Reinforcement Learning:

From basic concepts to deep Q-networks

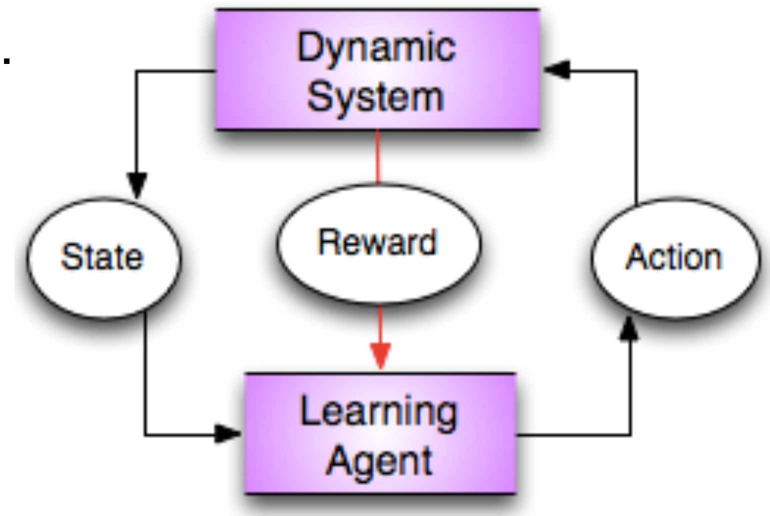
Joelle Pineau
McGill University

Deep Learning Summer School
August 2016

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.

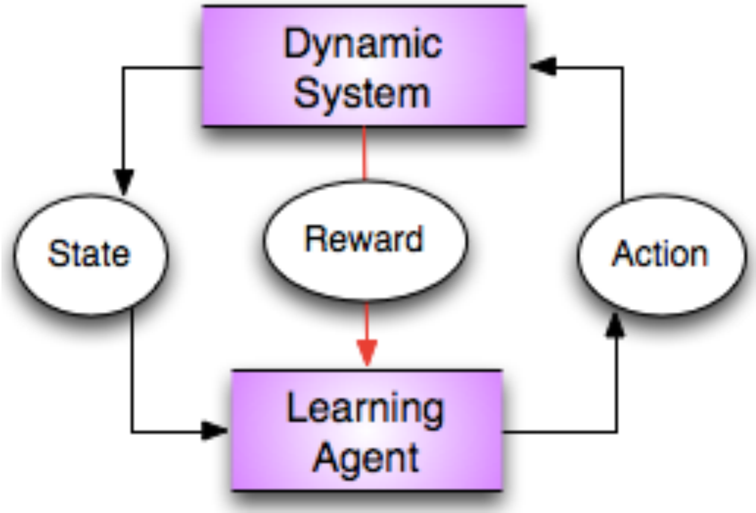


Reinforcement learning

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, $Pr(s_t|s_{t-1},a_t)$.

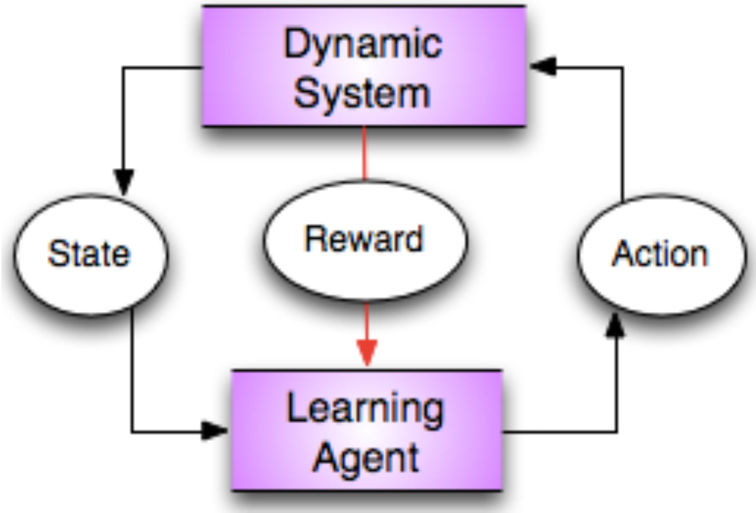


Reinforcement learning

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.

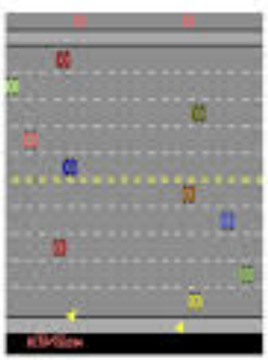
$$argmax_\pi E_\pi [r_0 + r_1 + \ldots + r_T \mid s_0]$$

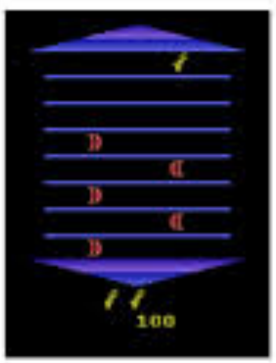
http://en.wikipedia.org/wiki/Animal_training

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

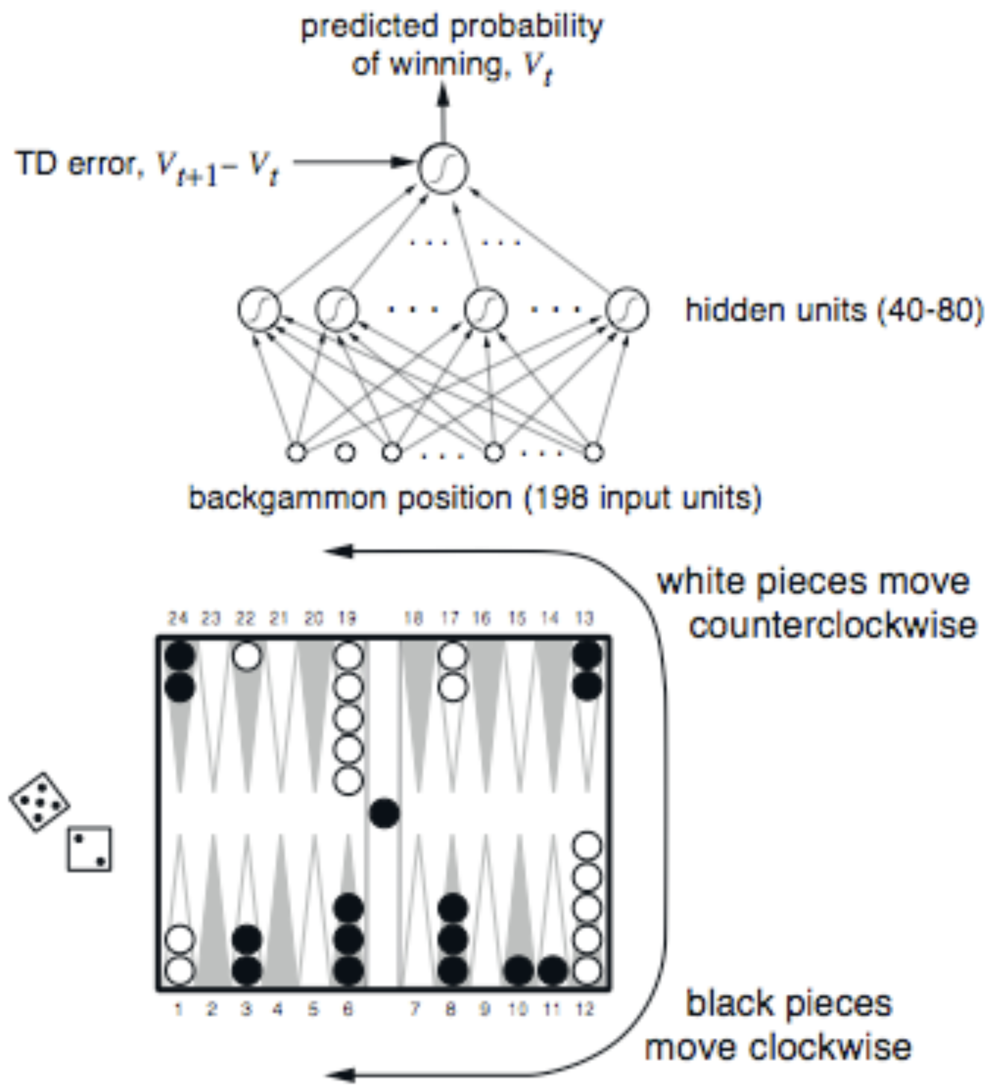
Many applications of RL

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





RL system circa 1990's: TD-Gammon



Reward function:

+100 if win

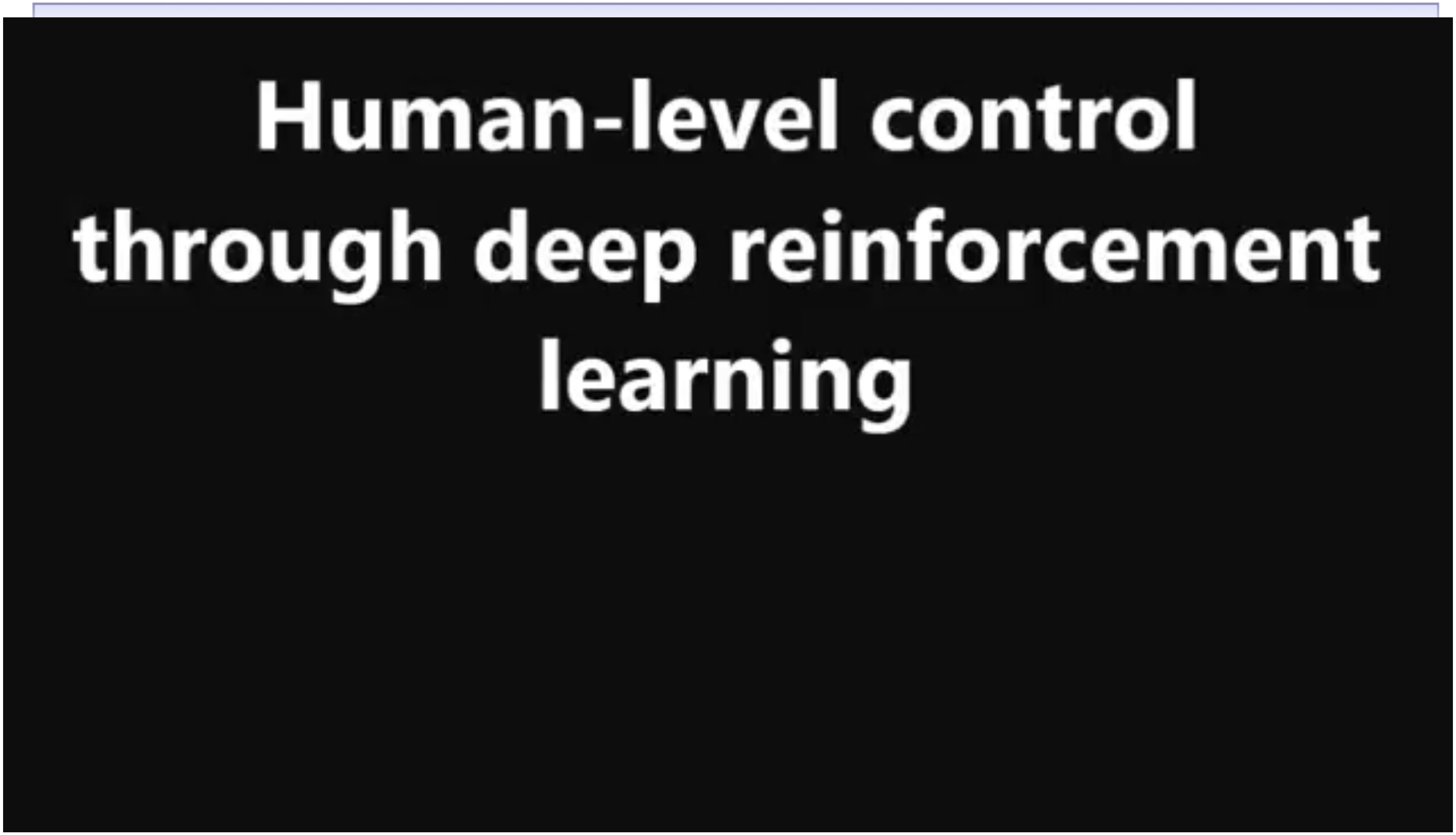
- 100 if lose

0 for all other states

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

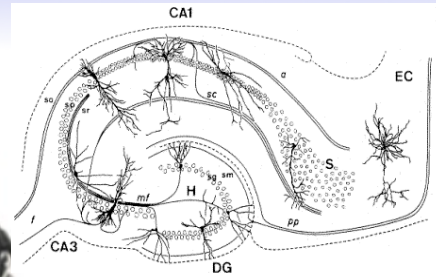
Enough to beat the best human player.

