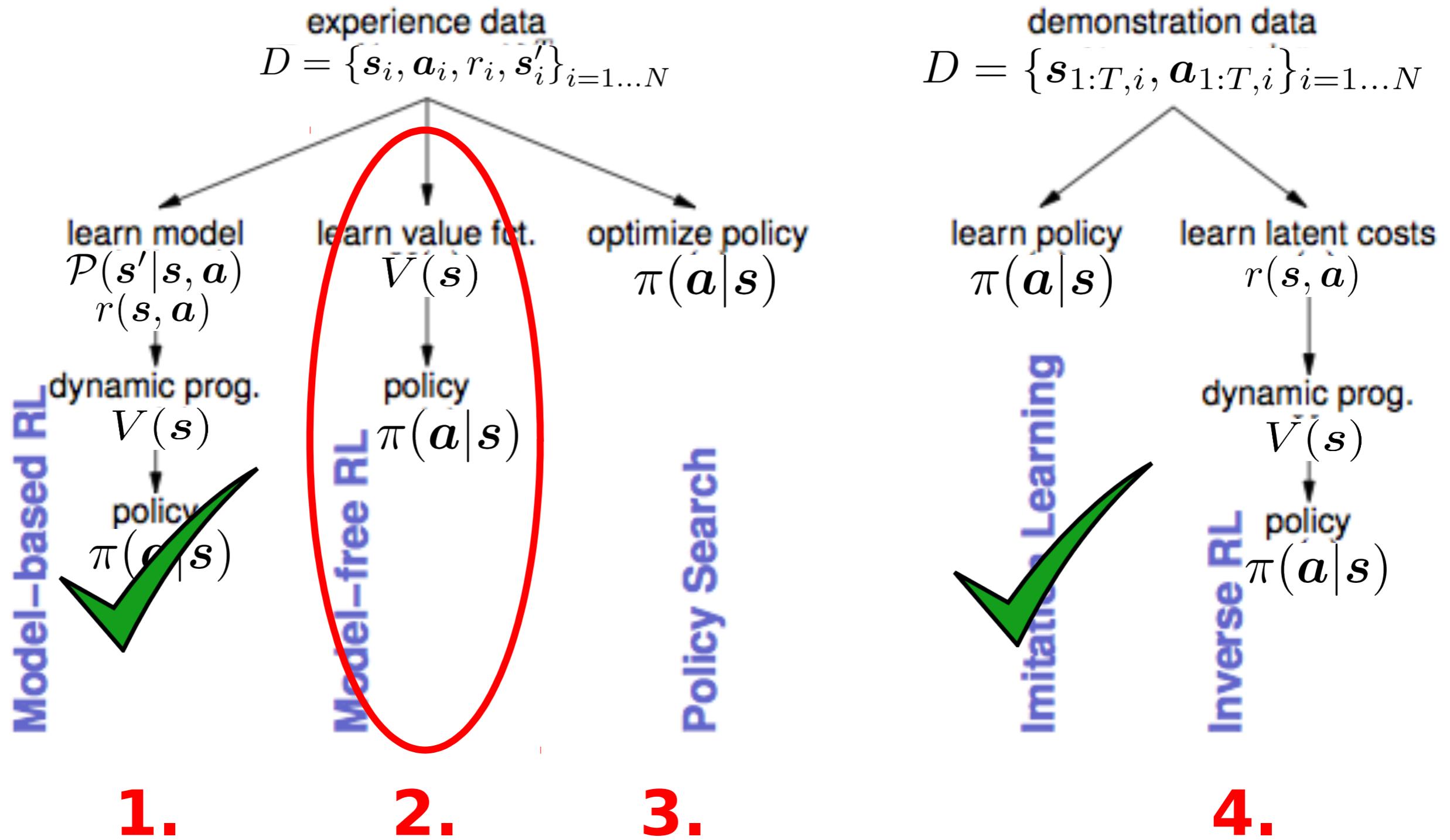


Reinforcement Learning Part 2: Value Function Methods

**Jan Peters
Gerhard Neumann**



The Bigger Picture: How to learn policies



CHRIS ATKESON Humanoids 2016

American election:

- Clinton was model-based, and used strong predictive models of who would vote and how they would vote.
- Trump did not use any models.



Purpose of this Lecture

Often, learning a good model is too hard

- The optimization inherent in optimal control is **prone to model errors**, as the controller may achieve the objective only because model errors get exploited
- Optimal control methods based on linearization of the dynamics work only for **moderately non-linear tasks**
- **Model-free** approaches are needed that do not make any assumption on the structure of the model

Classical Reinforcement Learning:

- Solve the optimal control problem by **learning the value function, not the model!**

Outline of the Lecture



- 1. Quick recap of dynamic programming**
- 2. Reinforcement Learning with Temporal Differences**
- 3. Value Function Approximation**
- 4. Batch Reinforcement Learning Methods**

Least-Squares Temporal Difference Learning

Fitted Q-Iteration

- 5. Robot Application: Robot Soccer**

Final Remarks



Markov Decision Processes (MDP)

Classical reinforcement learning is typically formulated for the infinite horizon objective

Infinite Horizon: maximize **discounted accumulated reward**

$$J_\pi = \mathbb{E}_{\mu_0, \mathcal{P}, \pi} [\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)]$$

$$0 \leq \gamma < 1$$

... discount factor

Trades-off long term vs. immediate reward

Value functions and State-Action Value Functions



Refresher: Value function and state-action value function can be computed iteratively

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left[r(s, a) + \gamma \mathbb{E}_{\mathcal{P}} [V^\pi(s')] \mid s \right] \\ &= \int \pi(a|s) \left(r(s, a) + \gamma \int \mathcal{P}(s'|s, a) V^\pi(s') ds' \right) da \end{aligned}$$

$$\begin{aligned} Q^\pi(s, a) &= r(s, a) + \gamma \mathbb{E}_{\mathcal{P}, \pi} \left[Q^\pi(s', a') \mid s, a \right] \\ &= r(s, a) + \gamma \int \mathcal{P}(s'|s, a) \int \pi(a'|s') Q^\pi(s', a') da' ds' \end{aligned}$$



Finding an optimal value function

Bellman Equation of optimality

$$V^*(s) = \max_a \left(r(s, a) + \gamma \mathbb{E}_{\mathcal{P}} [V^*(s') | s, a] \right)$$

→ erating the Bellman Equation converges to the optimal value function V^* and is called **value iteration**

Alternatively we can also **iterate Q-functions...**

$$Q^*(s, a) = r(s, a) + \gamma \mathbb{E}_{\mathcal{P}} [\max_{a'} Q^*(s', a') | s, a]$$

Outline of the Lecture



- 1. Quick recap of dynamic programming**
- 2. Reinforcement Learning with Temporal Differences**
- 3. Value Function Approximation**
- 4. Batch Reinforcement Learning Methods**

