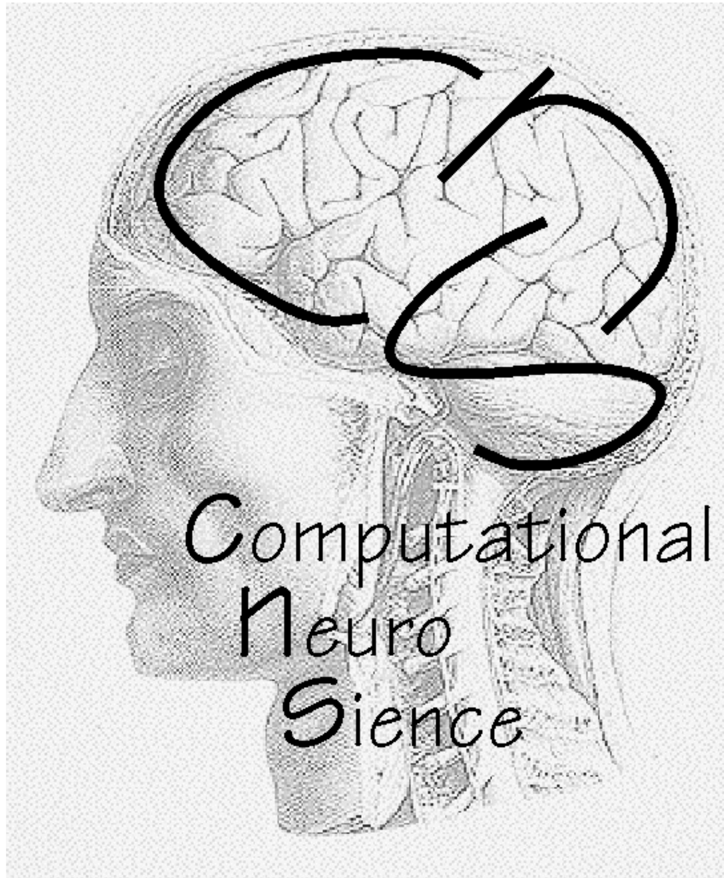
Advanced Computational Neuroscience



Lecture 7:

Reinforcement Learning

Part A

Reinforcement Learning (RL)

Learning from rewards (and punishments)

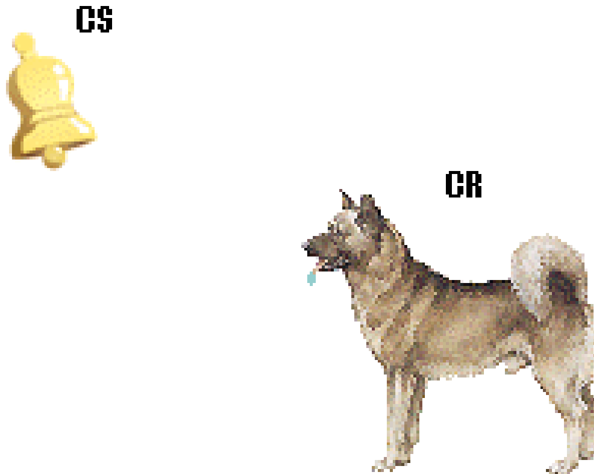
Learning to assess the value of states.

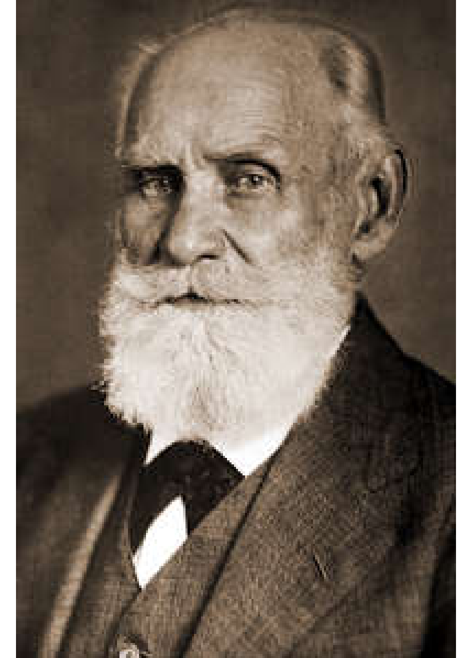
Learning goal directed behavior.