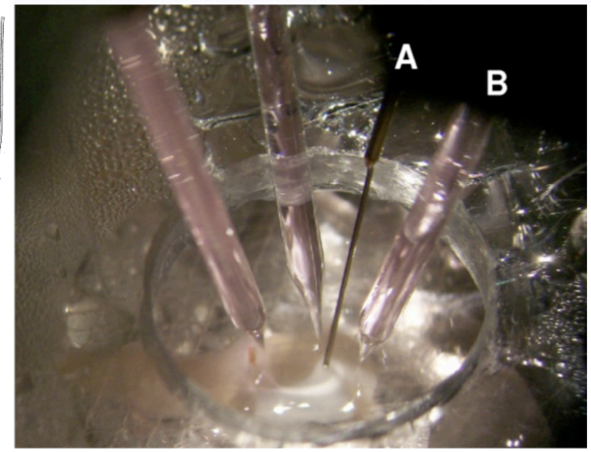
Human-level Atari agent (2015)

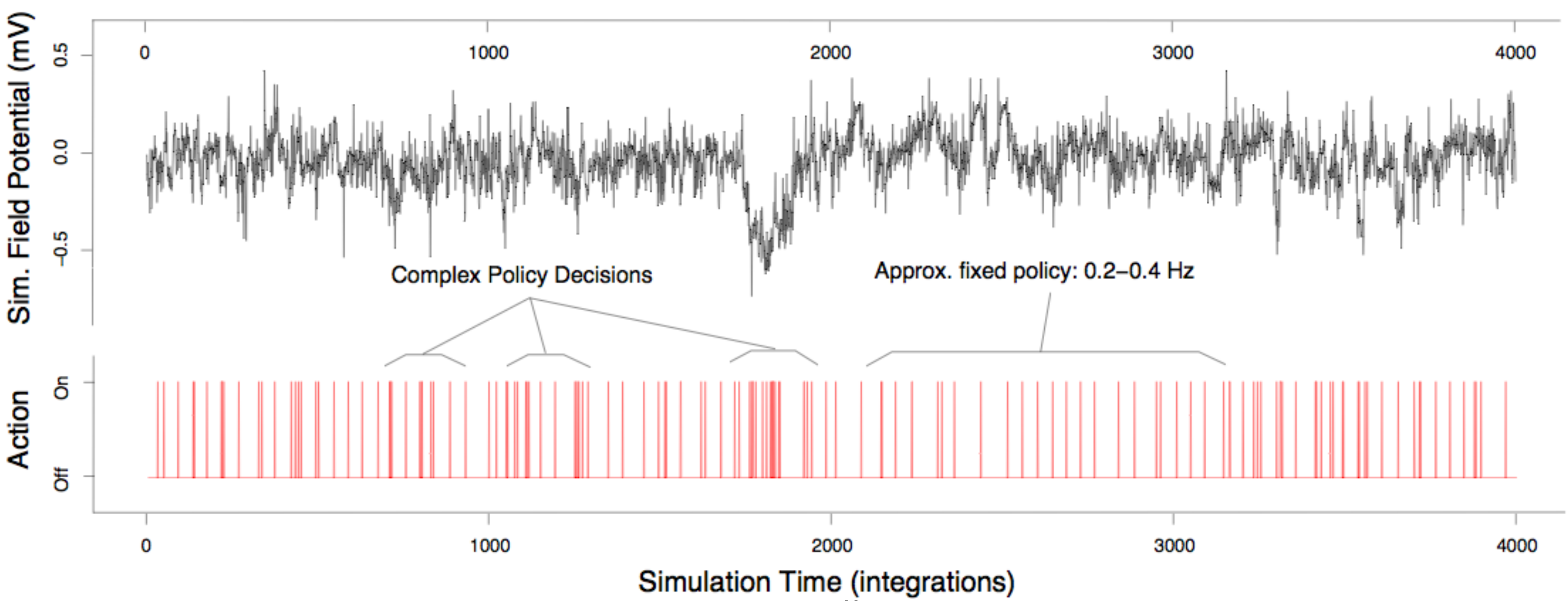


DeepMind's AlphaGo (2016)

Adaptive neurostimulation for epilepsy suppression



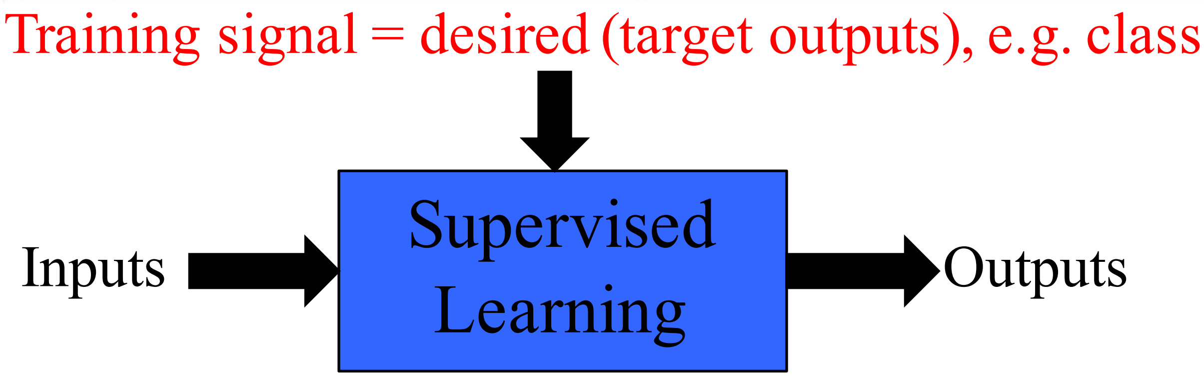




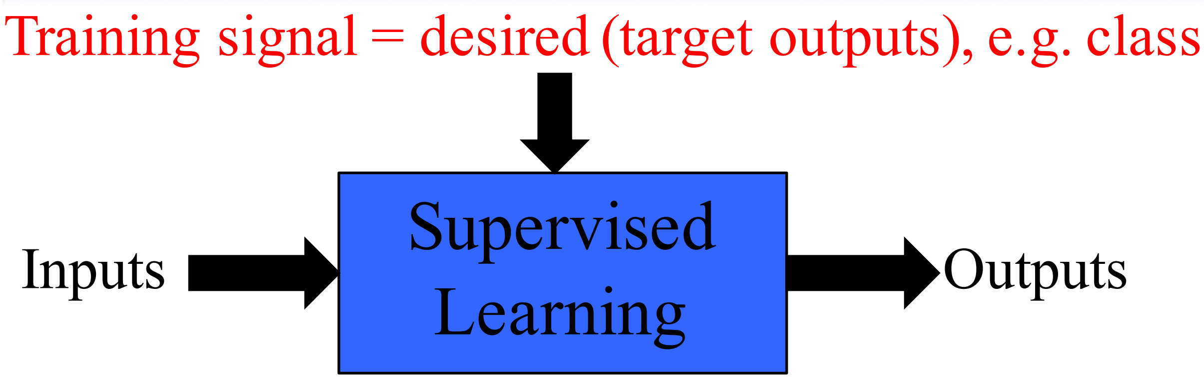
When to use RL?

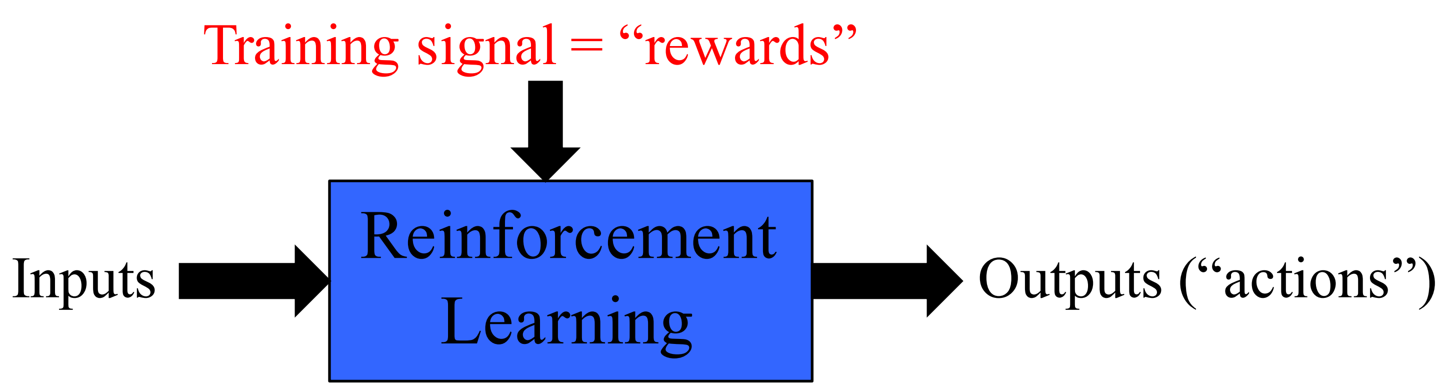
- Data in the form of trajectories.
- Need to make a sequence of (related) decisions.
- Observe (partial, noisy) feedback to state or choice of actions.
- There is a gain when optimizing action choice over a portion of the trajectory.