Least-Squares Temporal Difference Learning

Fitted Q-Iteration

- 5. Robot Application: Robot Soccer**

Final Remarks



Value-based Reinforcement Learning

Classical Reinforcement Learning

Updates the value function based on samples

$$\mathcal{D} = \{s_i, a_i, r_i, s'_i\}_{i=1\dots N}$$

We do not have a model and we do not want to learn it

Use the samples to update Q-function (or V-function)

Lets start simple:

Discrete states/actions ➔ Tabular Q-function



Temporal difference learning

Given a transition (s_t, a_t, r_t, s_{t+1}) , we want to update the V-function

$$V(s_t)$$

- Estimate of the current value:
 $\hat{V}(s_t) = r_t + \gamma V(s_{t+1})$
- 1-step prediction of the current value:
 $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
- 1-step prediction error (called temporal difference (TD) error)

Update current value with the temporal difference error
 $V_{\text{new}}(s_t) = V(s_t) + \alpha \delta_t = (1 - \alpha)V(s_t) + \alpha(r_t + \gamma V(s_{t+1}))$



Temporal difference learning

The **TD error**

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

compares the **one-time step lookahead prediction**

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

with the **current estimate** of the value function

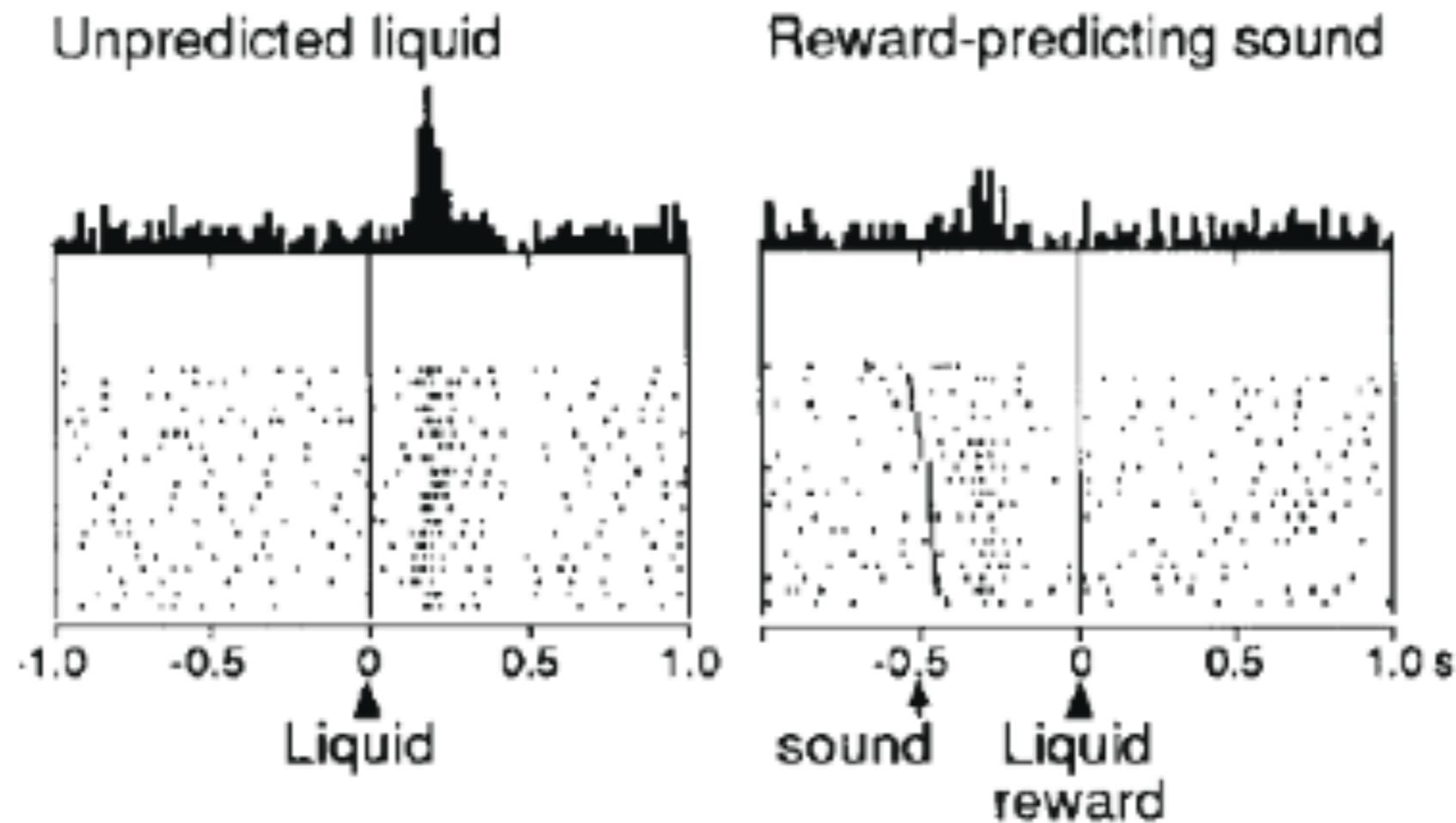
→ $\hat{V}(s_t) > V(s_t)$ then $V(s_t)$ is increased

→ $\hat{V}(s_t) < V(s_t)$ then $V(s_t)$ is decreased



Dopamine as TD-error?

Temporal difference error signals can be measured in the brain of monkeys



Monkey brains seem to have it...



Algorithmic Description of TD Learning

Init: $V_0^*(s) \leftarrow 0$

Repeat $t = t + 1$

Observe transition (s_t, a_t, r_t, s_{t+1})

Compute TD error $\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t)$

Update V-Function $V_{t+1}(s_t) = V_t(s_t) + \alpha \delta_t$

until convergence of V

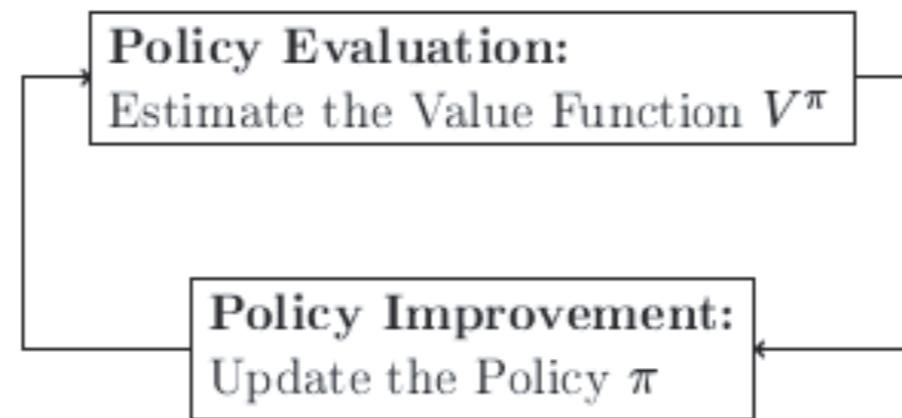
→ Used to compute Value function of behavior policy

→ Sample-based version of policy evaluation



Temporal difference learning for control

So far: Policy evaluation with TD methods



Can we also do the policy improvement step with samples?