RL has been developed rather independently from two different fields:

1) Dynamic Programming and Machine Learning (Bellman Equation).
2) Psychology (Classical Conditioning) and later Neuroscience (Dopamine System in the brain)

Back to Classical Conditioning





I. Pawlow

U(C)S = Unconditioned Stimulus

U(C)R = Unconditioned Response

CS = Conditioned Stimulus

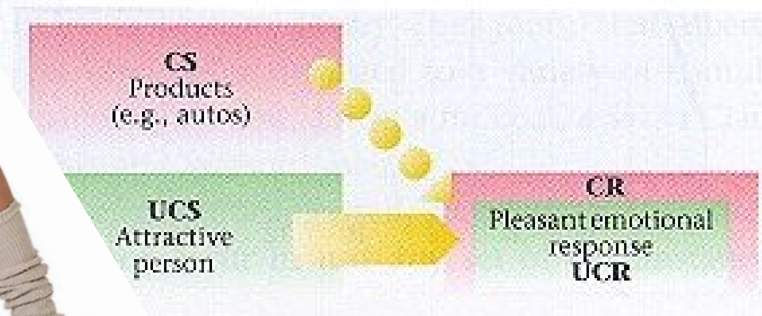
CR = Conditioned Response

Less "classical" but also Conditioning !
(Example from a car advertisement)

Learning the association

CS → U(C)R

Porsche → Good Feeling



Why would we want to go back to CC at all??

So far: We had treated Temporal Sequence Learning in time-continuous systems (ISO, ICO, etc.)
Now: We will treat this in time-discrete systems.

ISO/ICO so far did NOT allow us to learn:

GOAL DIRECTED BEHAVIOR

ISO/ICO performed:

DISTURBANCE COMPENSATION (Homeostasis Learning)

The new RL= formalism to be introduced now will indeed allow us to reach a goal:

LEARNING BY EXPERIENCE TO REACH A GOAL

Overview over different methods – Reinforcement Learning

Machine Learning

Classical Conditioning

Anticipatory Control of Actions and Prediction of Values

Synaptic Plasticity

Correlation of Signals

REINFORCEMENT LEARNING
example based

UN-SUPERVISED LEARNING
correlation based

Dynamic Prog.
(Bellman Eq.)

δ-Rule supervised L.

You are here !

Rescorla/
Wagner

LTP
(LTD=anti)

Eligibility Traces

TD(λ)
often λ=0

TD(1)

TD(0)

Differential
Hebb Rule
("slow")

Differential
Hebb-Rule
("fast")

Monte Carlo
Control

Neur.TD-formalism

STDP-Models
biophysical & network

STDP

Neur.TD-Models
("Critic")

ISO-Learning

Actor/Critic
technical & Basal Gangl.

