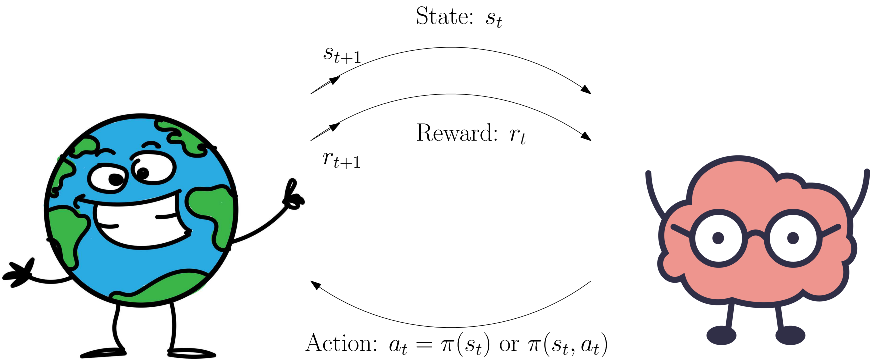
(Deep) Reinforcement Learning

Tómas P. Rúnarsson

The Future of AI

Reinforcement learning is the future of AI.
Reinforcement learning is the best representative of the idea that an intelligent system must be able to learn on its own, without constant supervision (Richard Sutton).

The reinforcement learning problem



learning what to do – mapping situations to actions – so as to maximize a numerical reward signal. The learner is not instructed which actions to take, but must discover which actions yield the most reward by trying them.

The reward hypothesis

*That all of **what we mean by goals and purposes** can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward).*

Markov Decision Processes

A finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. Given any state s and action a, the probability of each possible next state, s', is

$$\mathcal{P}_{ss'}^{a} = \Pr\left[s_{t+1} = s' \middle| s_t = s, a_t = a\right]$$

also known as *transition probabilities*. Similarly, the expected value of the next reward is,

$$\mathcal{R}_{ss'}^{a} = \mathrm{E}\left[r_{t+1} \middle| s_t = s, a_t = a, s_{t+1} = s'\right]$$

The quantities $\mathcal{P}_{ss'}^{a}$ and $\mathcal{R}_{ss'}^{a}$ completely specify the most important aspects of the dynamics of a finite MDP.

State-Value Function for Policy π

Recall that a policy, π, is a mapping from each state, $s \in \mathfrak{S}$, and action $a \in \mathfrak{A}(s)$, to the probability of $\pi(s,a)$ of taking action a when in state s. The *value* of a state s under a policy π is the expected return when starting in state s and following π thereafter, or formally for MDPs as,

$$V^{\pi}(s) = \mathbf{E}_{\pi}\left[R_t \middle| s_t = s\right] = \mathbf{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right]$$

where $\mathbf{E}_{\pi}$ denotes the expected value given that the agent follows policy π. Note that the value of a terminal state is zero, since we set in this case $0 = r_{T+1} = r_{T+2} = \ldots$ and in this case we can also set $\gamma = 1$.

Action-Value Function for Policy π

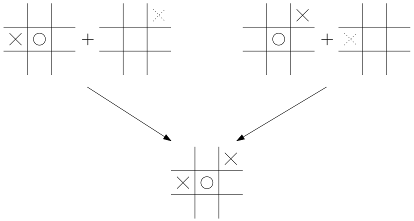
Deciding which action to take using the value function V^π requires a one-step-ahead search to yield the long-term optimal action. This is not always possible (for example a robot cannot take the action to jump off a cliff and see what the value of the resulting state is) and so action-values must be used.

It is also possible to define the value of taking action a in state s under a policy π, as the expected return starting from s, taking action a, and thereafter following policy π, formally

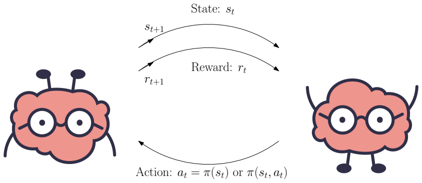
$$\begin{aligned} Q^\pi(s,a) &= \mathbf{E}_\pi\left[R_t \middle| s_t = s, a_t = a\right] \\ &= \mathbf{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a\right] \end{aligned}$$

Afterstate-Value Function for Policy π

Afterstates are states immediately after an agent takes some action. Values for these afterstates are called *afterstate values*. Afterstates are useful when we have knowledge of an initial part of the environment's dynamics but not the full dynamics. For example, in chess we know what the resulting position will be for each possible move, but not how the opponent will reply. Learning afterstate values for games should reduce learning time because there may be many state-actions (position-move) pairs that produce the same resulting state (position). An Afterstate-value function is therefore neither a state-value function nor an action-value function.

