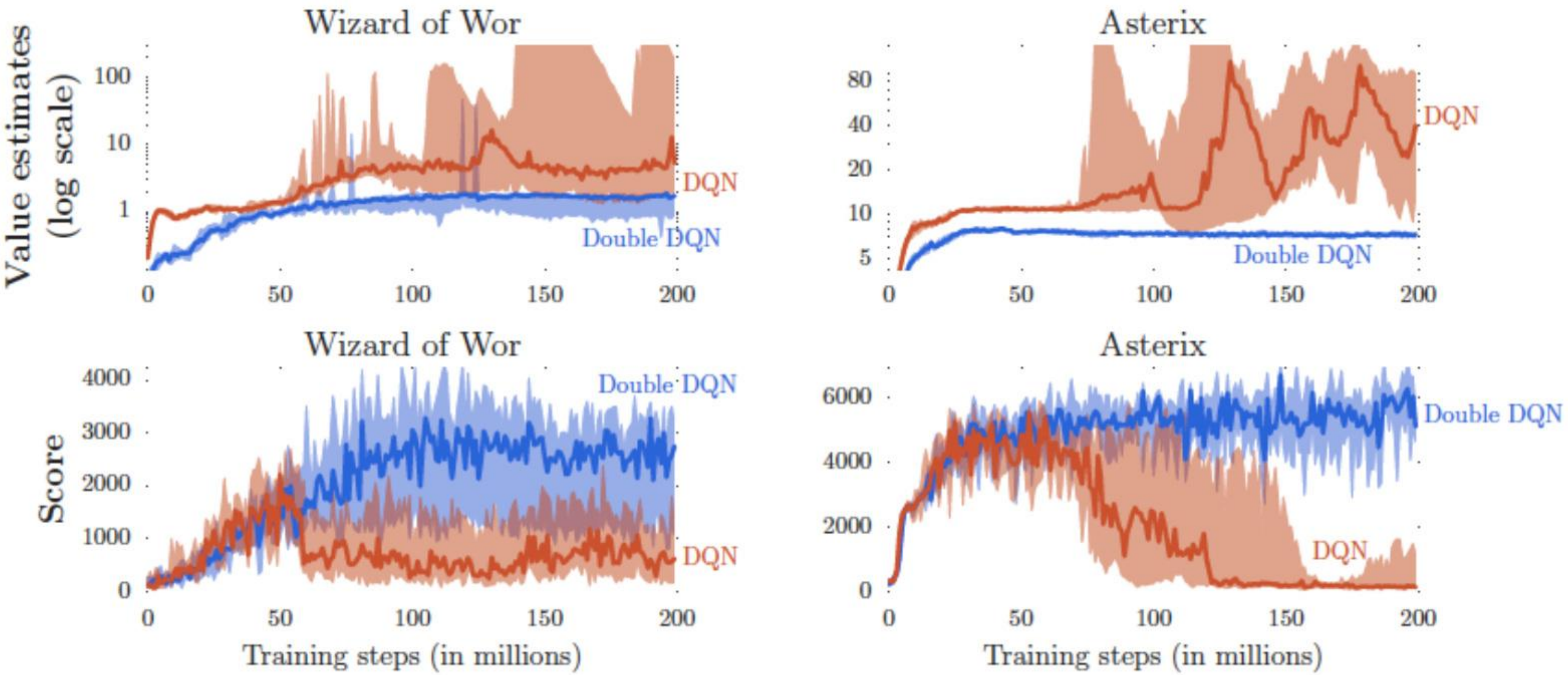
DQN: Useful tips for stability

- **Periodic updates to target value** *[Mnih et al., 2015]*
  - Use a fixed target network $Q_\theta^-()$ to calculate the error.
  - Apply updates to a separate network $Q_\theta^+()$.
  - Every k iterations substitute $Q_\theta^-() <= Q_\theta^+()$.