ISO-Model
of STDP

SARSA

Biophys. of Syn. Plasticity
Dopamine Glutamate

Q-Learning

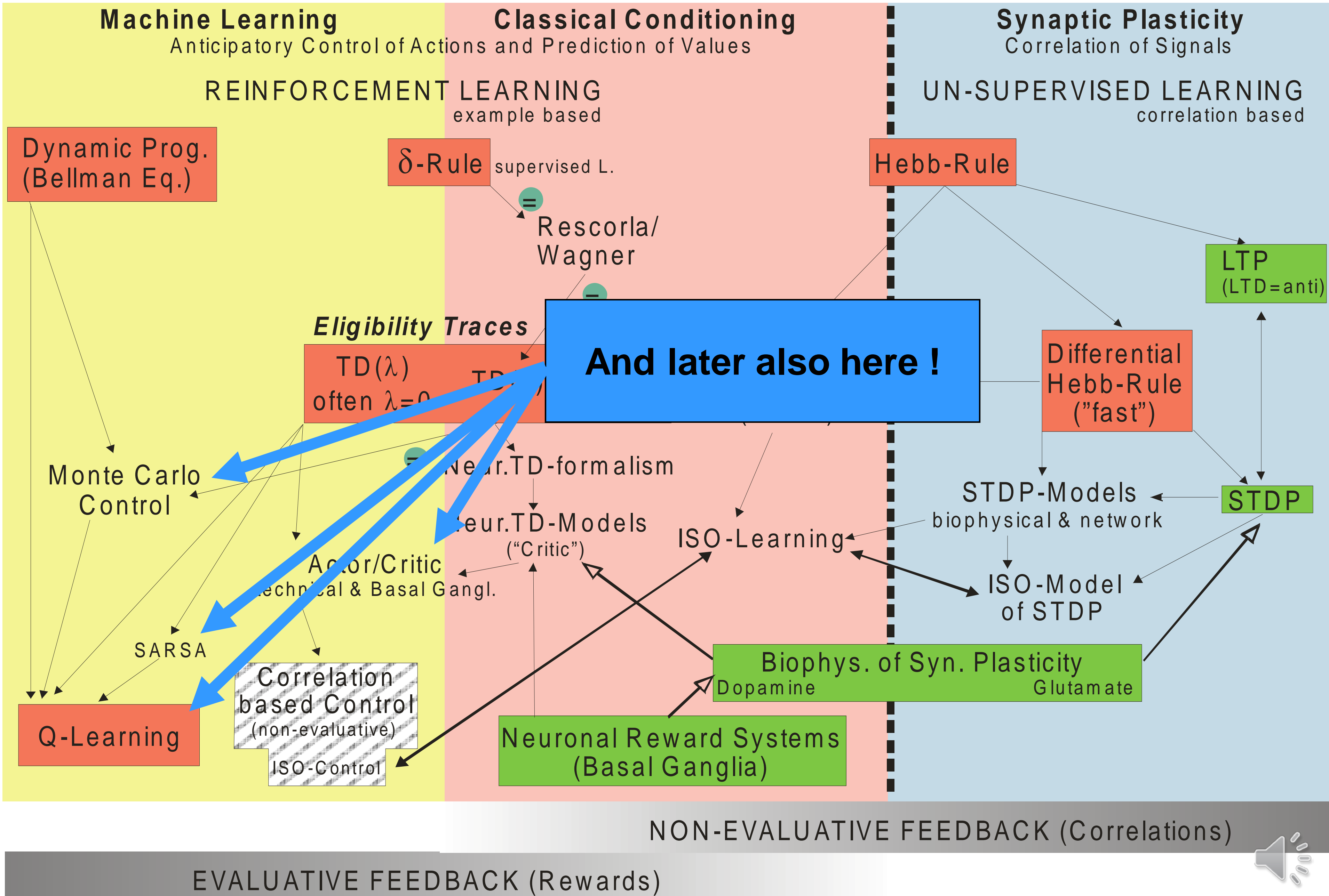
Correlation
based Control
(non-evaluative)
ISO-Control

Neuronal Reward Systems
(Basal Ganglia)

NON-EVALUATIVE FEEDBACK (Correlations)

EVALUATIVE FEEDBACK (Rewards)

Overview over different methods – Reinforcement Learning



Notation

US = r,R = "Reward" (similar to X_0 in ISO/ICO)

CS = s,u = Stimulus = "State[1]" (similar to X_1 in ISO/ICO)

CR = v,V = (Strength of the) Expected Reward = "Value"

UR = --- (not required in mathematical formalisms of RL)

Weight = ω = weight used for calculating the value; e.g. $v=\omega u$

Action = a = "Action"

Policy = π = "Policy"

"…" = Notation from Sutton & Barto 1998, red from S&B as well as from Dayan and Abbott.