Yes, but we need to enforce exploration!

$$\text{Epsilon-Greedy Policy } \pi(a|s) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}|, & \text{if } a = \operatorname{argmax}_{a'} Q^\pi(s, a') \\ \epsilon/|\mathcal{A}|, & \text{otherwise} \end{cases}$$

Soft-Max Policy:

$$\pi(a|s) = \frac{\exp(\beta Q(s, a))}{\sum_{a'} \exp(\beta Q(s, a'))}$$

→ o not always take greedy action



Temporal difference learning for control

Update equations for learning the Q-function

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \delta_t, \quad \delta_t = r_t + \gamma Q_t(s_{t+1}, a?) - Q_t(s_t, a_t)$$

Two different methods to estimate

Q-learning: $a? = \operatorname{argmax}_a Q_t(s_{t+1}, a)$

Estimates Q-function of optimal policy

Off-policy samples: $a? \neq a_{t+1}$

SARSA: $a? = a_t$, where $a_{t+1} \sim \pi(a|s_{t+1})$

Estimates Q-function of exploration policy

On-policy samples

1 **Note:** The policy for generating the actions depends on the Q-function → on-stationary policy

Outline of the Lecture



- 1. Quick recap of dynamic programming**
- 2. Reinforcement Learning with Temporal Differences**
- 3. Value Function Approximation**
- 4. Batch Reinforcement Learning Methods**

Least-Squares Temporal Difference Learning

Fitted Q-Iteration

- 5. Robot Application: Robot Soccer**

Final Remarks



Approximating the Value Function

In the continuous case, we need to approximate the V-function (except for LQR)

Lets keep it simple, we use **a linear model** to represent the V-function

$$V^\pi(s) \approx V_\omega(s) = \phi^T(s)\omega$$

How can we find the parameters ω ?



Again with **Temporal Difference Learning**



TD-learning with Function Approximation

Derivation:

Use the **recursive definition of V-function**:

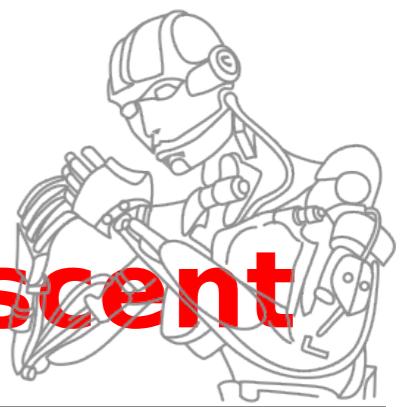
$$\text{MSE}(\omega) \approx \text{MSE}_{\text{BS}}(\omega) = 1/N \sum_{i=1}^N \left(\hat{V}^\pi(s_i) - V_\omega(s_i) \right)^2$$

with $\hat{V}^\pi(s) = \mathbb{E}_\pi \left[r(s, a) + \mathbb{E}_{\mathcal{P}} [V_{\omega_{\text{old}}}(s') | s, a] \right]$

 **Bootstrapping (BS):** Use the old approximation to get the target values for a new approximation

How can we **minimize** this function ?

Lets use **stochastic gradient descent**



Refresher: Stochastic Gradient Descent

Consider an **expected error function**,

$$E_{\omega} = \mathbb{E}_p[e_{\omega}(x)] \approx 1/N \sum_{i=1}^N e_{\omega}(x_i), \quad x_i \sim p(x)$$

We can find a local minimum of E by **Gradient descent**:

$$\omega_{k+1} = \omega_k - \alpha_k \frac{dE_{\omega}}{d\omega} = \omega_k - \alpha_k \sum_{i=1}^N \frac{de_{\omega}(x_i)}{d\omega}$$

Stochastic Gradient Descent does the gradient update already after a **single sample**

$$\omega_{k+1} = \omega_k - \alpha_k \frac{de_{\omega}(x_k)}{d\omega}$$

Converges under the stochastic approximation conditions

$$\sum_{k=1}^{\infty} \alpha_k = \infty, \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$



Temporal difference learning

Stochastic gradient descent on our error function MSE_{BS}

$$MSE_{BS,t}(\omega) = \frac{1}{N} \sum_{i=1}^N \left(\hat{V}(s_t) - V_\omega(s_i) \right)^2$$

$$= \frac{1}{N} \sum_{i=1}^N \left(r_i + \gamma V_{\omega_t}(s'_i) - V_\omega(s_i) \right)^2$$

Update rule (for current time step t, $V_\omega(s) = \phi^T(s)\omega$)

$$\omega_{t+1} = \omega_t + \alpha_t \left. \frac{dMSE_{BS}}{d\omega} \right|_{\omega=\omega_t}$$

$$\begin{aligned} \omega_{t+1} &= \omega_t + \alpha \left(r(s_t, a_t) + \gamma V_{\omega_t}(s_{t+1}) - V_{\omega_t}(s_t) \right) \phi^T(s_t) \\ &= \omega_t + \alpha \delta_t \phi^T(s_t) \end{aligned}$$

with $\delta_t = r(s_t, a_t) + \gamma V_{\omega_t}(s_{t+1}) - V_{\omega_t}(s_t)$



Temporal difference learning

TD with function approximation

$$\omega_t = \omega_t + \alpha \delta_t \phi^T(s_t)$$

Difference to discrete algorithm:

- TD-error is correlated with the feature vector
- Equivalent if tabular feature coding is used, $\phi(e_{s_i}) = e_i$

Similar update rules can be obtained for SARSA and Q-learning

$$\omega_{t+1} = \omega_t + \alpha \left(r(s_t, a_t) + \gamma Q_{\omega_t}(s_{t+1}, a?) - Q_{\omega_t}(s_t, a_t) \right) \phi^T(s_t, a_t)$$

$$Q_{\omega}(s, a) \approx \phi^T(s, a)\omega$$

where



Temporal difference learning

Some remarks on temporal difference learning:

- Its **not a proper** stochastic gradient descent!!
- **Why?** Target values $\hat{V}^\pi(s)$ **change after each parameter update!**
We ignore the fact that $\hat{V}^\pi(s)$ also depends on ω
- **Side note:** This „ignorance“ actually introduces a bias in our optimization, such that we are optimizing a different objective than the *MSE*
- In certain cases, we also get **divergence** (e.g. off-policy samples)
 - TD-learning is very fast in terms of computation time $O(\# \text{features})$, but not data efficient with **each sample is just used once!**