- **Gradient clipping**

- **Double DQN** *[van Hasselt et al., 2016]*
  - Q-values are biased (over-estimated) due to *max* operator.
  - Use output: $y_i := r_i + \gamma\ Q_{\theta-}(s_i', argmax_a Q_{\theta+}(s_i',a))$
    - » $Q_{\theta+}$ is used to select the action
    - » $Q_{\theta-}$ is used to calculate the error.

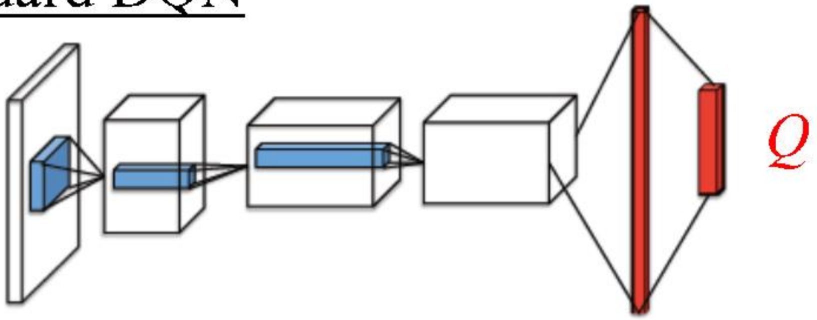
Double DQN: Avoiding positive bias



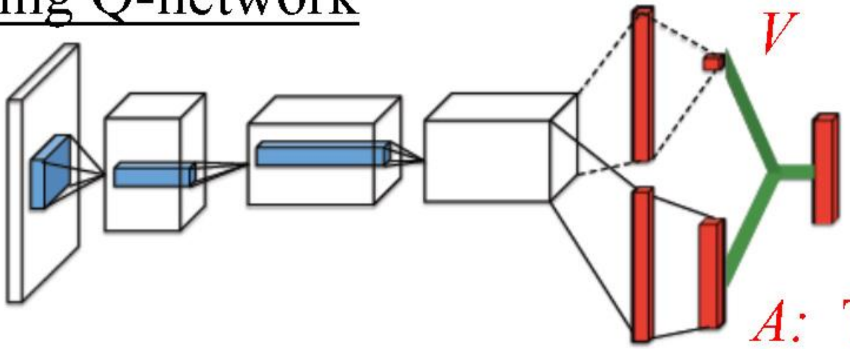
[DeepMind: van Hasselt et al., 2015].