[1] Note: The notion of a "state" really only makes sense as soon as there is more than one state.

A note on “Value” and “Reward Expectation”

If you are at a certain state then you would value this state according to how much reward you can expect when moving on from this state to the end-point of your trial.

Hence:

Value = Expected Reward !

More accurately:

Value = Expected cumulative future discounted reward.
(for this, see later!)

Types of Rules

1) Rescorla-Wagner Rule: Allows for explaining several types of conditioning experiments.
2) TD-rule (TD-algorithm) allows measuring the value of states and allows accumulating rewards. Thereby it generalizes the Resc.-Wagner rule.
3) TD-algorithm can be extended to allow measuring the value of actions and thereby control behavior either by ways of
    a) Q or SARSA learning or with
    b) Actor-Critic Architectures

Overview over different methods – Reinforcement Learning

Rescorla-Wagner Rule

	Pre-Train	Train	Result
Pavlovian:		$u \rightarrow r$	$u \rightarrow v = max$
Extinction:	$u \rightarrow r$	$u \rightarrow \bullet$	$u \rightarrow v = 0$
Partial:		$u \rightarrow r$ $u \rightarrow \bullet$	$u \rightarrow v < max$

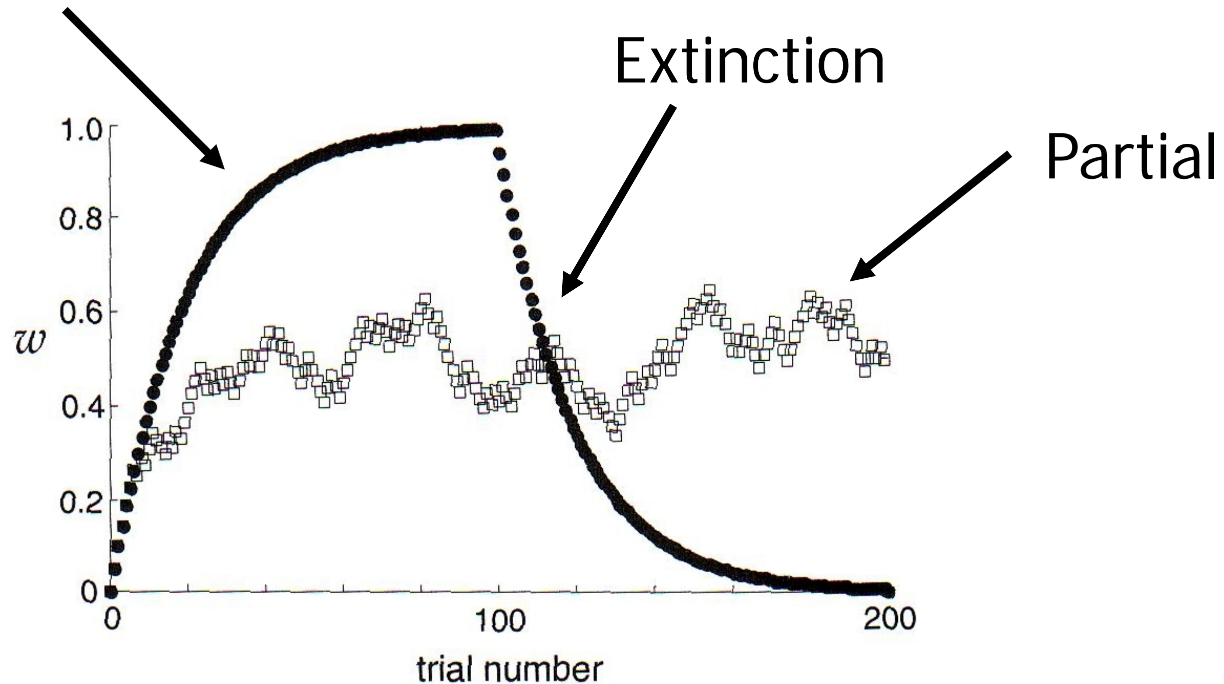
We define: $v = \omega u$, with $u=1$ or $u=0$, binary and
$\omega \rightarrow \omega + \mu\delta u$ with $\delta = r - v$

The associability between stimulus u and reward r is represented by the learning rate μ.