RL vs supervised learning

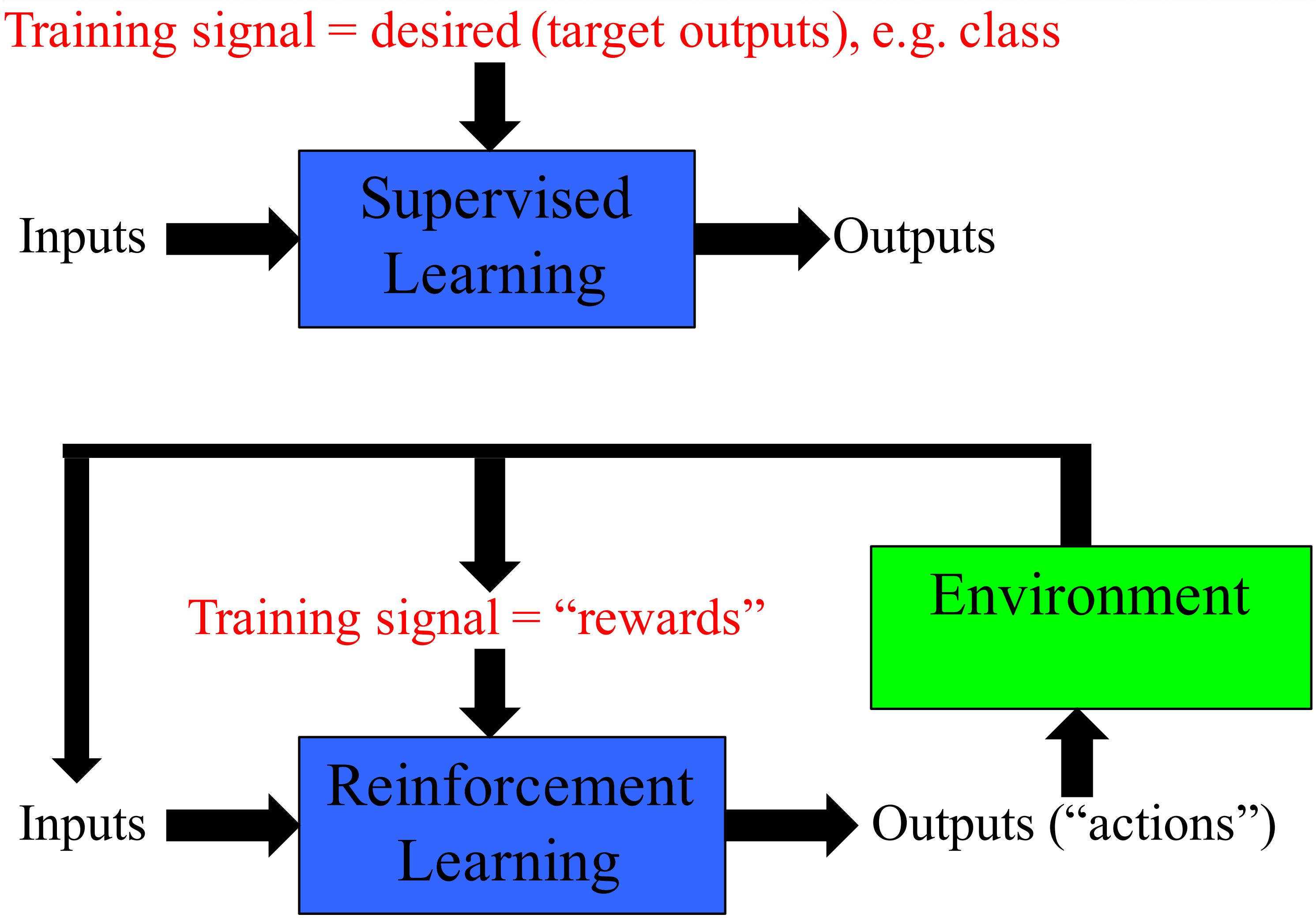


RL vs supervised learning

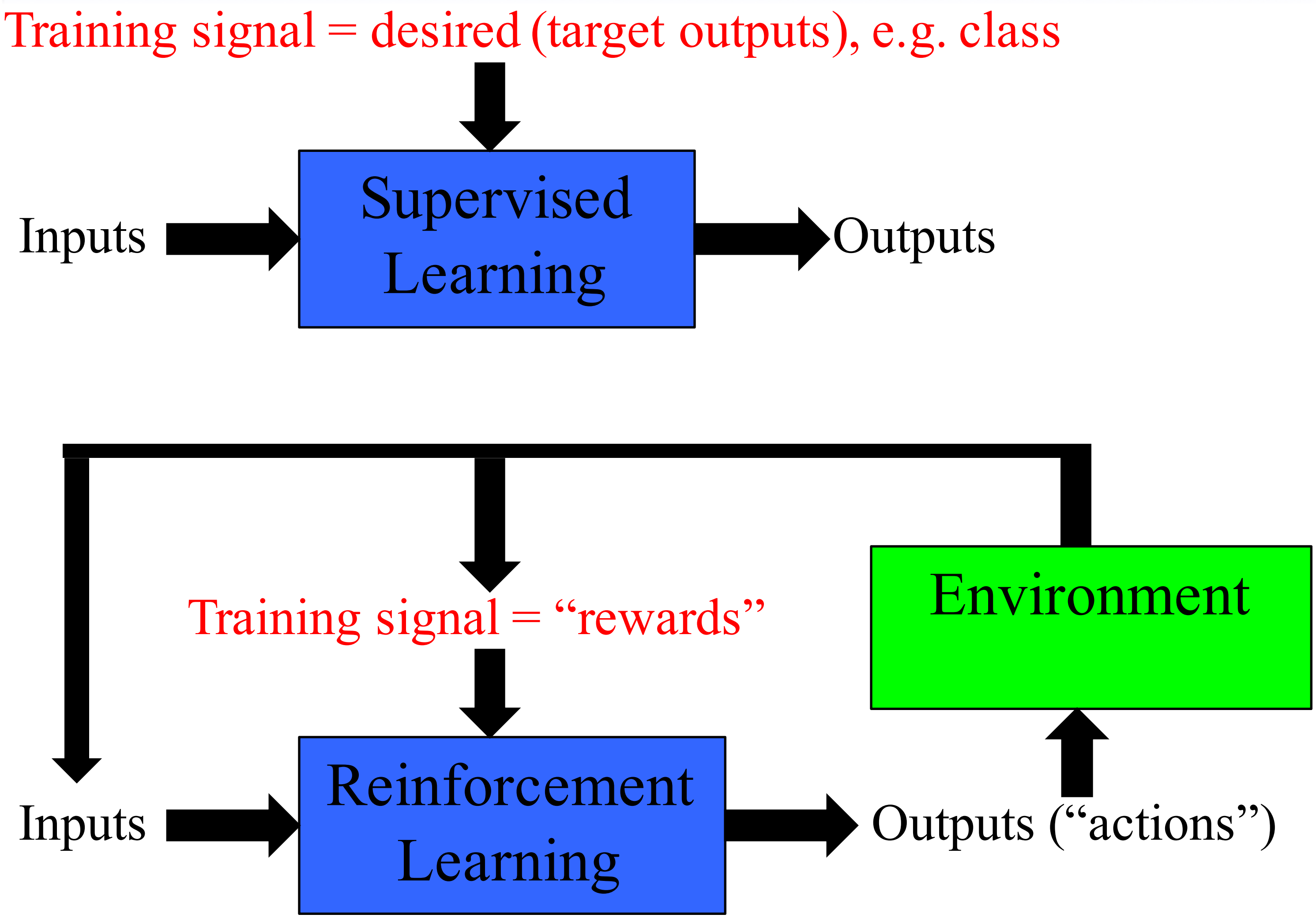




RL vs supervised learning



RL vs supervised learning



<u>Challenges:</u>

Jointly learning AND planning from correlated samples.

Data distribution changes with action choice.

Need access to the environment.

Markov Decision Process (MDP)

Defined by:

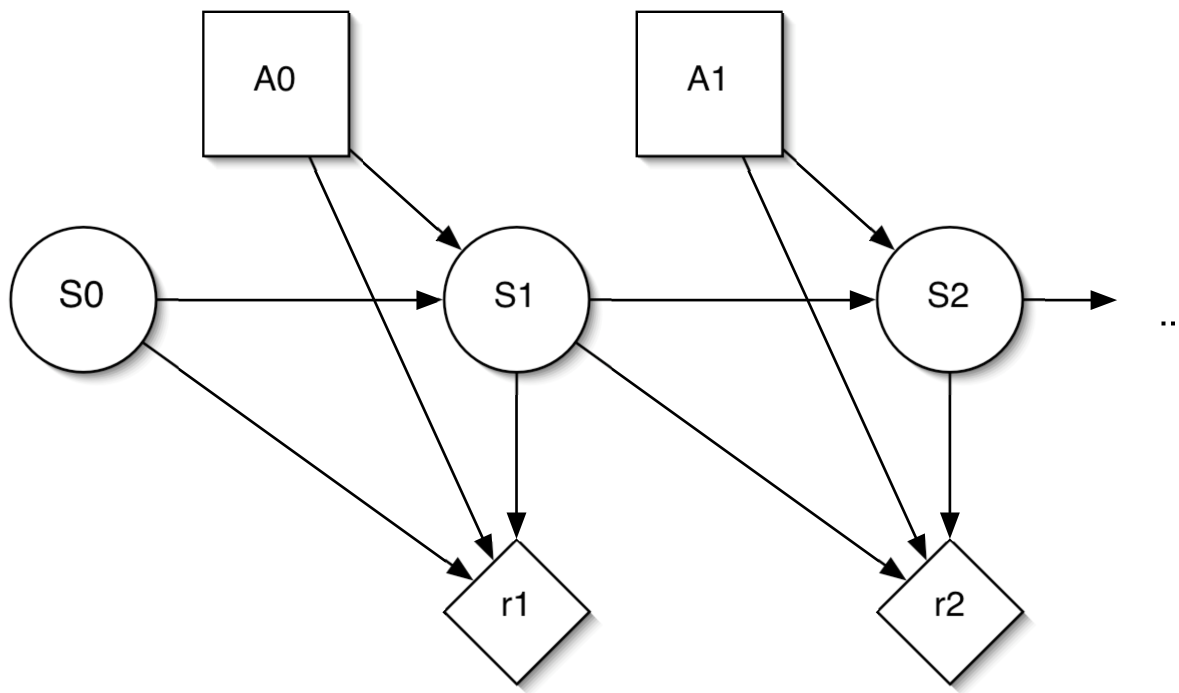
S: Set of states

A: Set of actions

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

r_t : Reward function

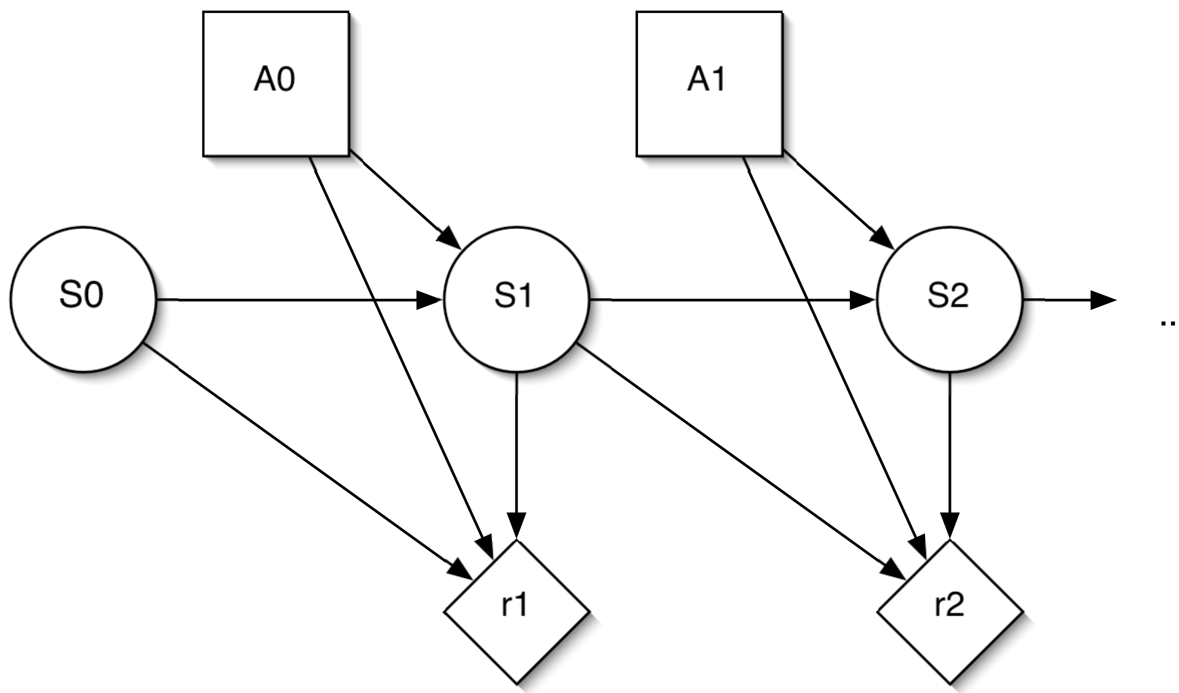
μ_t : Initial state distribution



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.)

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

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

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

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:

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

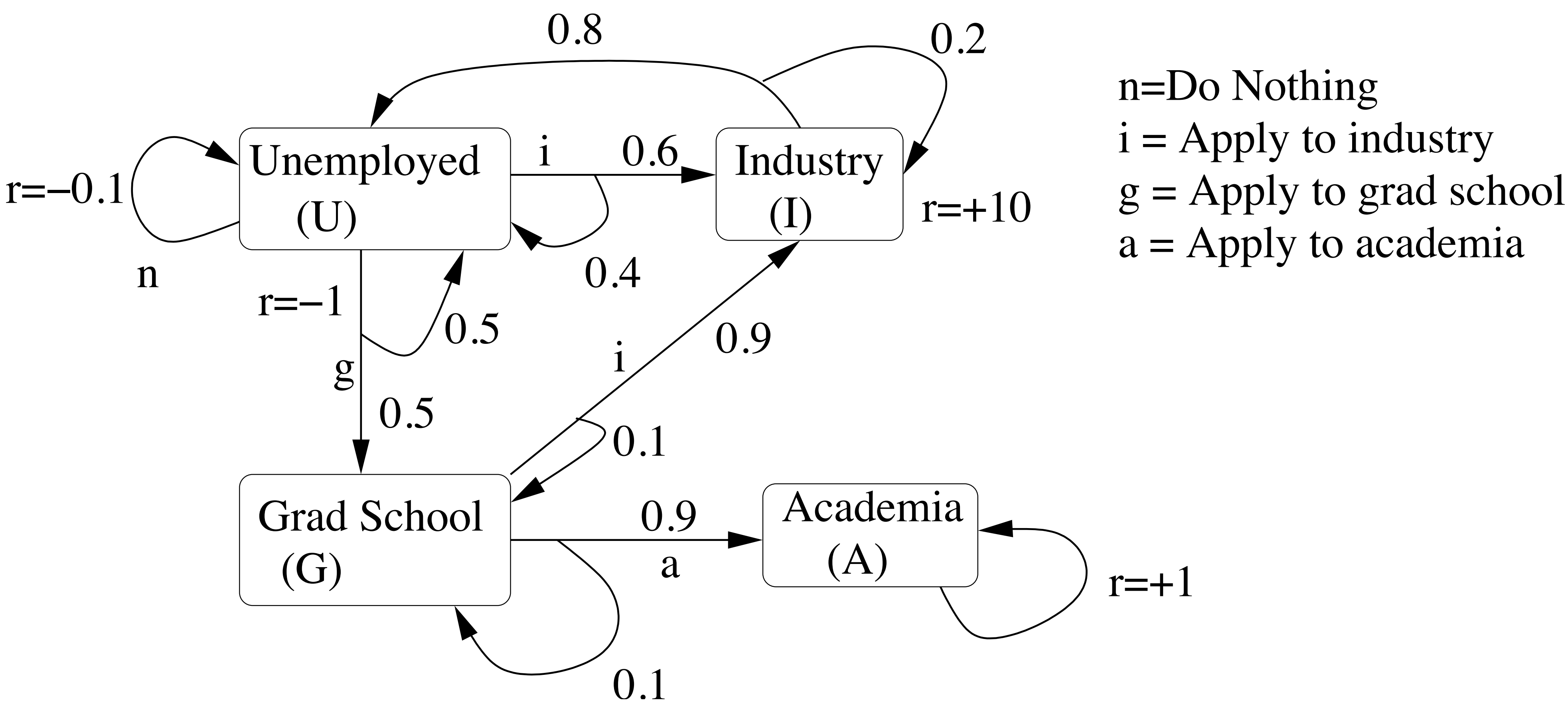
(Can be stochastic as above, or deterministic, $S \rightarrow A$.)

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

(But there are many policies!)

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

Example: Career Options



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.

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

The value of a policy

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

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

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

The value of a policy

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

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

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

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

The value of a policy

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

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

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

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

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

This is a **dynamic programming** algorithm.

The value of a policy

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

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

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:

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

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:

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

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 $V_{k+1}(s)$:

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

- 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.

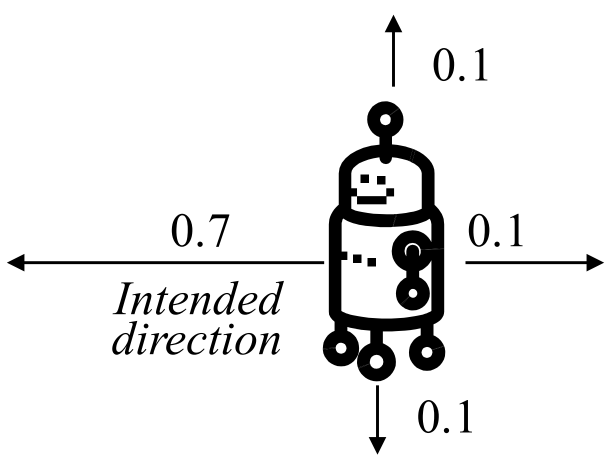
The algorithm converges (in the limit) to the true V^*.