Sucessful examples

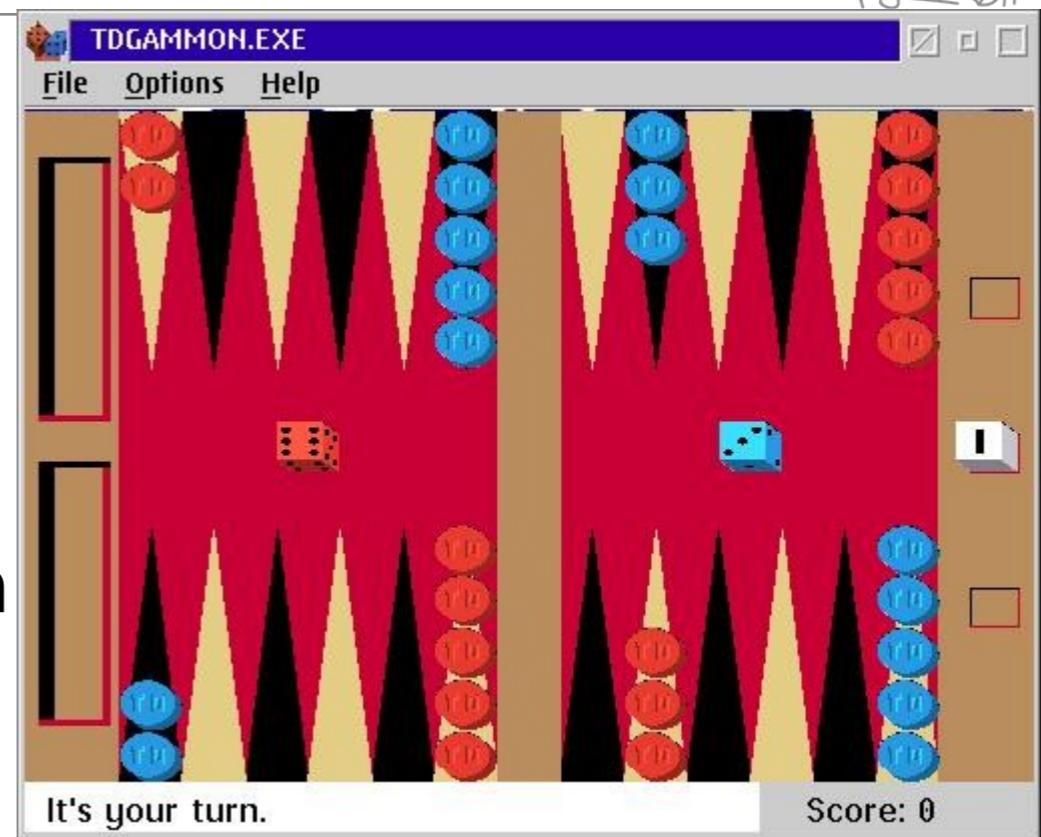
Linear function approximation

Tetris, Go

Non-linear function approximation

TD Gammon (Worldchampion level)

Atari Games (learning from raw pixel input)





Outline of the Lecture

- 1. Quick recap of dynamic programming**
- 2. Value function approximation**
- 3. Reinforcement Learning with Temporal Differences**
- 4. Batch Reinforcement Learning Methods**

Least-Squares Temporal Difference Learning

Fitted Q-Iteration

- 5. Robot Application: Robot Soccer**

Final Remarks



Batch-Mode Reinforcement Learning

Online methods are typically **data-inefficient** as they use each data point only once

$$D = \left\{ s_i, a_i, r_i, s'_i \right\}_{i=1 \dots N}$$

Can we **re-use the whole „batch“** of data to **increase data-efficiency?**

- **Least-Squares Temporal Difference (LSTD) Learning**

→ **Fitted Q-Iteration**

Computationally **much more expensive** than TD-learning!



Least-Squares Temporal Difference (LSTD)

Lets **minimize the bootstrapped MSE** objective (MSE_{BS})

$$\begin{aligned} MSE_{BS} &= 1/N \sum_{i=1}^N \left(r(\mathbf{s}_i, \mathbf{a}_i) + \gamma V_{\omega_{\text{old}}}(\mathbf{s}'_i) - V_{\omega}(\mathbf{s}_i) \right)^2 \\ &= 1/N \sum_{i=1}^N \left(r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \boldsymbol{\phi}^T(\mathbf{s}'_i) \boldsymbol{\omega}_{\text{old}} - \boldsymbol{\phi}^T(\mathbf{s}_i) \boldsymbol{\omega} \right)^2 \end{aligned}$$

Least-Squares Solution:

$$\boldsymbol{\omega} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T (\mathbf{R} + \gamma \boldsymbol{\Phi}' \boldsymbol{\omega}_{\text{old}})$$

with $\boldsymbol{\Phi} = [\boldsymbol{\phi}(\mathbf{s}_1), \boldsymbol{\phi}(\mathbf{s}_2), \dots, \boldsymbol{\phi}(\mathbf{s}_N)]^T$

$$\boldsymbol{\Phi}' = [\boldsymbol{\phi}(\mathbf{s}'_1), \boldsymbol{\phi}(\mathbf{s}'_2), \dots, \boldsymbol{\phi}(\mathbf{s}'_N)]^T$$



Least-Squares Temporal Difference (LSTD)

Least-Squares Solution:

$$\omega = (\Phi^T \Phi)^{-1} \Phi^T (R + \gamma \Phi' \omega_{\text{old}})$$

Fixed Point: In case of convergence, we want $\omega_{\text{old}} = \omega$

$$\begin{aligned}\omega &= (\Phi^T \Phi)^{-1} \Phi^T (R + \gamma \Phi' \omega) \\ (I - \gamma(\Phi^T \Phi)^{-1} \Phi^T \Phi') \omega &= (\Phi^T \Phi)^{-1} \Phi^T R \\ (\Phi^T \Phi)^{-1} \Phi^T (\Phi - \gamma \Phi') \omega &= (\Phi^T \Phi)^{-1} \Phi^T R \\ \Phi^T (\Phi - \gamma \Phi') \omega &= \Phi^T R\end{aligned}$$

$$\omega = (\Phi^T (\Phi - \gamma \Phi'))^{-1} \Phi^T R$$



Least-Squares Temporal Difference (LSTD)

LSTD solution:

$$\omega = (\Phi^T(\Phi - \gamma\Phi'))^{-1} \Phi^T R$$

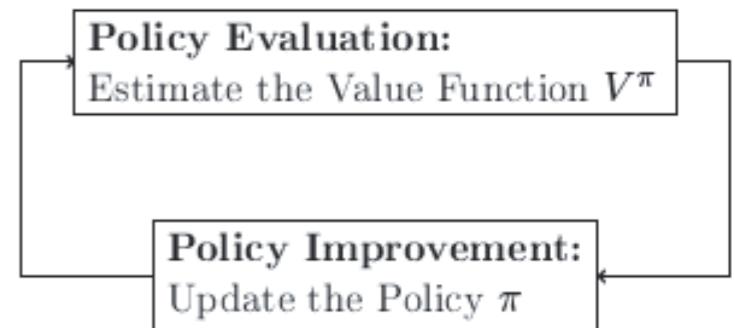
Same solution as convergence point of **TD-learning**