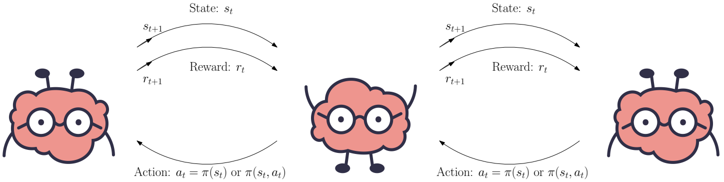


Self Play



- Self-play will result in play optimally against itself. Random exploratory moves will cause all positions to be encountered.
- There is a notion that a self-play player will play in a universal 'optimal' way.
- Playing "optimally" has really only meaning with respect to a particular opponent.
- A self-play player may learn to play better against many opponents.
- A self-play player will play a position which force a win (unless a exploratory move).

Self Play


State: s_t
s_{t+1}
Reward: r_t
r_{t+1}
Action: $a_t = \pi(s_t)$ or $\pi(s_t, a_t)$
State: s_t
s_{t+1}
Reward: r_t
r_{t+1}
Action: $a_t = \pi(s_t)$ or $\pi(s_t, a_t)$

- A self-play player will prefer moves from which it is unlikely to loose even when it occasionally makes random exploratory moves.
- A reinforcement player said to play against itself is not strictly doing so. Rather it plays against a copy of itself adjusted to learn after first-player moves learning separately about each individual position.

Policy Improvement

The *policy improvement theorem*, states: let π and π' be any pair of deterministic policies such that, for all $s \in \mathfrak{S}$,

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

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states, $s \in \mathfrak{S}$:

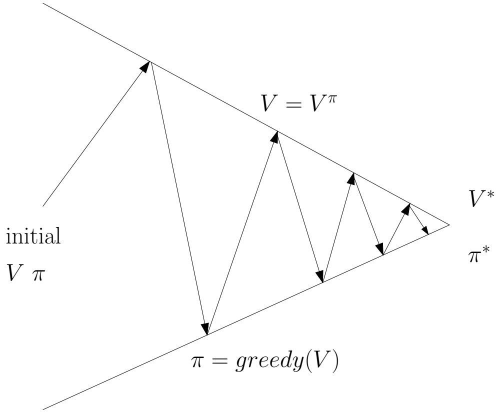
$$V^{\pi'}(s) \geq V^{\pi}(s).$$

The process of making a new policy that improves on an original policy, by making it greedy or nearly greedy with respect to the value function of the original policy, is called *policy improvement*. This is a direct result of the Bellman optimality criteria, that is

$$\begin{aligned} V^{\pi}(s) &= \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \Big(\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s') \Big) \\ V^{*} &= \max_{a} \sum_{s'} \mathcal{P}^{a}_{ss'} \Big(\mathcal{R}^{a}_{ss'} + \gamma V^{*}(s') \Big) \end{aligned}$$

Generalized Policy Iteration

The term *generalized policy iteration* (GPI) is used to refer to the general idea of interacting policy evaluation and policy improvement processes. Almost all reinforcement learning methods can be described as GPI.



Temporal Difference Learning

Recall the iterative policy evaluation

$$\begin{aligned} V^\pi(s) &= \mathbf{E}_\pi\Big[r_{t+1} + \gamma V^\pi(s_{t+1}) \Big| s_t = s\Big] \\ V^\pi(s) &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}\Big(\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\Big) \end{aligned}$$

In dynamic programming a complete model of the environment is given, that is $\mathcal{P}_{ss'}$ and $\mathcal{R}^a_{ss'}$ are known. In *temporal difference* (TD) learning this is unknown. For this reason it is only possible to use the first equation. Using the incremental implementation and the current estimate of V (instead of the true V^π, in the sense that the environment is known), the update becomes

$$\begin{aligned} V(s_t) &\leftarrow \mathbf{E}\Big[r_{t+1} + \gamma V(s_{t+1})\Big] \\ &\leftarrow V(s_t) + \alpha\Big(\big[r_{t+1} + \gamma V(s_{t+1})\big] - V(s_t)\Big) \end{aligned}$$

Generalized n step TD Prediction

The TD learning rule may be written as

$$V(s_t) \leftarrow V(s_t) + \alpha\Big(R_t - V(s_t)\Big)$$

where we have approximated R_t as $r_{t+1} + \gamma V(s_{t+1})$. It is also possible to look one step further and set

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})$$

indeed we can generalize this to looking (n) steps into the future,

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

in this way one would observe the rewards obtained during the next n time steps before updating the value function.