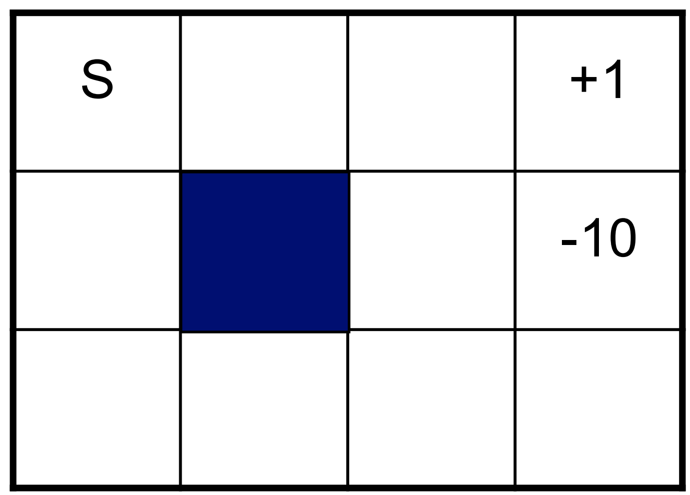
Questions?

- Policy evaluation
- Policy iteration
- Value iteration

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 $\gamma = 0.99$.





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

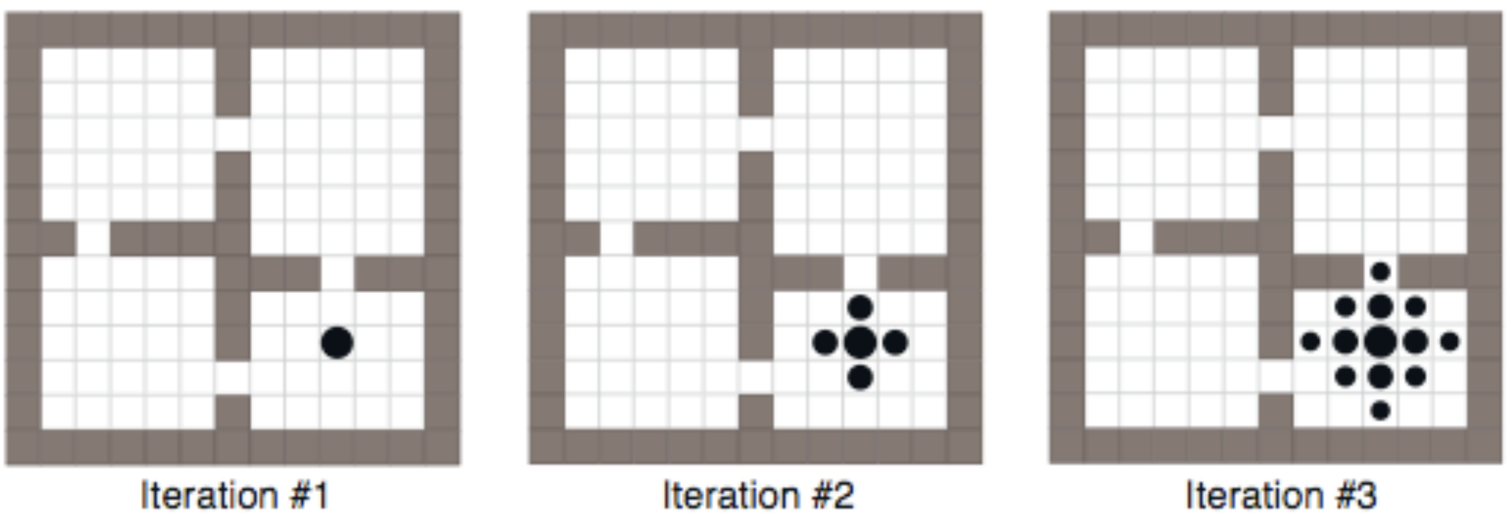
Value Iteration (20)

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

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

Another example: Four Rooms

- Four actions, fail 30% of the time.
- No rewards until the goal is reached, $\gamma = 0.9$.
- Values propagate backwards from the goal.



Asynchronous value iteration

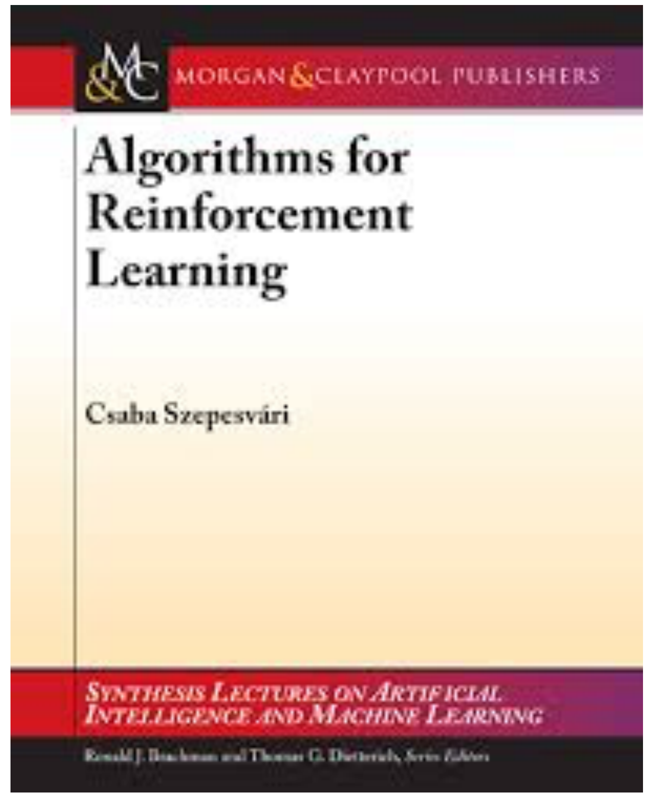
- Instead of updating all states on every iteration, focus on *important states*.
  - E.g., board positions that occur on every game, rather than just once in 100 games.

- 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



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 $V_t(s)$, $t=0..T$*

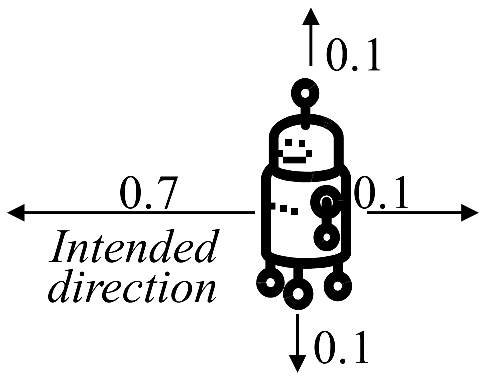
- **Continuing tasks:** e.g. tasks which may go on forever

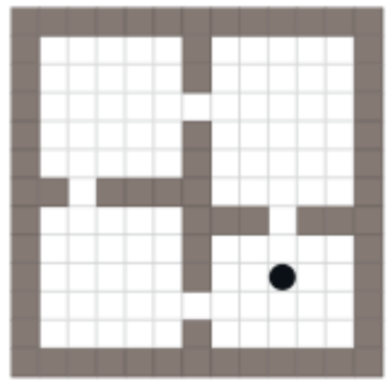
$$U_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \ldots = \sum_{k=0:\infty} \gamma^k r_{t+k}$$

** 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.





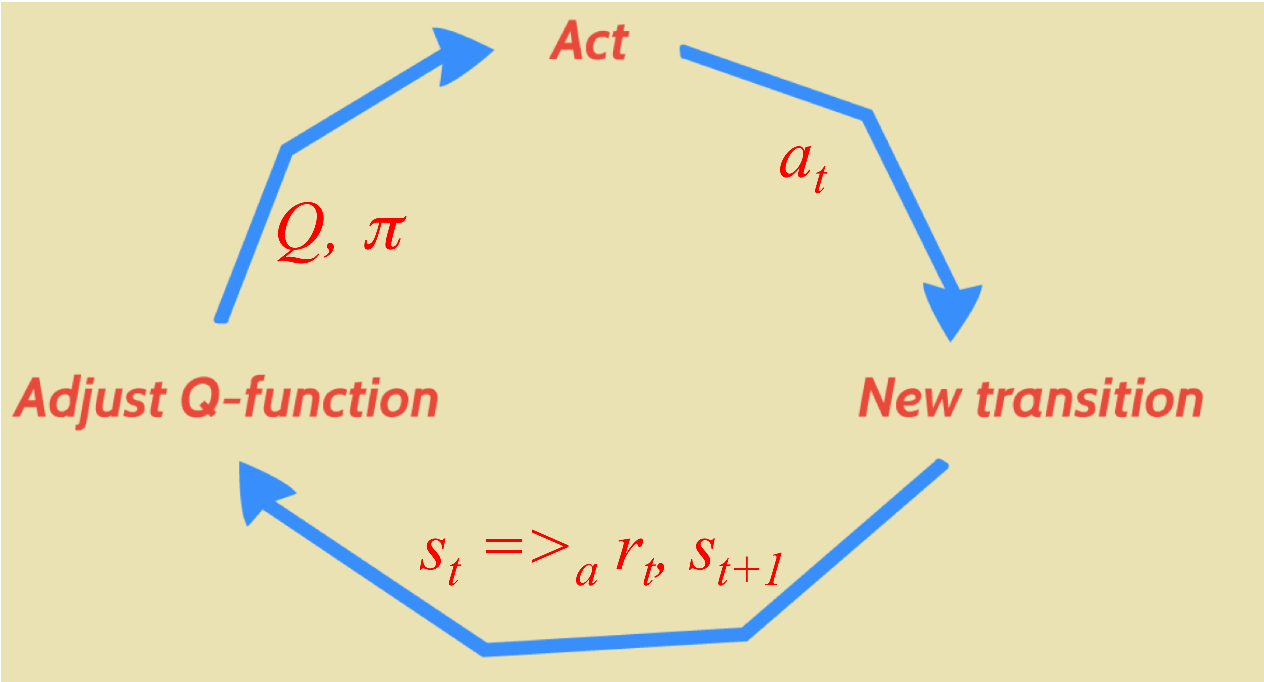
** Can prove many more* ***theoretical properties*** *in this case, about convergence, sample complexity.*

- **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)**:

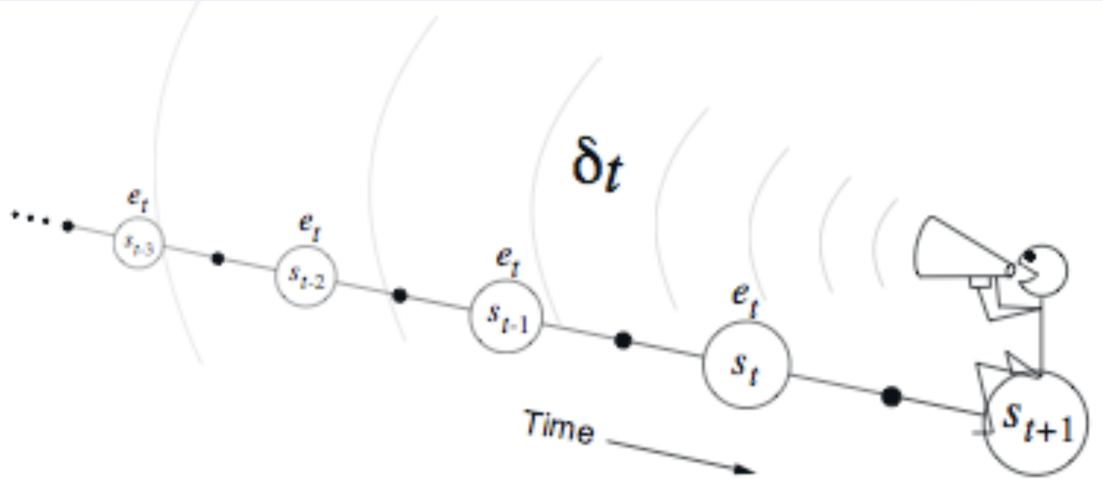
$$V(s_t) \leftarrow V(s_t) + \alpha\left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right) \forall t = 0, 1, 2, \ldots$$

- **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)

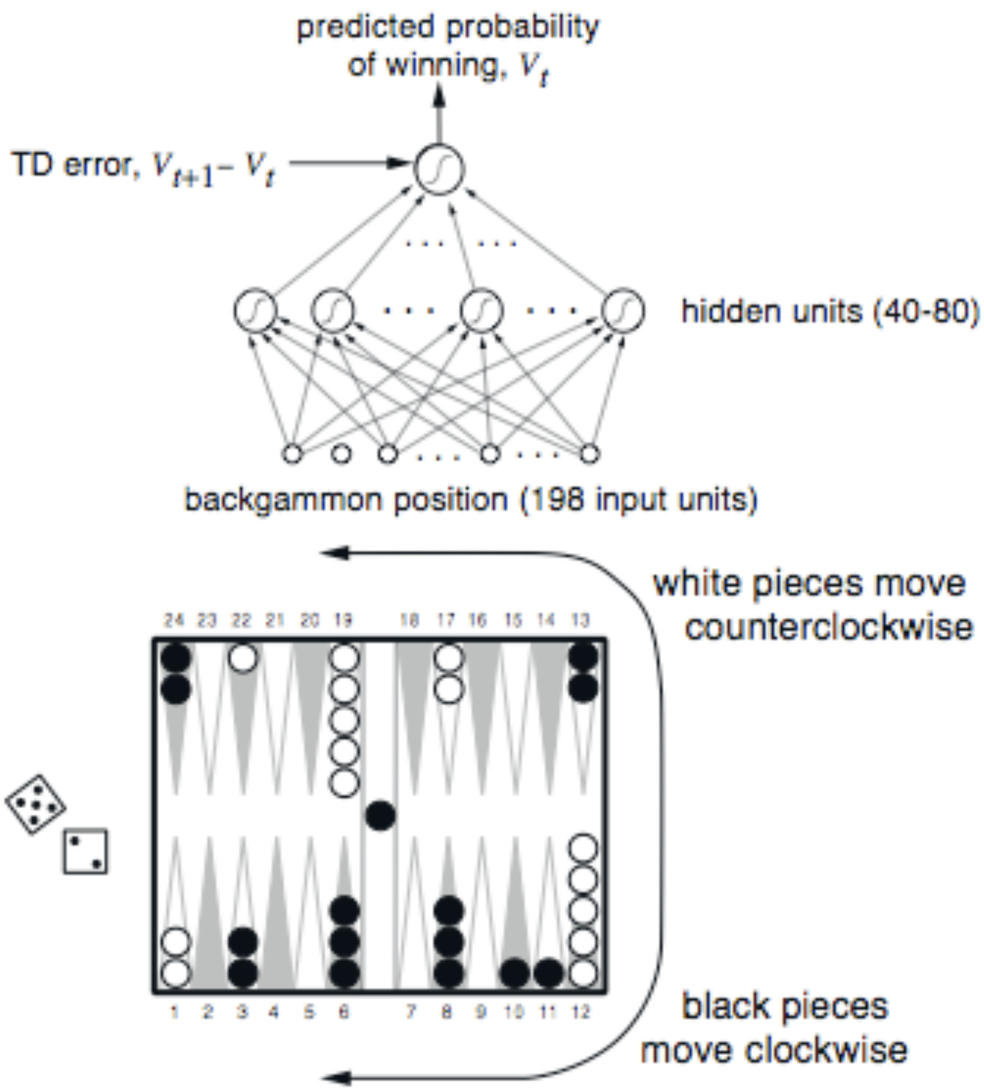


- On every time step t, we compute the TD error:

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

- 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 1.5×10^6 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

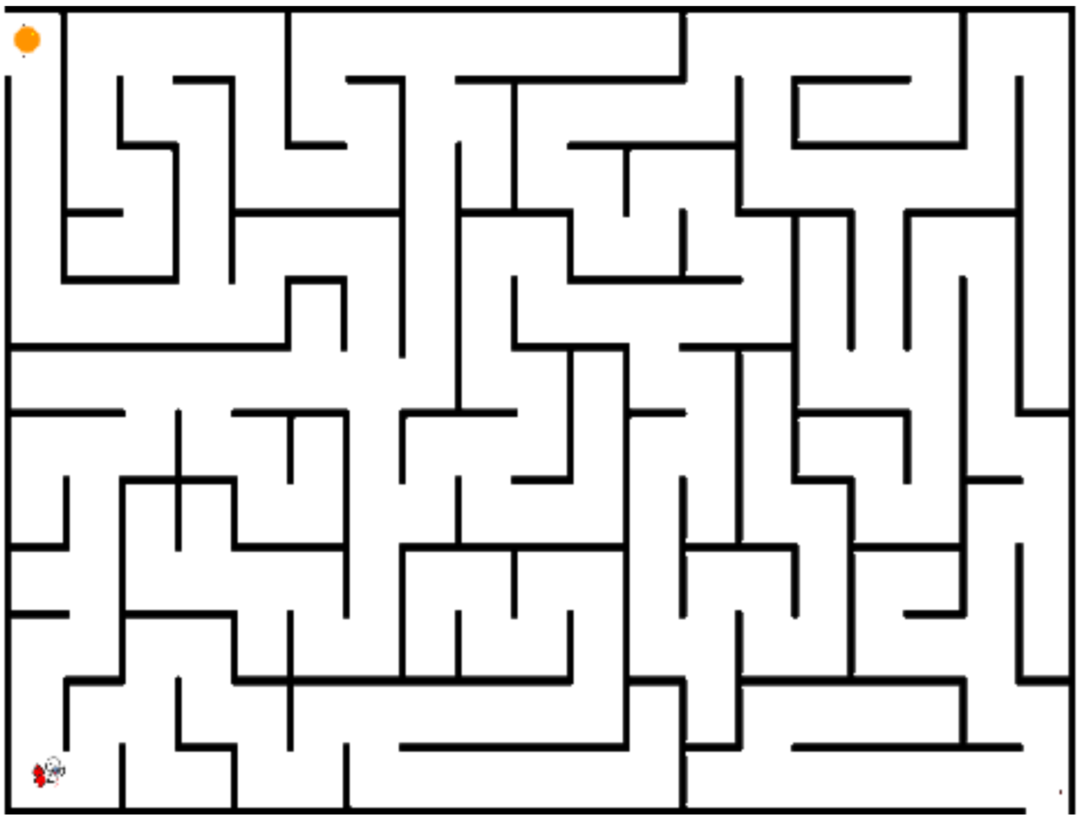
- Policy induces a distribution over the states (data).
  - Data distribution **changes** every time you change the policy!

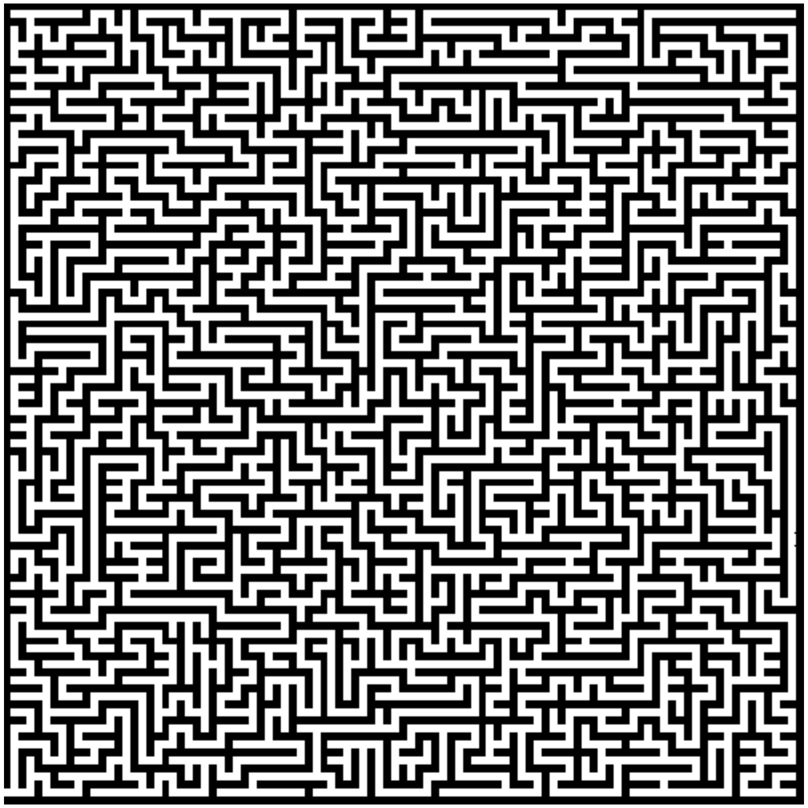
- Evaluating several policies with the same batch:
  - Need very big batch!
  - Need policy to adequately cover all *(s,a)* pairs.

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

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

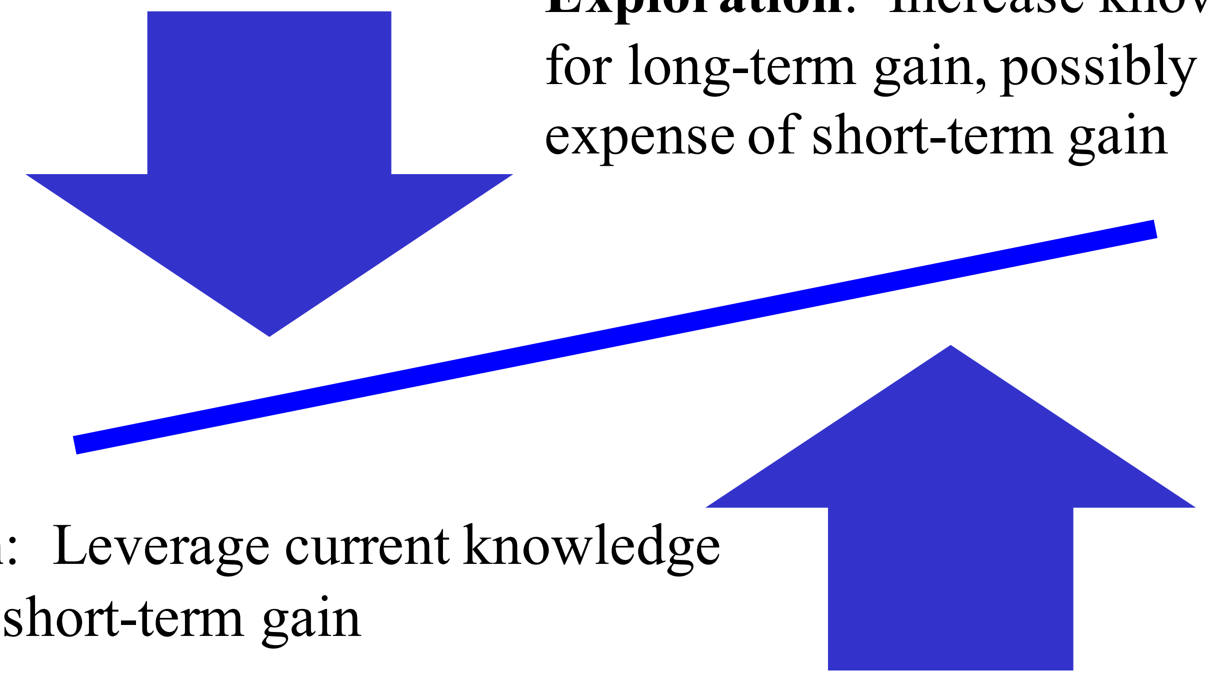
Exploration / Exploitation





Exploration / Exploitation

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



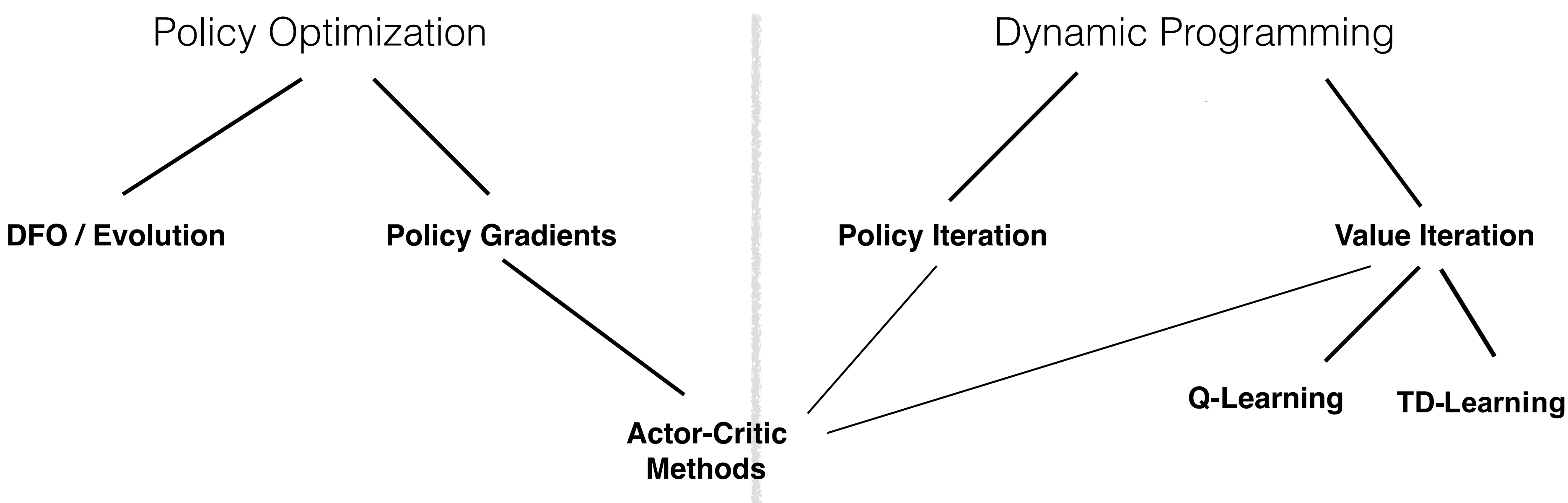
Exploitation: Leverage current knowledge to maximize short-term gain

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



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

In large state spaces: Need approximation

Challenge: finding good features

$$\hat{Q}^{\pi}(s,a) = \sum_{i=1}^{d} \theta_i \underline{\phi_i(s,a)}$$

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

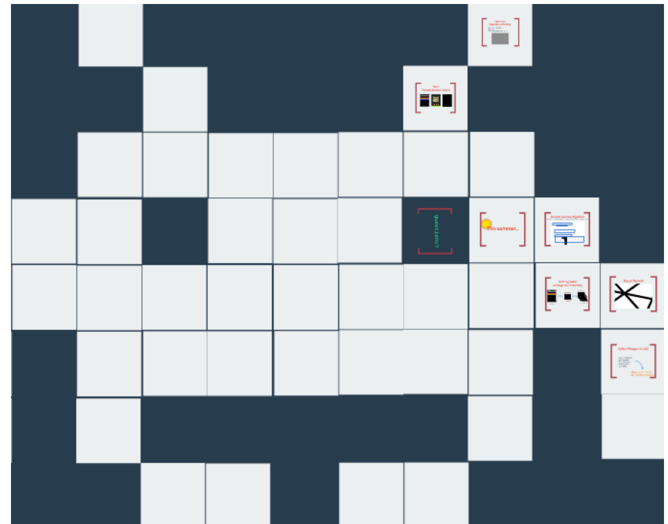
- Use **supervised learning** to estimate the **Q-function** from a batch of training data.
  - Input: $x_i := <s_i, a_i>, \ i=1..N$
  - Output: $y_i := r_i + \gamma \max_a Q_\theta(s_i',a)$
  - Loss: $\sum_i || r_i + \gamma \max_a Q_\theta(s_i',a) - Q_\theta(s_i,a_i) ||^2$

- Regression with linear function, neural network, etc.
  (Can use other functions, e.g. random forests.)