One shot! **No iterations** necessary for policy evaluation

LSQ: Adaptation for learning the Q-function

$$\Phi = [\phi(s_1, a_1), \phi(s_2, a_2), \dots, \phi(s_N, a_N)]^T$$

$$\Phi' = [\phi(s_2, a_2), \phi(s_3, a_3), \dots, \phi(s_{N+1}, a_{N+1})]^T$$



→ used for **Least-Squares Policy Iteration (LSPI)**

Lagoudakis and Parr, **Least-Squares Policy Iteration**, JMLR

Learning to Ride a Bicycle



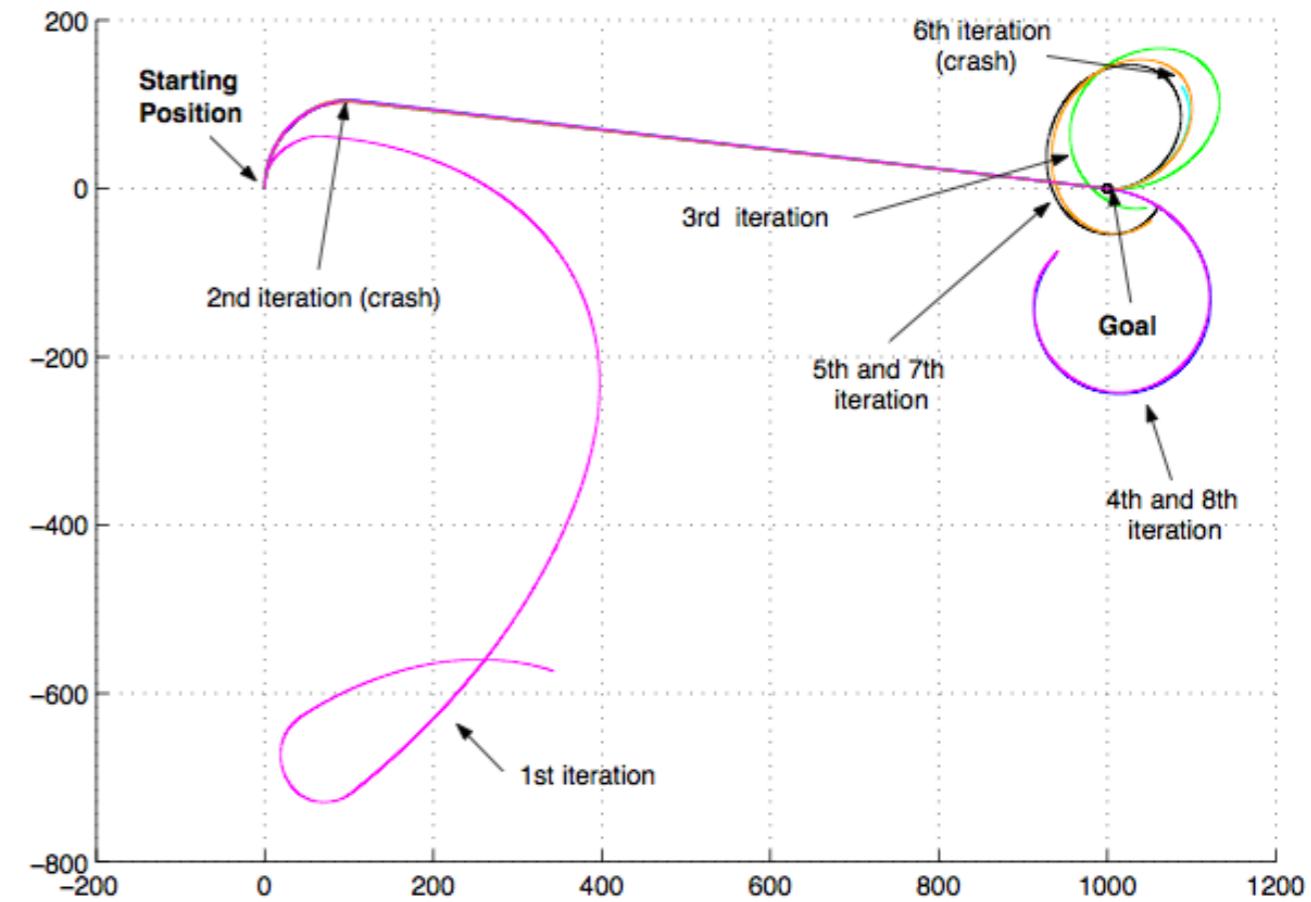
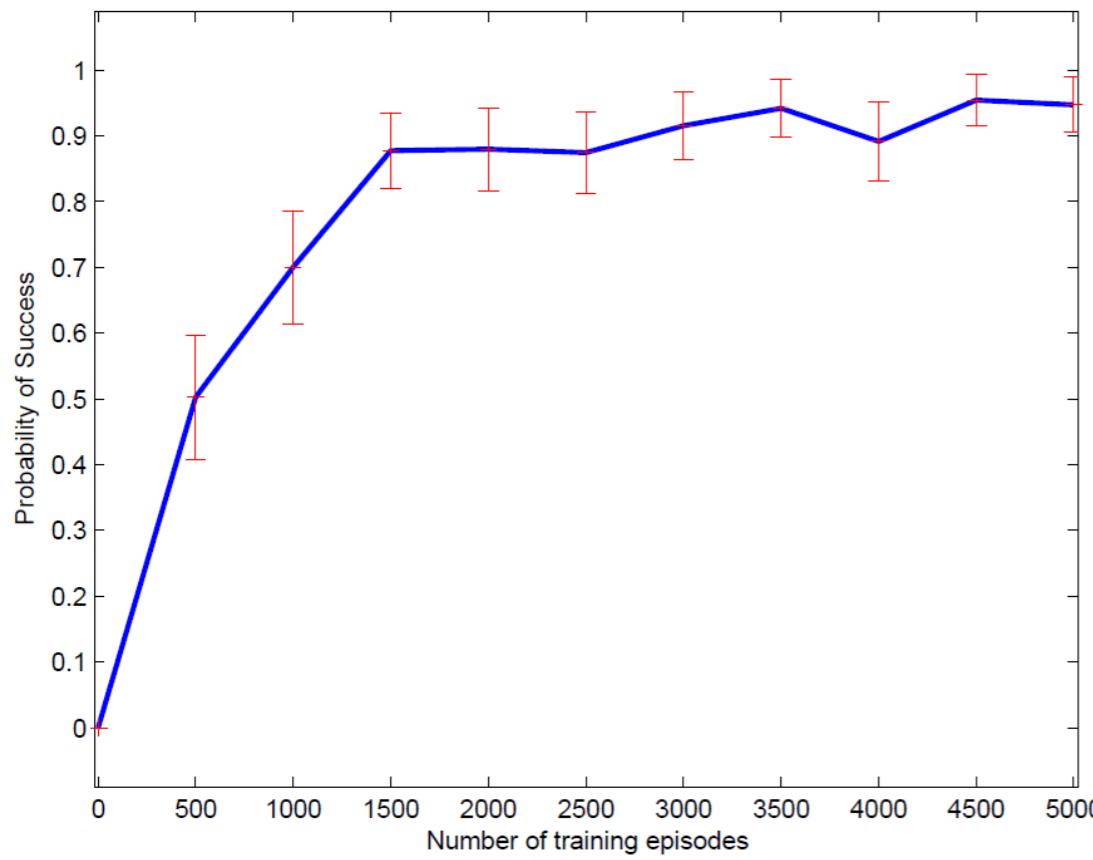
State space: $[\theta, \dot{\theta}, \omega, \dot{\omega}, \ddot{\omega}, \psi]$