TD(λ)

The *n*-step prediction is

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

and setting $n = 1$ is wowever, a more general version would be to use a weighted average of all *n*-step predicted returns, that is:

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

or in the case of a terminal state at $t = T$,

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t$$

the update is then simply as follows:

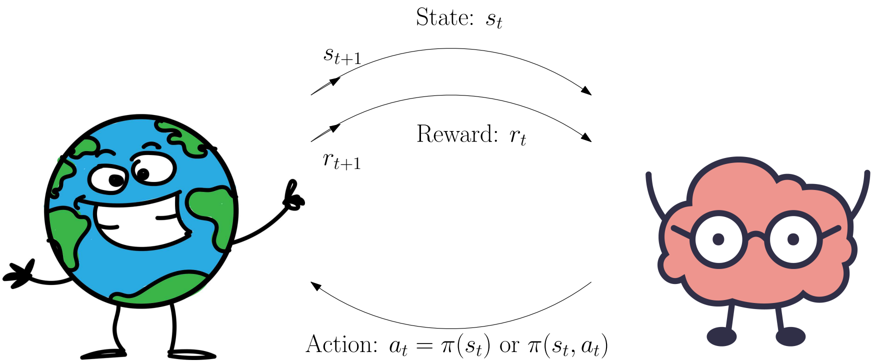
$$V(s_t) \leftarrow V(s_t) + \alpha\Big(R_t^{\lambda} - V(s_t)\Big)$$

TD(λ) – implementation using eligibility traces

The following is the on-line version of tabular TD(λ):

```
1   Initialize $V(s)$ arbitrarily
2   **for** each episode **do**
3       Zero traces $e(s) = 0\ \forall s \in \mathfrak{S}$
4       Initialize $s$
5       **for** each step in episode **do**
6           $a \leftarrow$ action given by $\pi$ for $s$
7           Take action $a$, observe reward, $r$, and next state, $s'$
8           $\delta \leftarrow r + \gamma V(s') - V(s)$
9           $e(s) \leftarrow e(s) + 1$
10          **for** all $s$ **do**
11              $V(s) \leftarrow V(s) + \alpha\delta e(s)$
12              $e(s) \leftarrow \gamma\lambda e(s)$
13          **od**
14          $s \leftarrow s'$
15      **od** until $s$ is a terminal state
16  **od**
```

Deep reinforcement learning



- the state space is too large
- the action space may be continuous or too large
- we would like to build a model of the world ?!

The perceptron or linear function approximator (predictor)

$$V_t = f(net) = f\left(\sum_{i=0}^{n} \theta_t(i) x_t(i)\right)$$

using the activation function[1] $f(\cdot)$.

Given some target value function $V^\pi(s_t)$, the gradient decent method can be used to update the parameter vector $\theta_t = (\theta_t(0), \theta_t(1), \ldots, \theta_t(n))$ by minimizing the MSE as follows:

$$\begin{aligned} \theta_{t+1} &= \theta_t - \tfrac{1}{2}\alpha \frac{\partial}{\partial \theta_t}\left[V^\pi(s_t) - V_t(s_t)\right]^2 \\ &= \theta_t + \alpha\left[V^\pi(s_t) - V_t(s_t)\right]\frac{\partial}{\partial \theta_t} V(s_t) \end{aligned}$$

[1]For example, the hyperbolic tangent sigmoid transfer function: $f(net) = \frac{2}{1+\exp(-2\ net)} - 1$ and $f'(net) = 1 - f(net)^2$