This learning rule minimizes the avg. squared error between actual reward r and the prediction v, hence $\min\langle(r-v)^2\rangle$

We realize that δ is the prediction error.

Pawlovian



Stimulus u is paired with r=1 in 100% of the discrete “epochs” for Pawlovian
and in 50% of the cases for Partial.

Rescorla-Wagner Rule, Vector Form for Multiple Stimuli

We define: $v = \mathbf{w} \cdot \mathbf{u}$, and

$\mathbf{w} \rightarrow \mathbf{w} + \mu\delta\mathbf{u}$ with $\delta = r - v$

Where we use stochastic gradient descent for minimizing δ

Do you see the similarity of this rule with the δ-rule discussed earlier !?

	Pre-Train	Train	Result
Blocking:	$u_1 \rightarrow r$	$u_1 + u_2 \rightarrow r$	$u_1 \rightarrow v = \max$, $u_2 \rightarrow v = 0$

For Blocking: The association formed during pre-training leads to $\delta = 0$. As ω_2 starts with zero the expected reward $v = \omega_1 u_1 + \omega_2 u_2$ remains at r. This keeps $\delta = 0$ and the new association with u_2 cannot be learned.

Rescorla-Wagner Rule, Vector Form for Multiple Stimuli

	Pre-Train	Train	Result
Inhibitory:		$u_1+u_2 \rightarrow \bullet$, $u_1 \rightarrow r$	$u_1 \rightarrow v=\max$, $u_2 \rightarrow v<0$

Inhibitory Conditioning: Presentation of one stimulus together with the reward and alternating presenting a pair of stimuli where the reward is missing. In this case the second stimulus actually predicts the ABSENCE of the reward (negative v).

Trials in which the first stimulus is presented together with the reward lead to $\omega_1>0$.

In trials where both stimuli are present the net prediction will be $v=\omega_1 u_1+\omega_2 u_2 = 0$.

As $u_{1,2}=1$ (or zero) and $\omega_1>0$, we get $\omega_2<0$ and, consequentially, $v(u_2)<0$.

Rescorla-Wagner Rule, Vector Form for Multiple Stimuli

	Pre-Train	Train	Result
Overshadow:		$u_1+u_2 \rightarrow r$	$u_1 \rightarrow v<max$, $u_2 \rightarrow v<max$

Overshadowing: Presenting always two stimuli together with the reward will lead to a "sharing" of the reward prediction between them. We get $v = \omega_1 u_1 + \omega_2 u_2 = r$. Using different learning rates μ will lead to differently strong growth of $\omega_{1,2}$ and represents the often observed different saliency of the two stimuli.

Rescorla-Wagner Rule, Vector Form for Multiple Stimuli

	Pre-Train	Train	Result
Secondary:	$u_1 \rightarrow r$	$u_2 \rightarrow u_1$	$u_2 \rightarrow v = max$

Secondary Conditioning reflect the "replacement" of one stimulus by a new one for the prediction of a reward.

As we have seen the Rescorla-Wagner Rule is very simple but still able to represent many of the basic findings of diverse conditioning experiments.

Secondary conditioning, however, CANNOT be captured.

(sidenote: The ISO/ICO rule can do this!)

Predicting *Future* Reward

The Rescorla-Wagner Rule cannot deal with the sequentiallity of stimuli (required to deal with Secondary Conditioning). As a consequence it treats this case similar to Inhibitory Conditioning lead to negative ω_2.

Animals can predict to some degree such sequences and form the correct associations. For this we need algorithms that keep track of time.

Here we do this by ways of states that are subsequently visited and evaluated.