θ : angle of handlebar, ω : vertical angle of bike, ψ : angle to goal

Action space: 5 discrete actions (torque applied to handle, displacement of rider)

Feature space: 20 basis functions...

$$(1, \omega, \dot{\omega}, \omega^2, \dot{\omega}^2, \omega\dot{\omega}, \theta, \dot{\theta}, \theta^2, \dot{\theta}^2, \theta\dot{\theta}, \omega\theta, \omega^2\theta, \psi, \dot{\psi}^2, \psi\dot{\psi}, \bar{\psi}, \dot{\bar{\psi}}^2, \bar{\psi}\dot{\bar{\psi}})^T$$





Fitted Q-iteration

In Batch-Mode RL it is also much easier to use **non-linear function approximators**

- Many of them **only exists in the batch setup**, e.g. regression trees
- **No catastrophic forgetting**, e.g., for neural networks.
- Strong divergence problems, fixed for Neural Networks by ensuring that there is a goal state where the Q-Function value is always zero (see Lange et al. below).

Fitted Q-iteration uses non-linear function approximators for **approximate value iteration**.

Ernsts, Geurts and Wehenkel, *Tree-Based Batch Mode Reinforcement Learning*, JMLR 2005
Lange, Gabel and Riedmiller. *Batch Reinforcement Learning*, *Reinforcement Learning: State of the Art*



Fitted Q-iteration

Given: Dataset $D = \{s_i, a_i, r_i, s'_i\}_{i=1\dots N}$

Algorithm:

Initialize $Q^{[0]}(s, a)$ **Input Data:**

for $k = 1$ to L

 Generate target values:

$$X = \begin{bmatrix} s_1^T & a_1^T \\ \vdots & \vdots \\ s_N^T & a_N^T \end{bmatrix}$$

$$\tilde{q}_i^{[k]} = r_i + \gamma \max_{a'} Q^{[k-1]}(s'_i, a')$$

 Learn new Q-function:

$$Q^{[k]}(s, a) \leftarrow \text{Regress}(X, \tilde{q}^{[k]})$$

end

→ Like Value-Iteration, but we use **supervised learning methods** to approximate the Q-function at each iteration k



Fitted Q-iteration

Some Remarks:

- Regression does the **expectation** for us

$$Q^{[k]}(s, a) \approx \mathbb{E}_{\mathcal{P}} [r(s, a) + \gamma \max_{a'} Q^{[k-1]}(s', a')]$$

- The max operator is still **hard to solve for continuous action spaces**

For continuous actions, see: Neumann and Peters, *Fitted Q-iteration by Advantage weighted regression*, NIPS, 2008