- **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

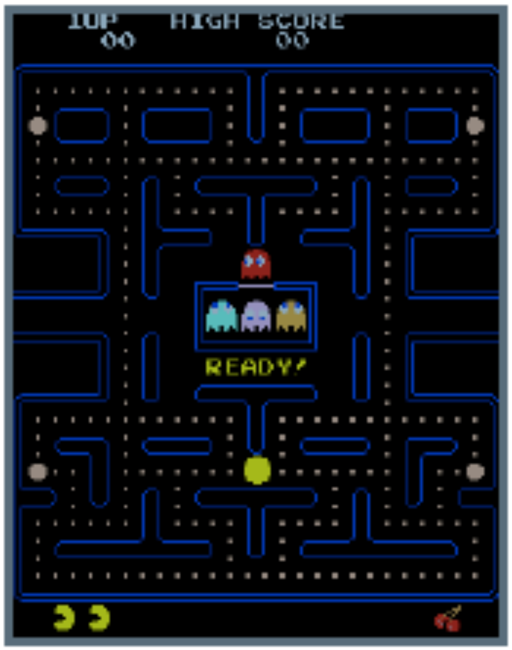


www.arcadelearningenvironment.org

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

Learning representations for RL

s



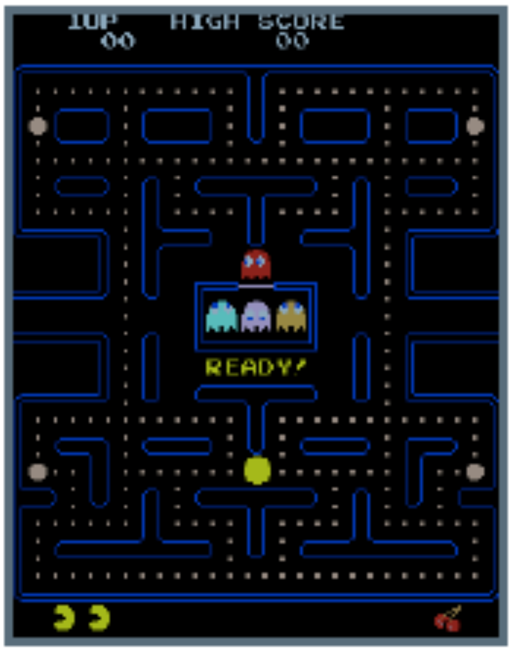
Original state

$\phi(s)?$

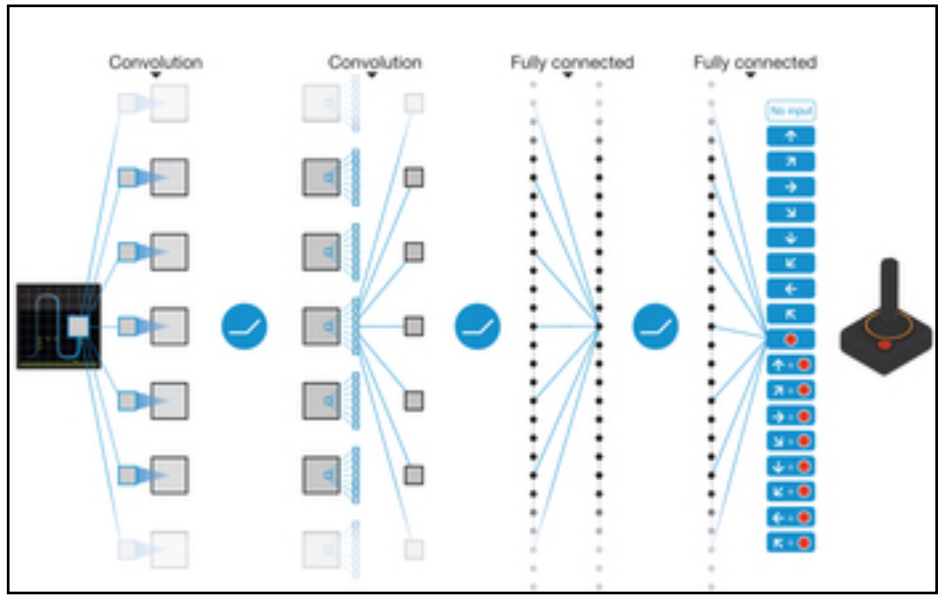
$Q_\theta(s,a)$

Deep Q-network (DQN)

s



Original state



Convolutional Neural Net

$Q_\theta(s,a)$

Trained with stochastic gradient descent.

[DeepMind: Mnih et al., 2015].

Train / Test protocol for RL

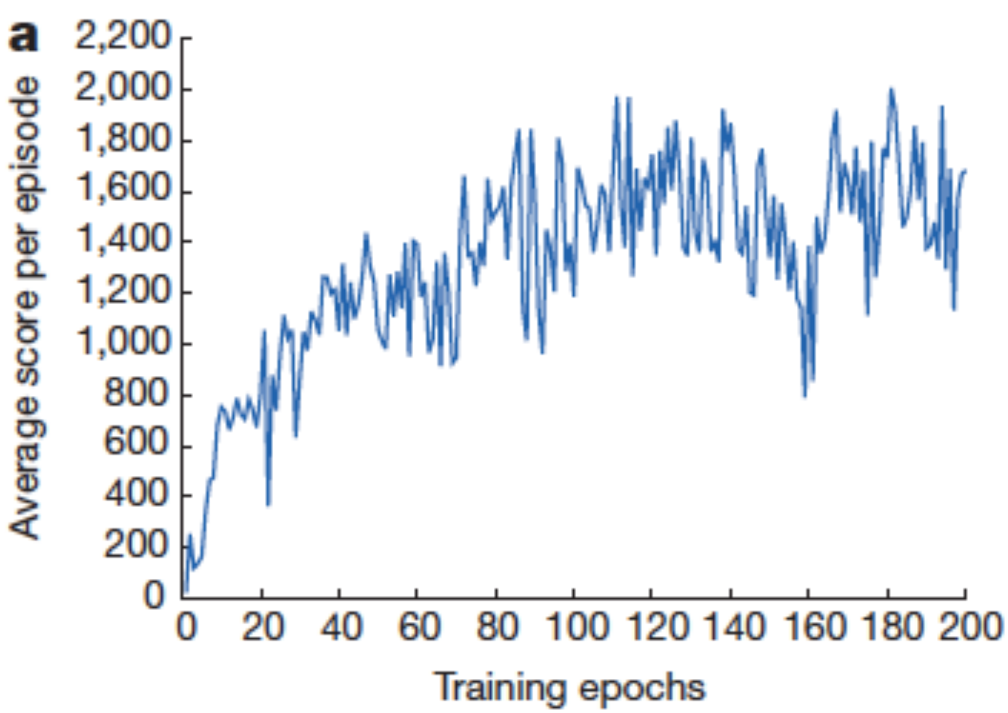
- Choose an exploration policy. Run it. Get a batch of data.
- Train your Q-function. (Stop training, fix *Q()*.)
- Use your learned Q-function to generate new trajectories. Measure the utility on these new trajectories.
- Repeat.

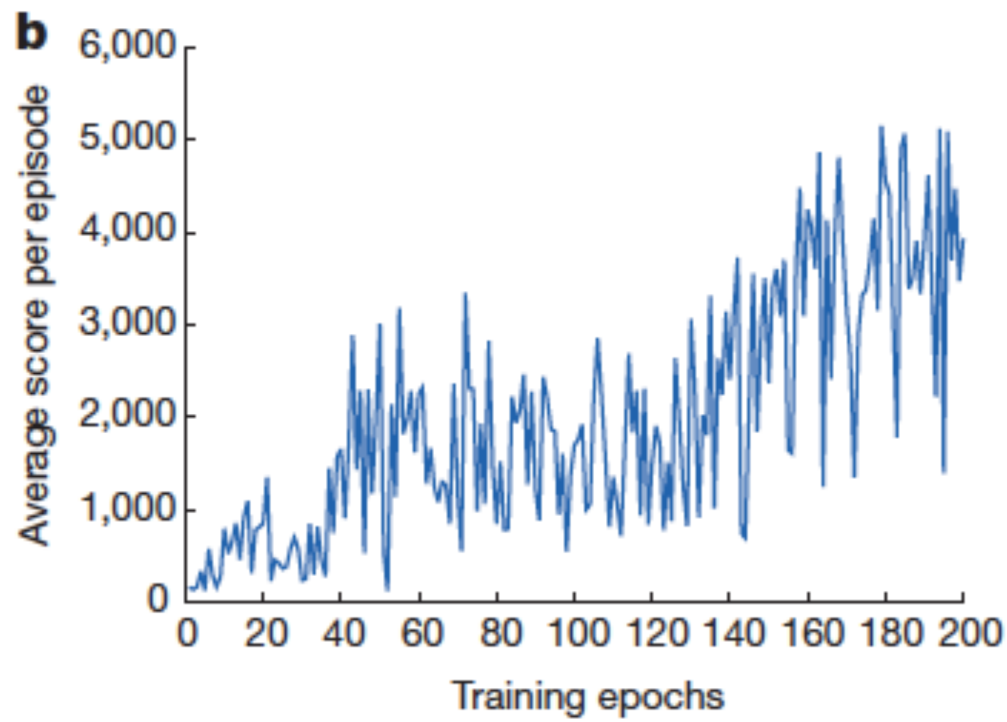
(<u>Never</u> report results for a hold-out test set.)

Training score

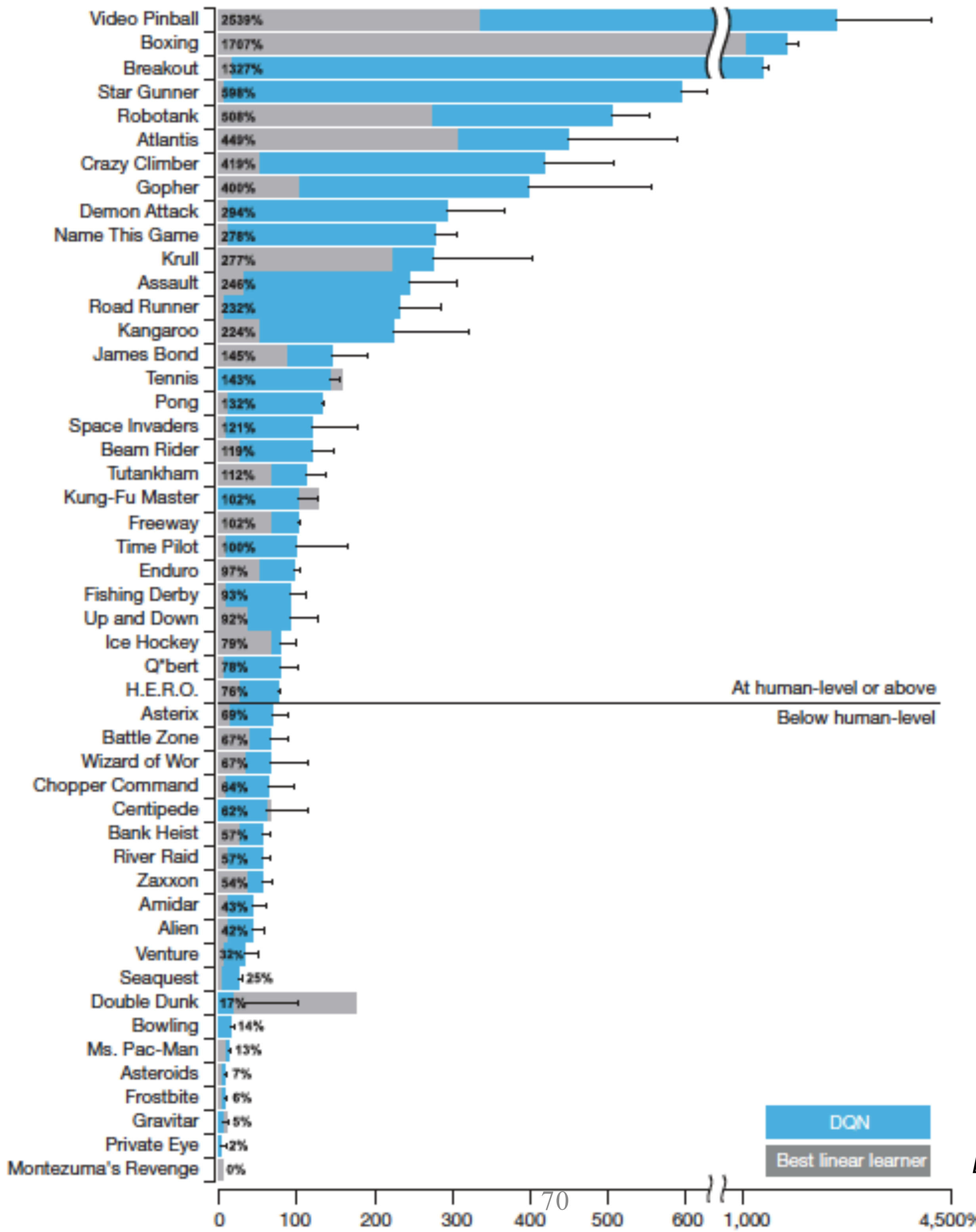
Space Invaders

Seaquest





[DeepMind: Mnih et al., 2015].