Sidenote: ISO/ICO treat time in a fully continuous way, typical RL formalisms (which will come now) treat time in discrete steps.

Prediction and Control

The goal of RL is two-fold:

1) To predict the value of states (exploring the state space following a policy) – Prediction Problem.

2) Change the policy towards finding the optimal policy – Control Problem.

Terminology (again):

- State,
- Action,
- Reward,
- Value,
- Policy

Markov Decision Problems (MDPs)

terminal states 15 16

13 14

9 10 11 12

rewards r_1 r_2

actions a_1 a_2 a_{14} a_{15}

states s 1 2 3 4 5 6 7 8

If the future of the system depends always only on the current state and action then the system is said to be "Markovian".

What does an RL-agent do ?

An RL-agent explores the state space trying to accumulate as much reward as possible. It follows a behavioral policy performing actions (which usually will lead the agent from one state to the next).

What does an RL-agent do ?

For the Prediction Problem: It updates the value of each given state by assessing how much future (!) reward can be obtained when moving onwards from this state (State Space). It does not change the policy, rather it *evaluates* it. (Policy Evaluation).

				R
		0.0		
x	x	x	x	x

			0.9	R
			0.8	0.9
	etc			
0.1	0.1	0.1	0.1	0.1

possible start locations

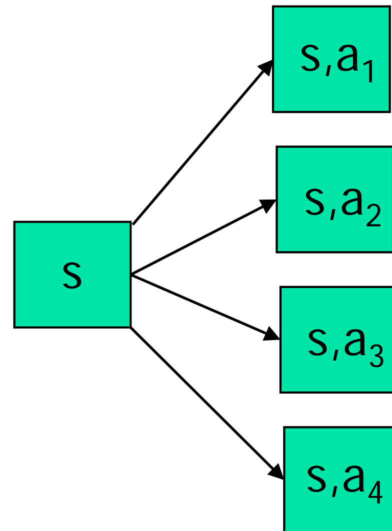
Policy Evaluation give values of states

What does an RL-agent do ?

For the Control Problem: It updates the value of each given action at a given state and of by assessing how much future reward can be obtained when performing this action at that state (State-Action Space, which is larger than the State Space). and all following actions at the following state moving onwards.

Policy:
p(N) = 0.5
p(S) = 0.125
p(W) = 0.25
p(E) = 0.125

			0.9	**R**
			0.8	**0.9**
	etc			
0.1	**0.1**	**0.1**	**0.1**	**0.1**



State-Action spaces are bigger than state spaces!

Guess: Will we have to evaluate ALL states and actions onwards?

What does an RL-agent do ?

Exploration – Exploitation Dilemma: The agent wants to get as much cumulative reward (also often called *return*) as possible. For this it should always perform the most rewarding action "exploiting" its (learned) knowledge of the state space. This way it might however miss an action which leads (a bit further on) to a much more rewarding path. Hence the agent must also "explore" into unknown parts of the state space. The agent must, thus, balance its policy to include exploitation and exploration.

Policies

1) Greedy Policy: The agent always exploits and selects the most rewarding action. This is sub-optimal as the agent never finds better new paths.