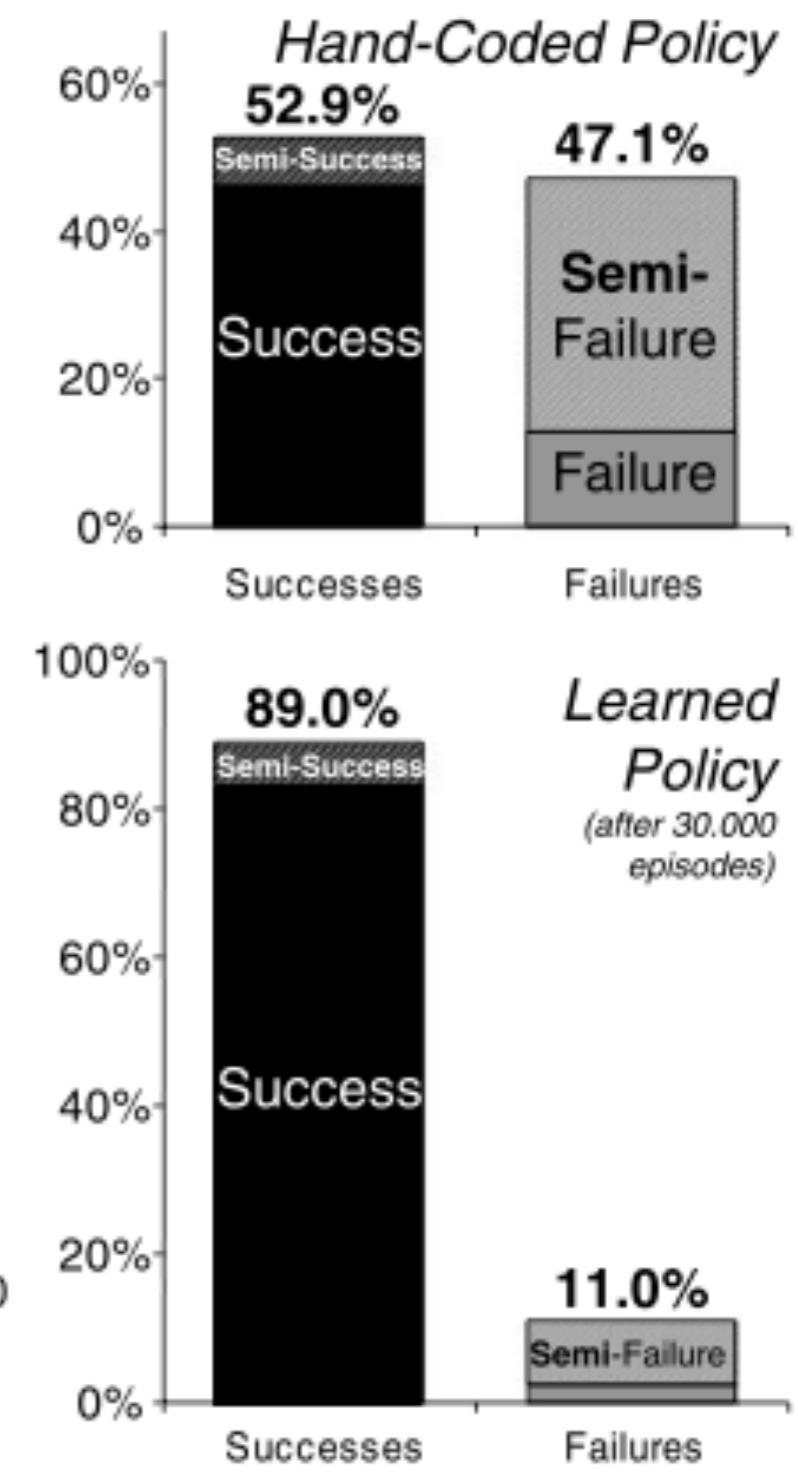


Case Study I: Learning Defense

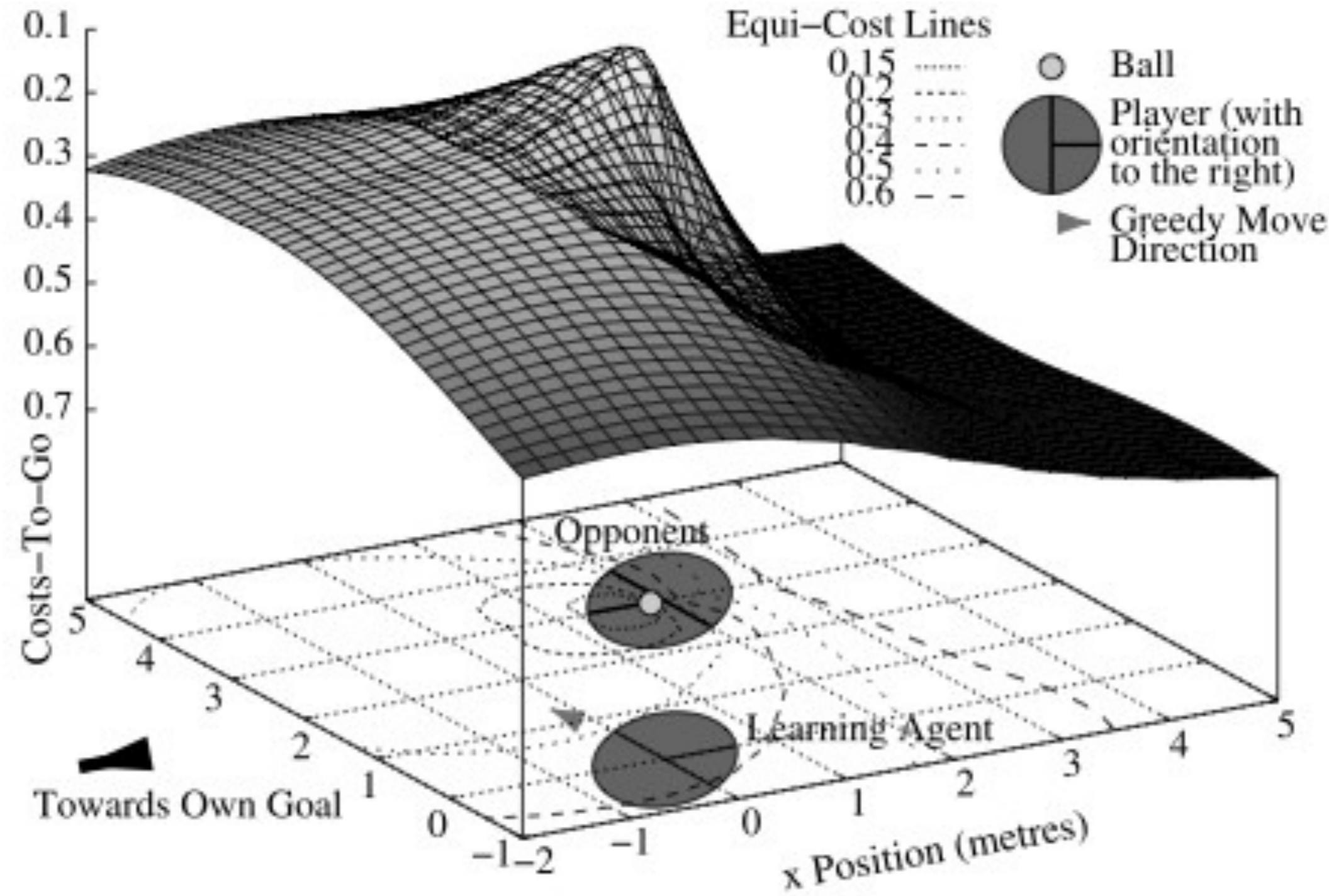


Within the RL framework, we model the ADB learning task as a terminal state problem with both terminal goal S^+ and failure states S^- . Intermediate steps are punished by constant costs of $c = 0.05$, whereas $J(s) = 0.0$ for $s \in S^+$ and $J(s) = 1.0$ for $s \in S^-$ by definition (cf. Eq. 8).