[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**

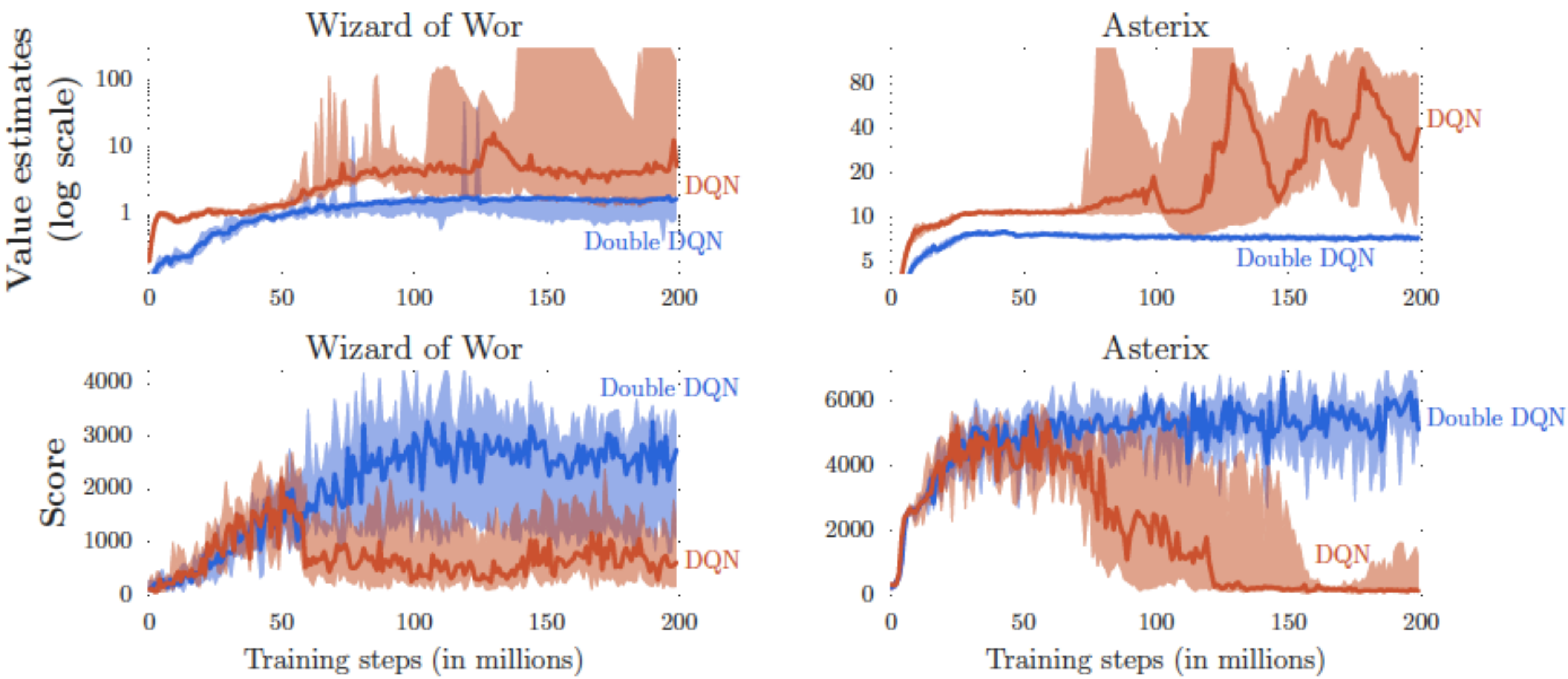
DQN: Useful tips for stability

- **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 $Q_\theta^-() \Leftarrow Q_\theta^+()$.

- **Gradient clipping**

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

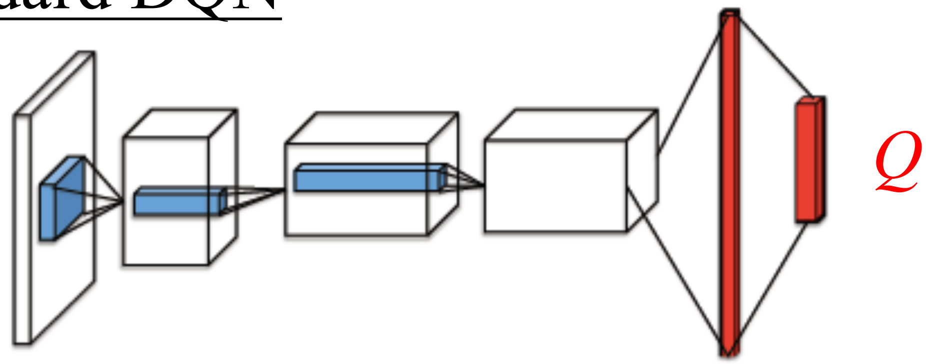
Double DQN: Avoiding positive bias



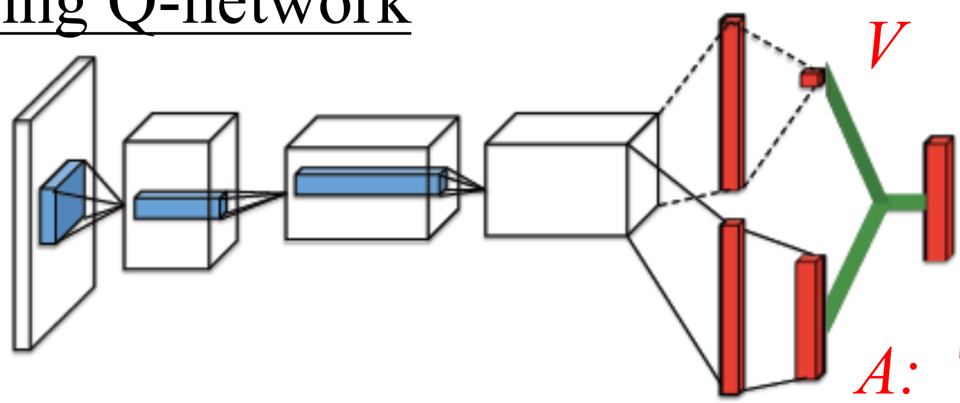
[DeepMind: van Hasselt et al., 2015].

Dueling Q-networks

Standard DQN



Dueling Q-network



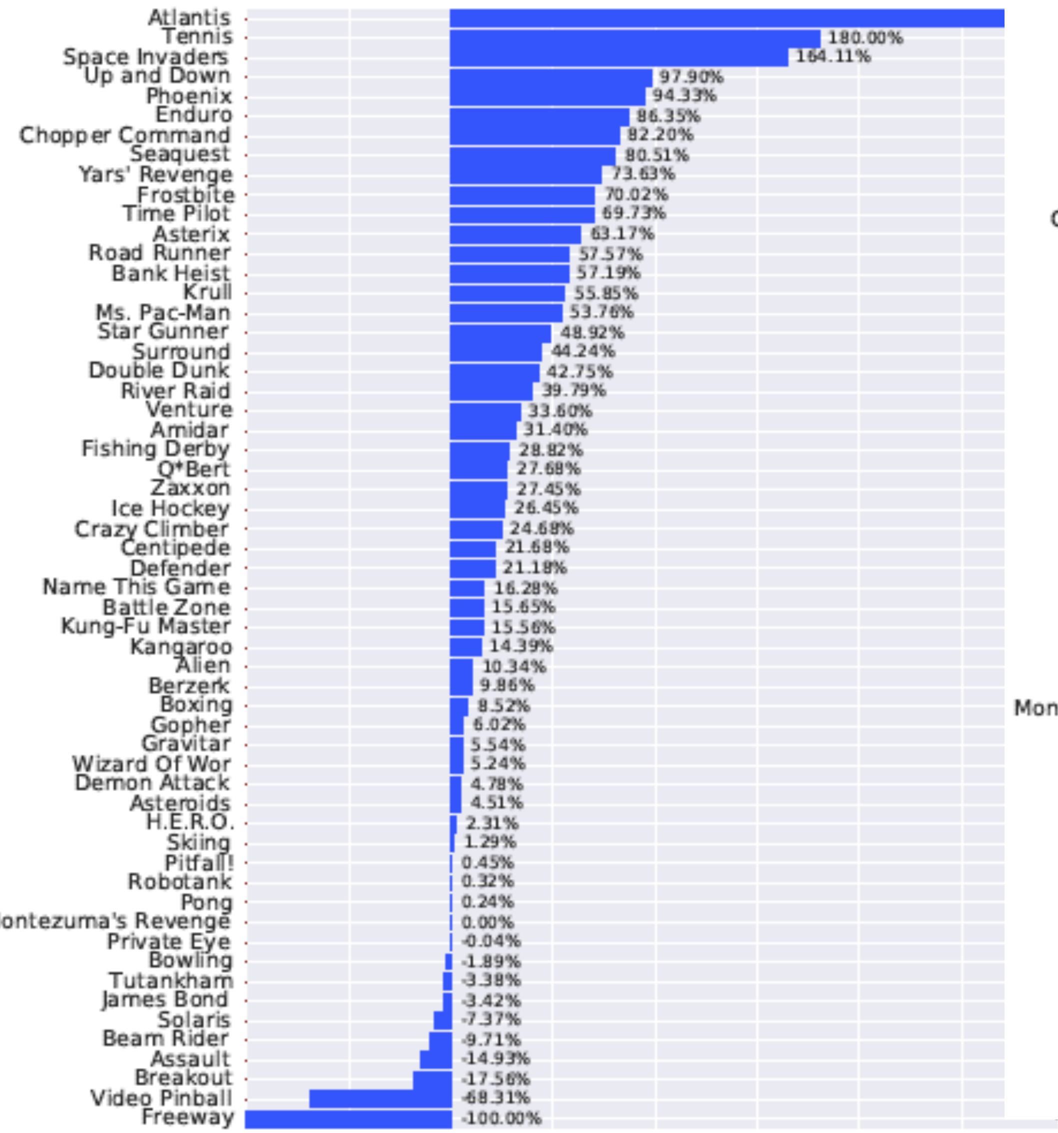
$$Q_{\theta,\alpha,\beta} = V_{\theta,\beta} + A_{\theta,\alpha}$$

A: The **advantage** function

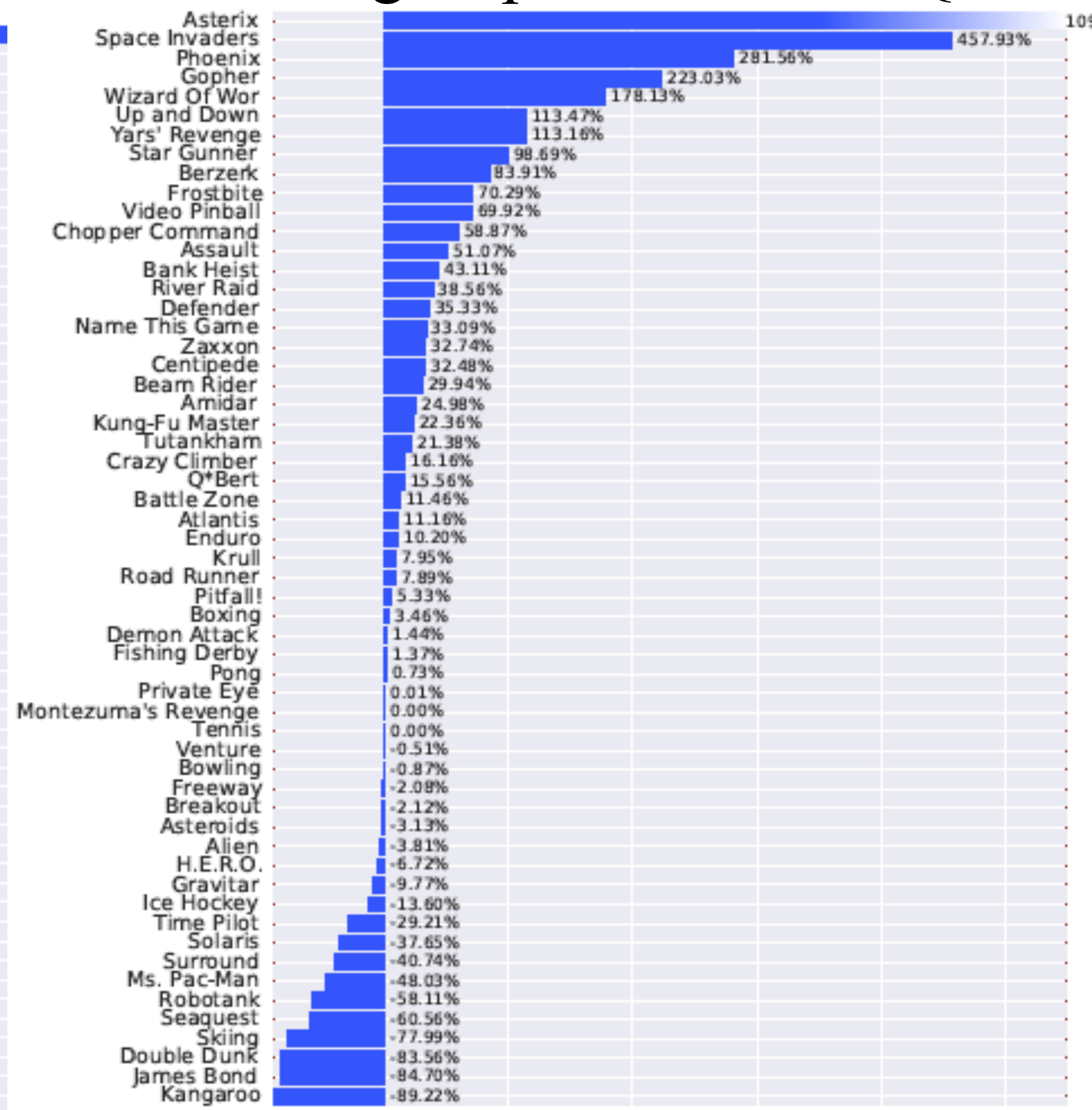
[DeepMind: Wang et al., 2016].