Approximating a predicted moving target

Since $V^{\pi}(s_t)$ is unknown one uses the target R_t^{λ} :

$$\theta_{t+1}(i) = \theta_t(i) + \alpha \left[R_t^{\lambda} - V_t(s_t) \right] \frac{\partial V_t(s_t)}{\partial \theta_t(i)}$$

where by the chain rule:

$$\frac{\partial V(s_t)}{\partial \theta_t(i)} = \frac{\partial net}{\partial \theta_t(i)} \times \frac{\partial V(s_t)}{\partial net}$$

$$\frac{\partial V(s_t)}{\partial \theta_t(i)} = x_t(i) \times f'(net)$$

Backward view

This was the forward view of *gradient-decent* $TD(\lambda)$ for the output layer of a neural network, the equivalent backward view is:

$$\theta_{t+1}(i) = \theta_t(i) + \alpha \delta_t e_t(i)$$

where

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

Updating the eligibility traces

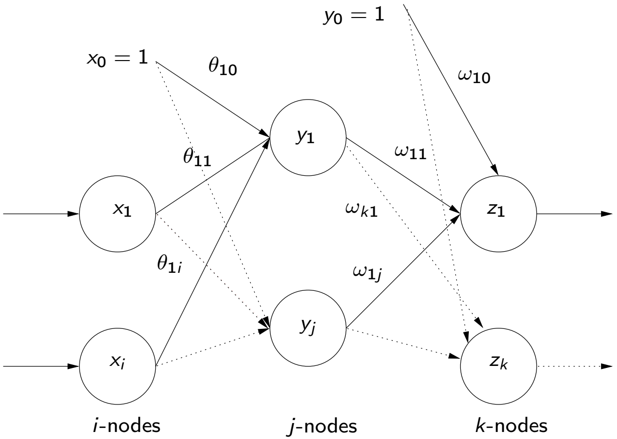
and the eligibility traces are updated as follows:

$$\begin{aligned} e_t(i) &= \gamma\lambda e_{t-1}(i) + \frac{\partial V(s_t)}{\partial \theta_t(i)} \\ &= \gamma\lambda e_{t-1}(i) + x_t(i) f'(net) \end{aligned}$$

For $\lambda < 1$, R_t^λ is not an unbiased estimate of $V^\pi(s_t)$. Also, the method is strictly not a gradient decent method, but it is useful to view it in this way with a bootstrapping approximation in place of the desired output.

Backpropagating the gradient

The following is a generalized description of how eligibility traces are computed for a multi-layered feed-forward neural network.



The update

The update using the target R_t^λ (forward view) is as follows:

$$\omega_{t+1}(k,j) = \omega_t(k,j) + \alpha\big[R_t^\lambda - Q(s_t, a_k)\big]\frac{\partial Q(s_t, a_k)}{\partial \omega_t(k,j)}$$

Careful! we can only consider the TD error for a single output node at a time! All action values, except for "the one" used for updating, should be ignored. Value functions of course have only one output V_{s_t}, i.e. $k = 1$. In both cases the eligibility traces can be computed by considering the gradient only.

Output layer update

Consider first the output layer. Introduce the notation:

$$\nabla_t^{(\omega)}(k) = f'(net_k)$$

where $net_k = \sum_j \omega_t(k,j)y_j$, where y_j are the node activations for layer j, see figure. The eligibility trace then for the weights (ω) between the output node k and the first hidden layer, immediately behind the output layer, are:

$$e_t^{(\omega)}(k,j) \leftarrow \gamma\lambda e_{t-1}^{(\omega)}(k,j) + y_j \nabla_t^{(\omega)}(k)$$

Hidden layer update

and, by backpropagation, for the hidden weights (θ):

$$e_t^{(\theta)}(j,i) \leftarrow \gamma\lambda e_{t-1}^{(\theta)}(j,i) + x_i \nabla_t^{(\theta)}(j)$$

where

$$\nabla_t^{(\theta)}(j) = f'(net_j) \sum_k \nabla_t^{(\omega)}(k)\omega_t(k,j)$$

and as usual $net_j = \sum_i \theta_t(j,i)x_i$. [2]

[2] This is generalized to as many layers (deep) as you like in the usual manner.