Success

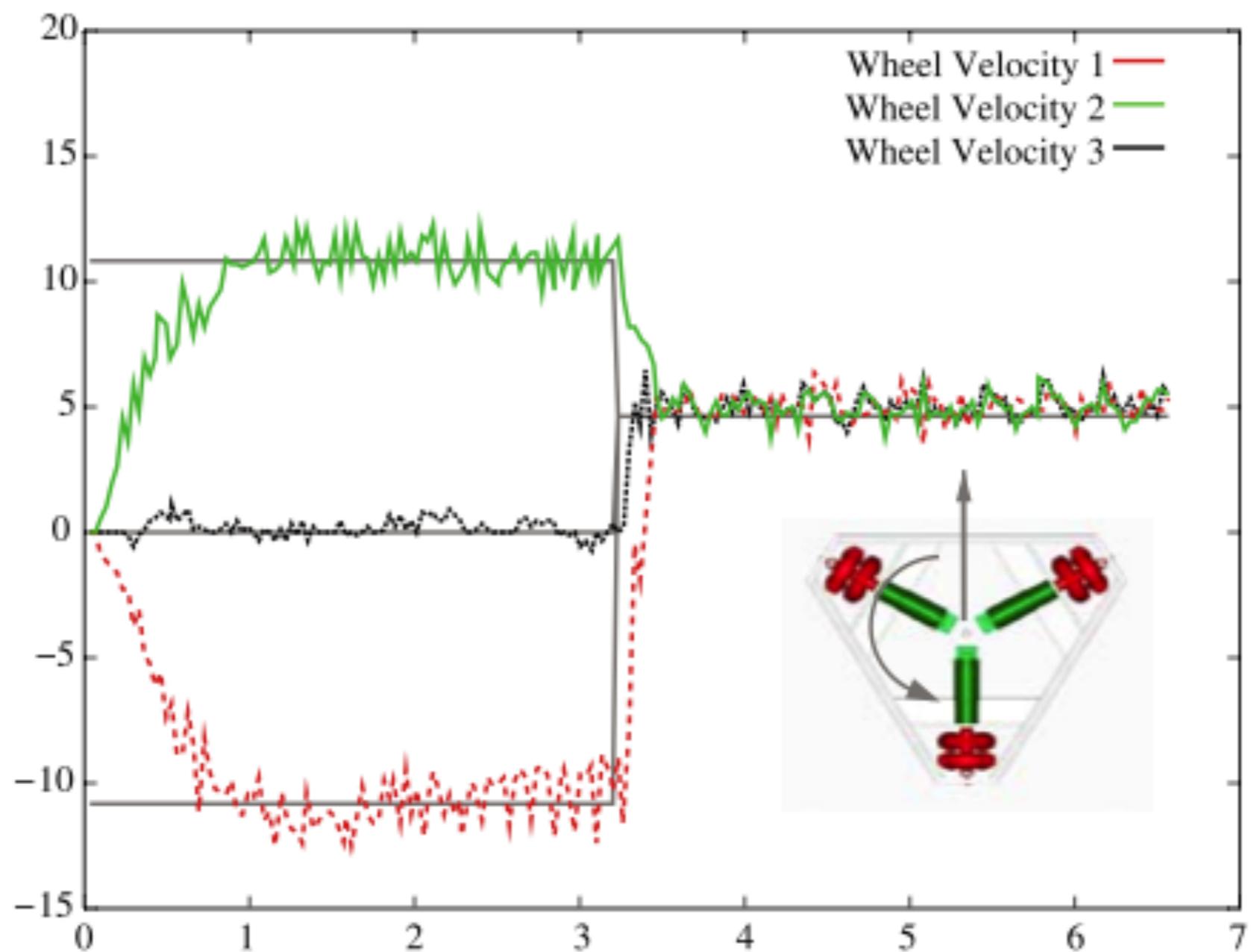


Dueling Behavior





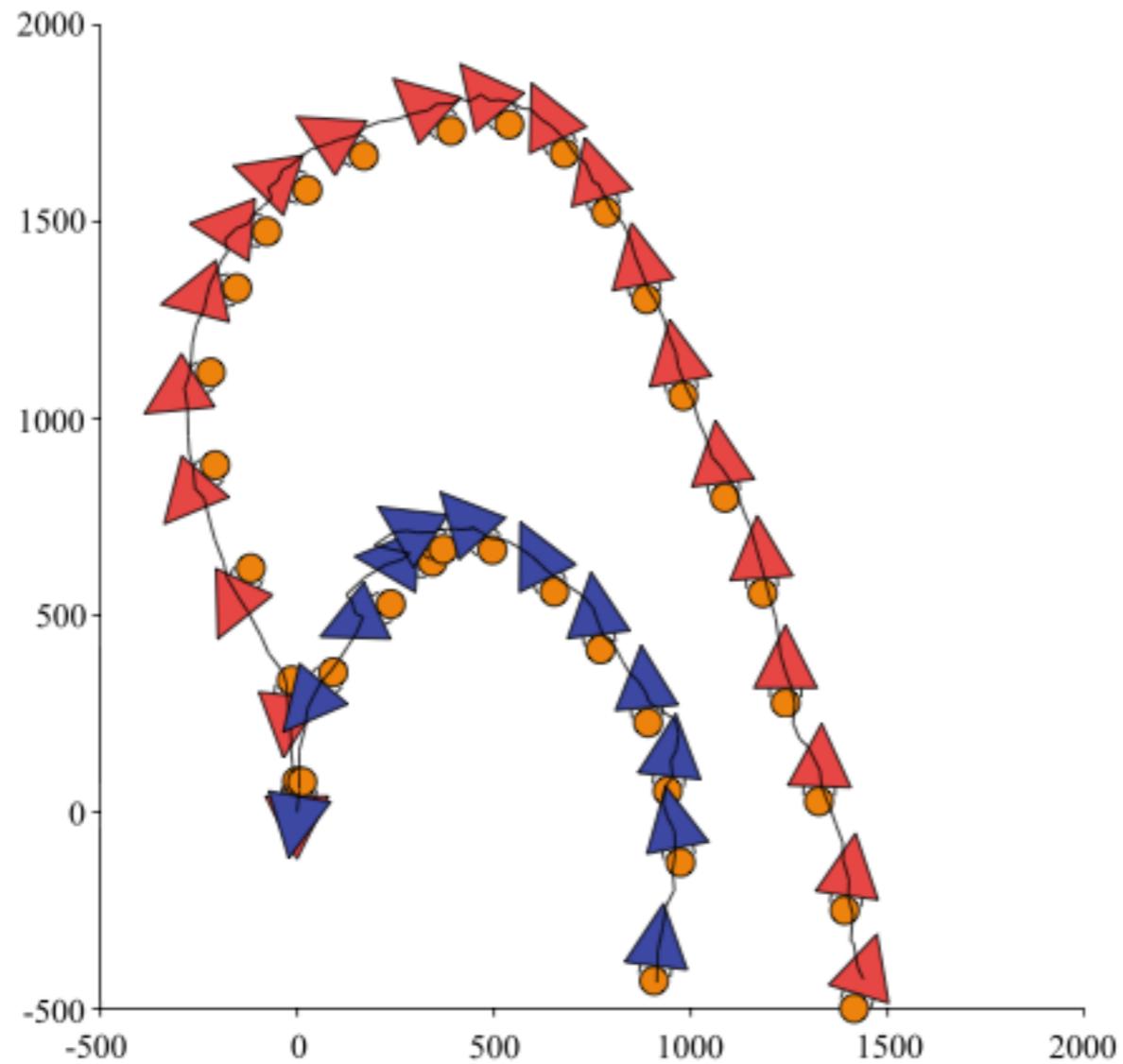
Case Study II: Learning Motor Speeds



$$c(s, a, s') = c(s) = \begin{cases} 0 & \text{if } |\dot{\omega}_d - \dot{\omega}| < \delta, \\ 0.01 & \text{else.} \end{cases}$$



Case Study III: Learning to Dribble



$$Q^{target}(s, a) := \begin{cases} 1.0, & \text{if } s' \in S^-, \\ 0.01, & \text{if } s' \in S^+, \\ 0.01 + \min_b \tilde{Q}(s', b), & \text{else} \end{cases}$$



Value Function Methods

- ... have been the driving reinforcement learning approach in the 1990s.
- You can do loads of cool things with them: Learn Chess at professional level, learn **Backgammon and Checkers at Grandmaster-Level** ... and winning the **Robot Soccer Cup** with a minimum of man power.

So, why are they not always the method of choice?

- You need to fill-up your state-action space up with sufficient samples.
 - Another curse of dimensionality with an exponential explosion.
- 3 → Errors in the Value function approximation might have a catastrophic effect on the policy, **can be very hard to control**