Dueling Q-networks

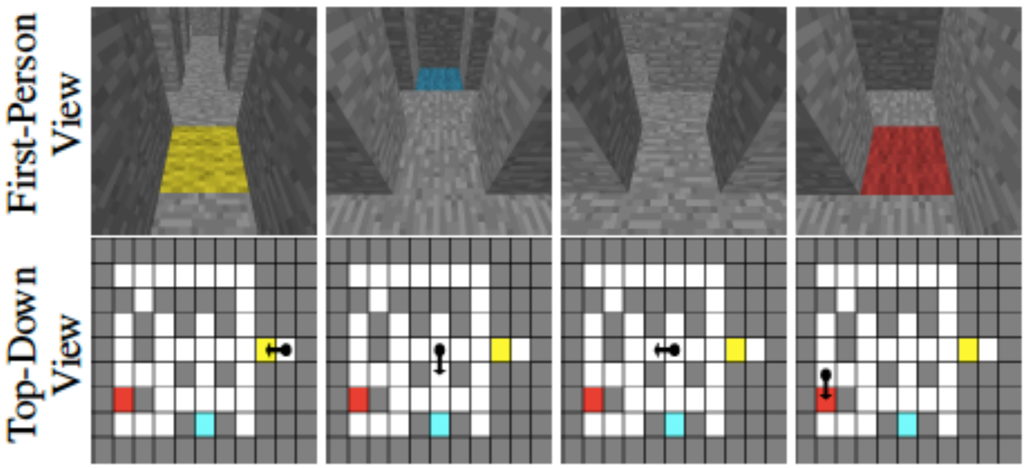
Dueling vs simple DQN



Dueling vs prioritized DDQN

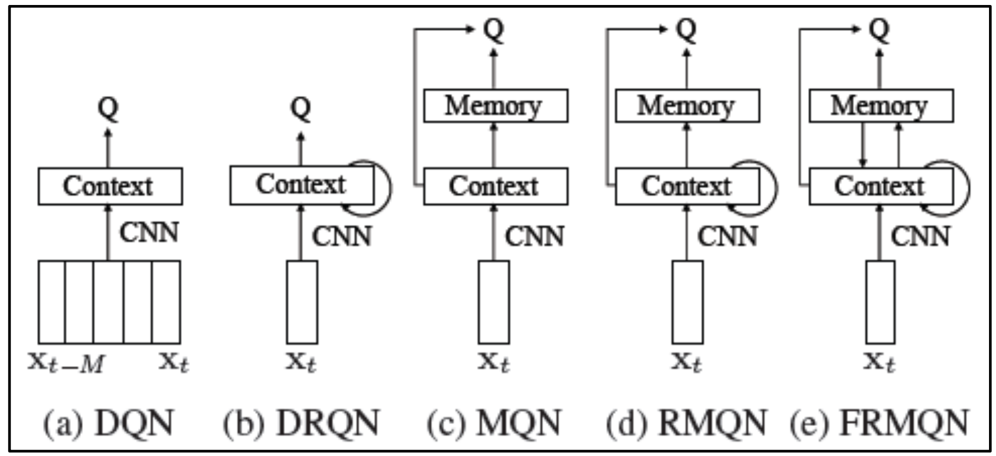


Deep RL in Minecraft



(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.



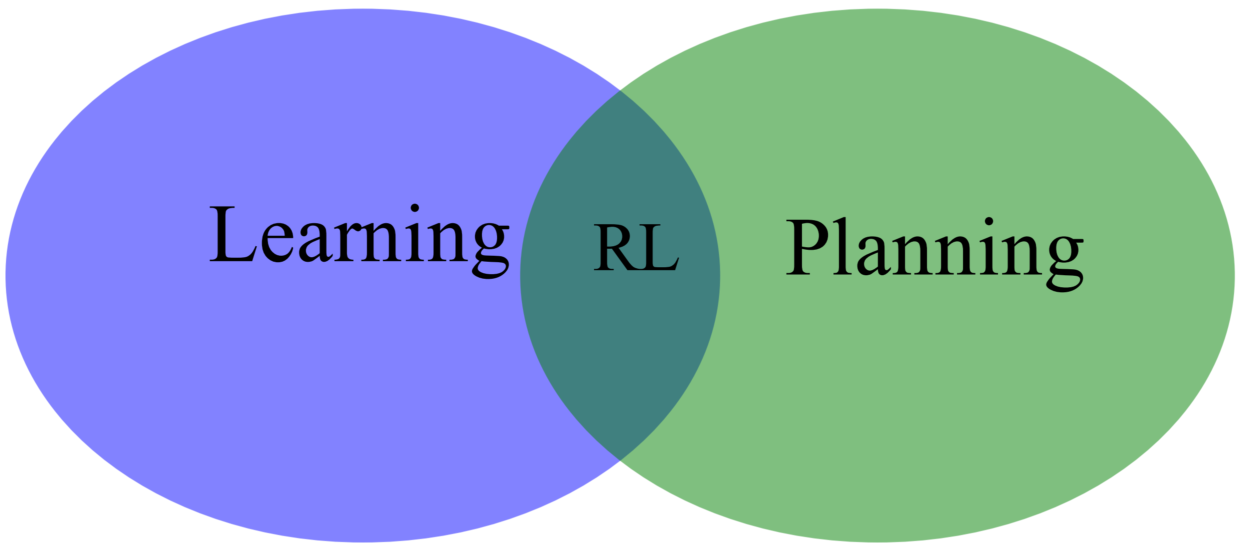
Many possible architectures,
Incl. memory and context

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

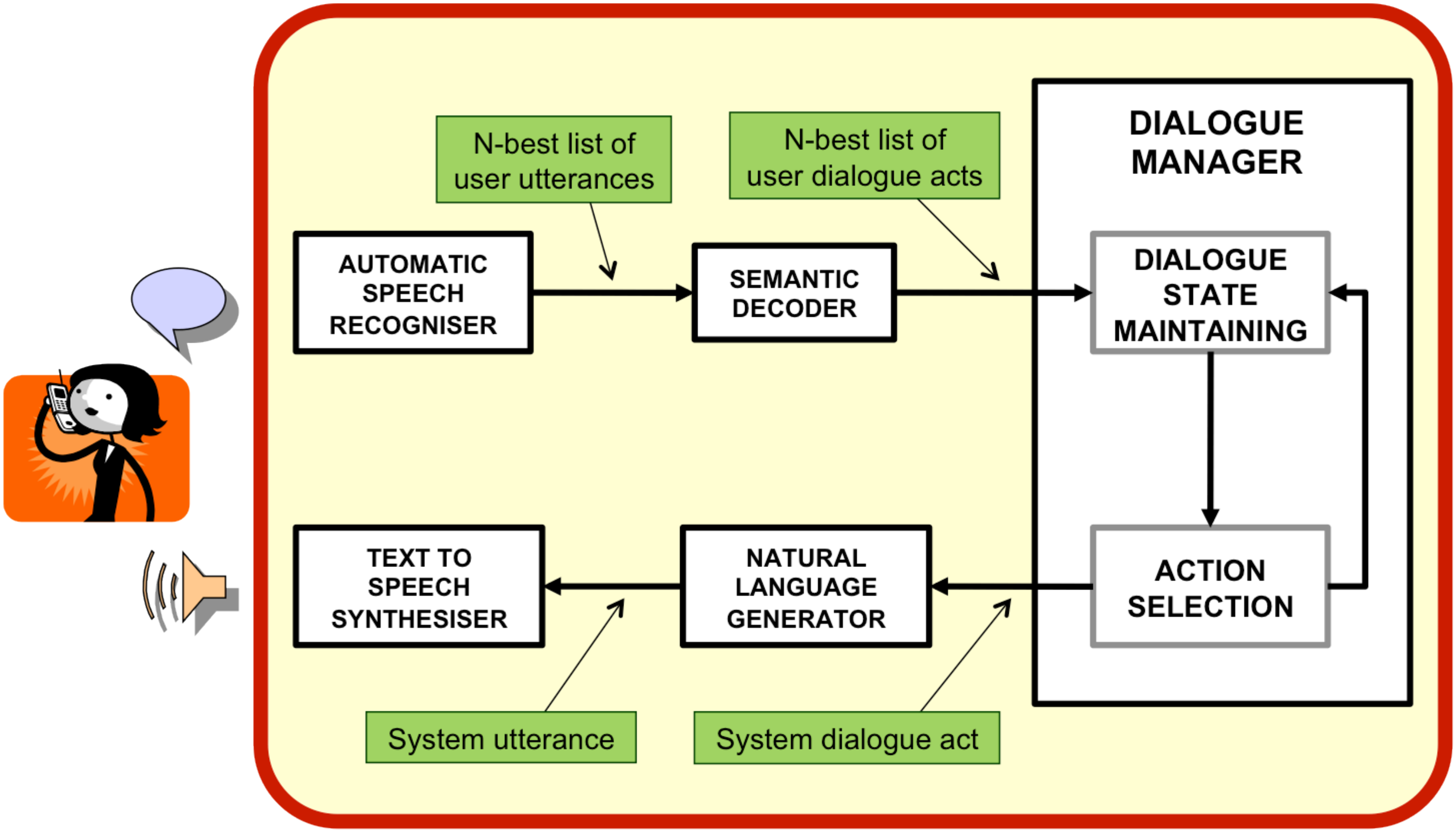
[U.Michigan: Oh et al., 2016].

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).

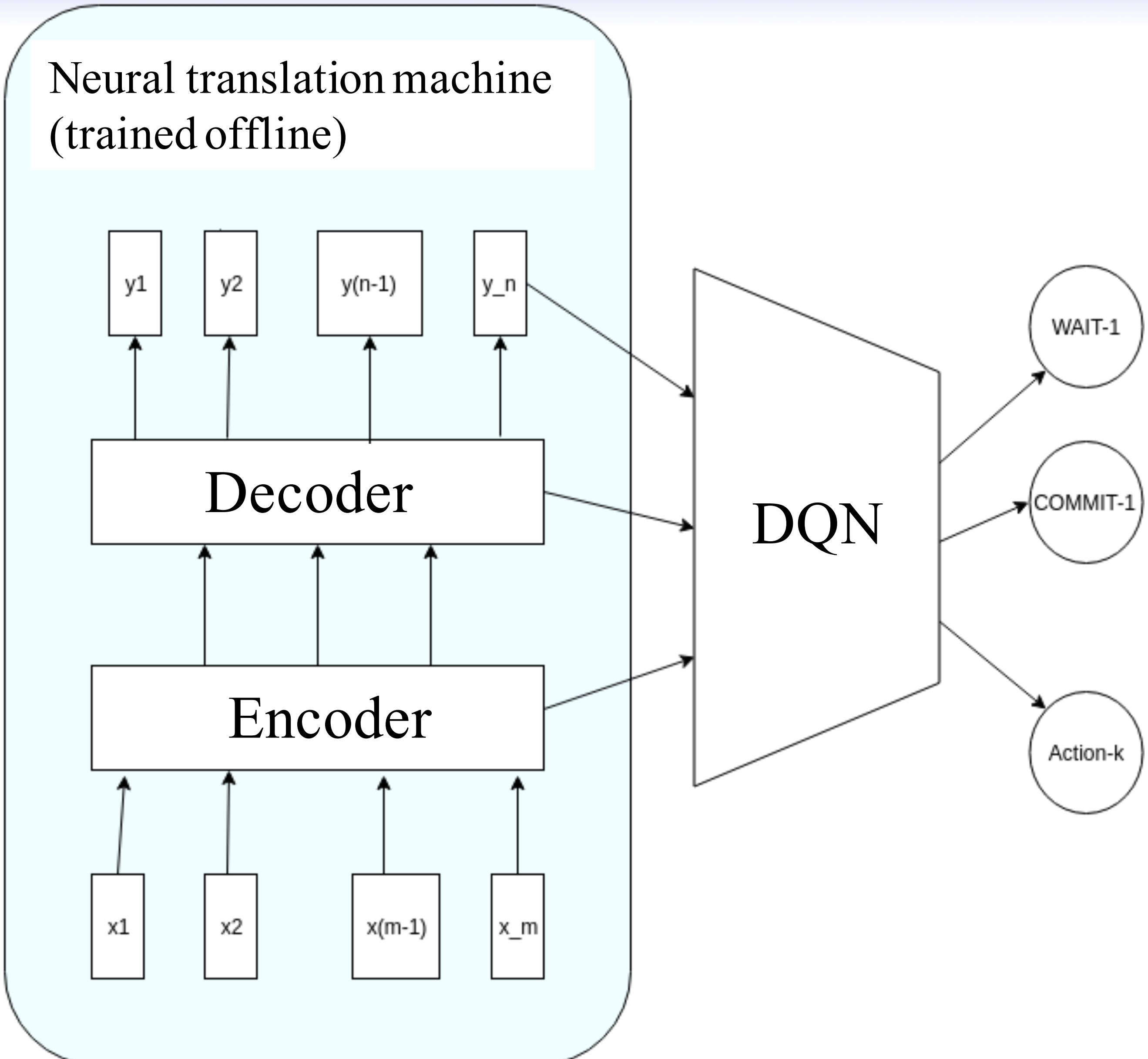


Dialogue systems



http://mi.eng.cam.ac.uk/research/dialogue/epsrc/

Neural interpretation machine



Questions?