Re-introduce the TD error

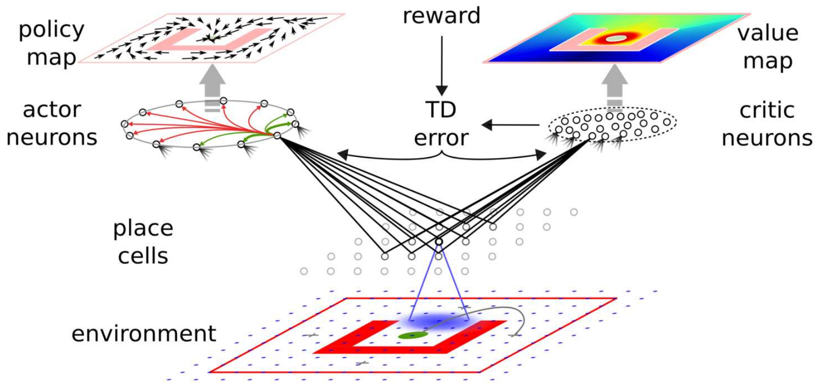
So the formula is exactly the backpropagation formula but without the TD error, i.e. $\delta = r + \gamma V_{s_t+1} - V_{s_t}$, and the step size ($\alpha$). But this is of course re-introduced again in the update formula:

$$\omega_{t+1}(k,j) \leftarrow \omega_t(k,j) + \alpha_\omega \delta_t e_t^{(\omega)}(k,j)$$

and

$$\theta_{t+1}(j,i) \leftarrow \theta_t(j,i) + \alpha_\theta \delta_t e_t^{(\theta)}(j,i).$$

Introducing the Actor-Critic

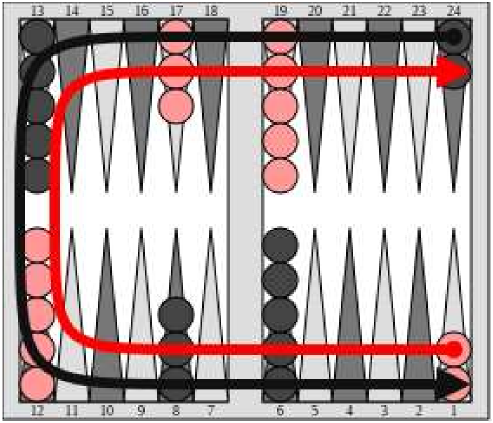


Actor-Critic with eligibility traces

```
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize $S$ (first state of episode)
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ ($d'$-component eligibility trace vector)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ ($d$-component eligibility trace vector)
    $I \leftarrow 1$
    Loop while $S$ is not terminal (for each time step):
        $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
        Take action $A$, observe $S', R$
        $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$        (if $S'$ is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma\lambda^{\mathbf{w}}\mathbf{z}^{\mathbf{w}} + \nabla\hat{v}(S,\mathbf{w})$
        $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma\lambda^{\boldsymbol{\theta}}\mathbf{z}^{\boldsymbol{\theta}} + I\nabla\ln\pi(A|S,\boldsymbol{\theta})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\mathbf{z}^{\mathbf{w}}$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathbf{z}^{\boldsymbol{\theta}}$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

TD(λ) and Backgammon

TD-Gammon is a computer backgammon program developed in 1992 by Gerald Tesauro at IBM's Thomas J. Watson Research Center.

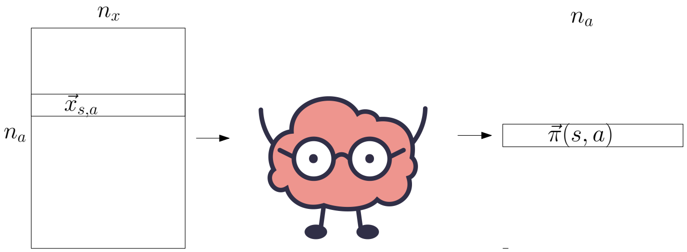


TD(λ) and Backgammon

TD-Gammon's exclusive training through self-play (rather than tutelage) enabled it to explore strategies that humans previously hadn't considered or had ruled out erroneously. Its success with unorthodox strategies had a significant impact on the backgammon community.

Alpha-zero has now done precisely the same for chess in 2017.

Actor-Critic for Backgammon?



Actor-Critic using PyTorch

```
dev=torch.device("cuda" if torch.cuda.is_available() else "cpu")
w1=Variable(torch.randn(nh,nx,device=dev), requires_grad = True)
b1=Variable(torch.zeros(nh,device=dev), requires_grad = True)
theta=Variable(torch.randn(1,nh,device=dev), requires_grad = True)
```

n_x

n_a