Dueling Q-networks

Standard DQN



Dueling Q-network

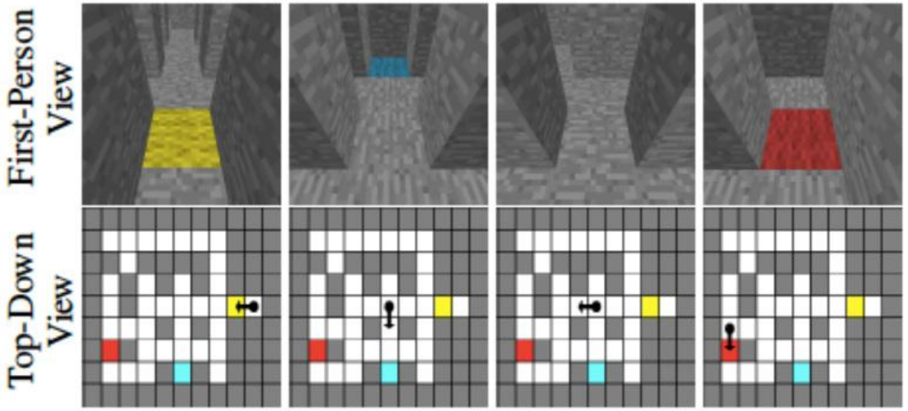


$Q_{\theta,\alpha,\beta} = V_{\theta,\beta} + A_{\theta,\alpha}$

A: The **advantage** function

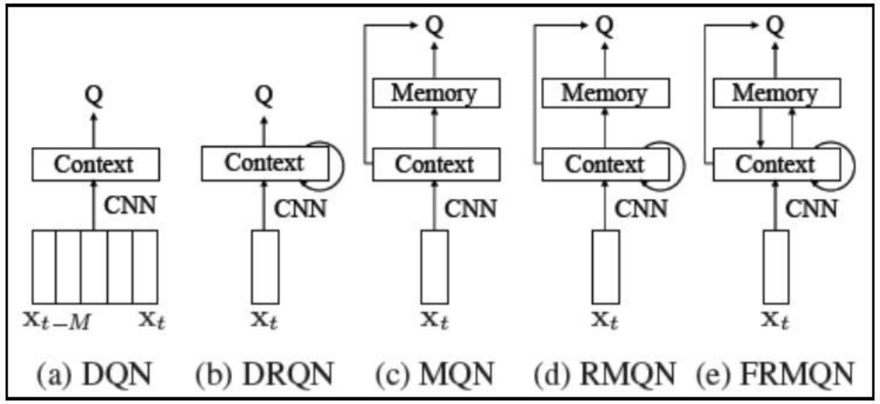
[DeepMind: Wang et al., 2016].

Deep RL in Minecraft



(a) t=3 (b) t=10 (c) t=11 (d) t=19

Figure 1. Example task in Minecraft. In this task, the agent should visit the red block if the indicator (next to the start location) is yellow. Otherwise, if the indicator is green, it should visit the blue block. The top row shows the agent's first-person observation.



Many possible architectures,
Incl. memory and context

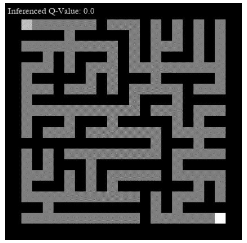
Online videos: *https://sites.google.com/a/umich.edu/junhyuk-oh/icml2016-minecraft*

[U.Michigan: Oh et al., 2016].

RL Libraries (short list)

TF Agents TensorFlow

pyqlearning



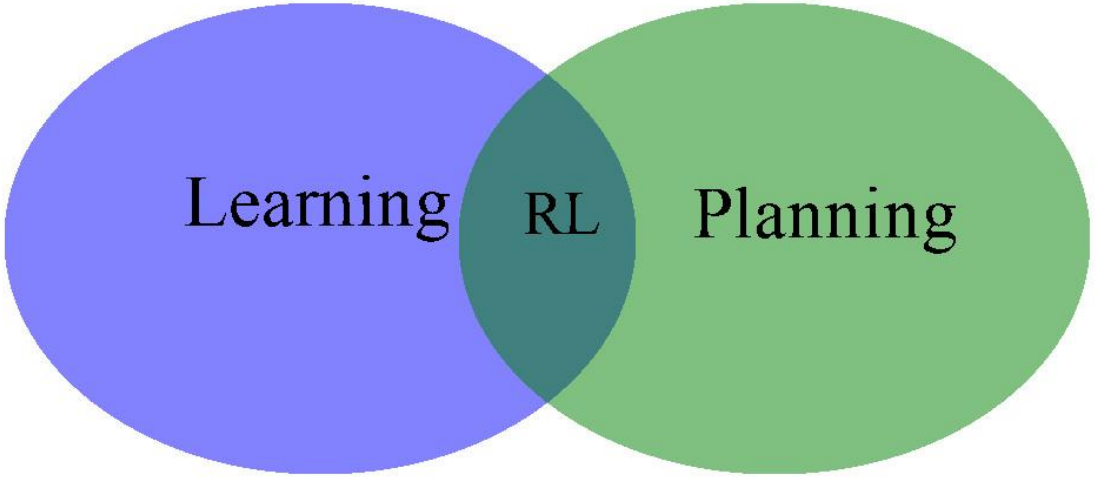
keras-rl2 Keras RL2

spinup OpenAI Spinning Up

RL Baselines3 Zoo

Deep Q-learning in the real world?

- More work on **Mario, Starcraft, Doom, ….**
- All these results make extensive use of a simulator.
- Domain is often (near-)deterministic.
- Relative small set of actions (=small policy space).


Learning
RL
Planning