```
for i in range(na):
  xa[i,:]=onehot(afterstate[i])

x = torch.tensor(xa)
h = torch.mm(w1,x) + b1
htanh = h.tanh()
```

n_a

```
pi = torch.mm(theta,htanh)
pi = pi.softmax(1)
a = torch.multinomial(pi, 1)

logpi = torch.log(pi[0,a])
logpi.backward()
```

Performance of PubEval, posted by Tesauro 1993

Learner	Training	Best average performance
Neurogammon	3202 moves × 20 cycles	59% vs. *Gammontool*
TD-Gammon, 1992	200,000 games	66.2% vs. *Gammontool*
pubeval	unknown	57% vs. *Gammontool*
HC-Gammon	200,000–800,000 games	40% vs. *pubeval*
ACT-R-Gammon	1,000 games	45.23% vs. *pubeval*
TD-Gammon, 2002	6,000,000+ games	not provided
GMARLB-Gammon	400,000 games	51.2% vs. *pubeval*
GP-Gammon	500,000–2,000,000 games	56.8% vs. *pubeval*
Fuzzeval	100 games	42% vs. *pubeval*
Fuzzeval-TDNN	400,000 games	59% vs. *pubeval*
FAA-SVRRL	5,000 games	51.2% vs. *pubeval*
WPM-9	1,000,000+ games	not provided
WPM-3	200,000 games	51% vs. *WPM-9*

Does Actor-Critic produce a better player for Backgammon?

- **No**: Actor-Critic fails to learn, the δ (TD) signal is too noisy.
- **Yes**, *Advantage* Actor-Critic is able to learn and results in a better player than the greedy TD(λ)

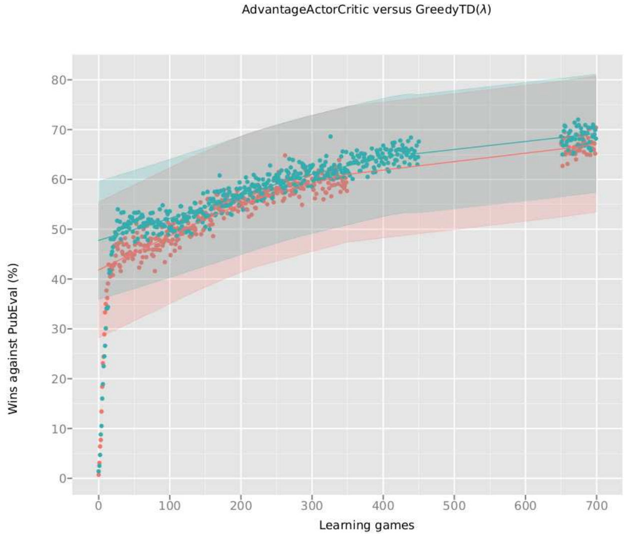
The advangage Actor Critic uses the signal:

$$A(s,a) = Q(s,a) - \sum_{a \in \mathfrak{A}(s)} \pi(s,a) Q(s,a) = Q(s,a) - V(s)$$

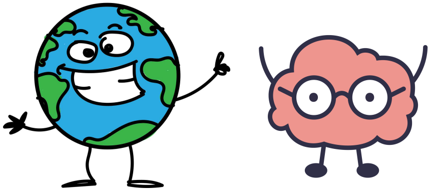
and

$$\theta \leftarrow \theta + \alpha^{\theta} A(s,a) \mathbf{z}^{\theta}$$

Performance of Actor-Critic and Greedy TD(λ)

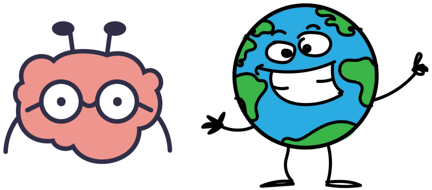


The future of Reinforcement Learning



- The winning feature of reinforcement learning is that you can learn during normal operation. Conventional deep learning learns from trained label training.
- Learning slow so that you can learn fast, learning from one shot. The phrase *you have to learn learning slow so that you can learn fast* by Sutton.

The future of Reinforcement Learning



- Games, technological systems, and systems in general that can be modelled can produce an infinite amount of data, for example via self-play. The real world operates at a different speed. However, we create data to feed such learning systems!
- The key will be to learn from ordinary unsupervised data.
- The focus on "prediction learning". Predicting what will happen, then learning based on what actually does happen.
- We have systems that plan with a learned model of the world. Similar to Dyna, where we integrate planning and learning.