Policies

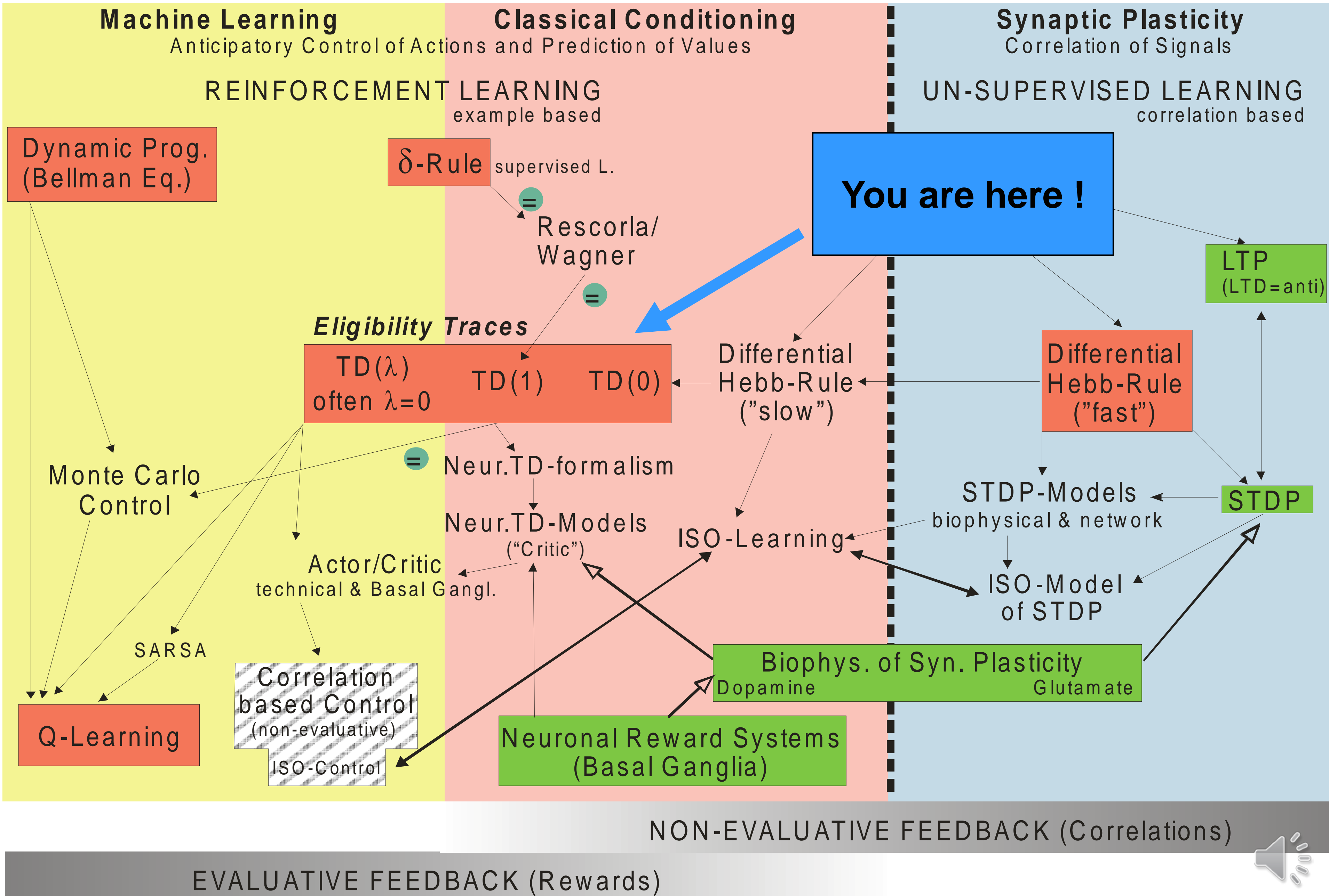
2) ε-Greedy Policy: With a small probability ε the agent will choose a non-optimal action. *All non-optimal actions are chosen with *equal* probability.* This can take very long as it is not known how big ε should be. One can also "anneal" the system by gradually lowering ε to become more and more greedy.

3) Softmax Policy: ε-greedy can be problematic because of (*). Softmax ranks the actions according to their values and chooses roughly following the ranking using for example:

$$\frac{\exp(\frac{Q_a}{T})}{\sum_{b=1}^{n} \exp(\frac{Q_b}{T})}$$

where Q_a is value of the currently to be evaluated action a and T is a temperature parameter. For large T all actions have approx. equal probability to get selected.

Overview over different methods – Reinforcement Learning



Towards TD-learning – Pictorial View

In the following slides we will treat "Policy evaluation": We define some given policy and want to *evaluate the state space.* We are at the moment still not interested in evaluating actions or in improving policies.

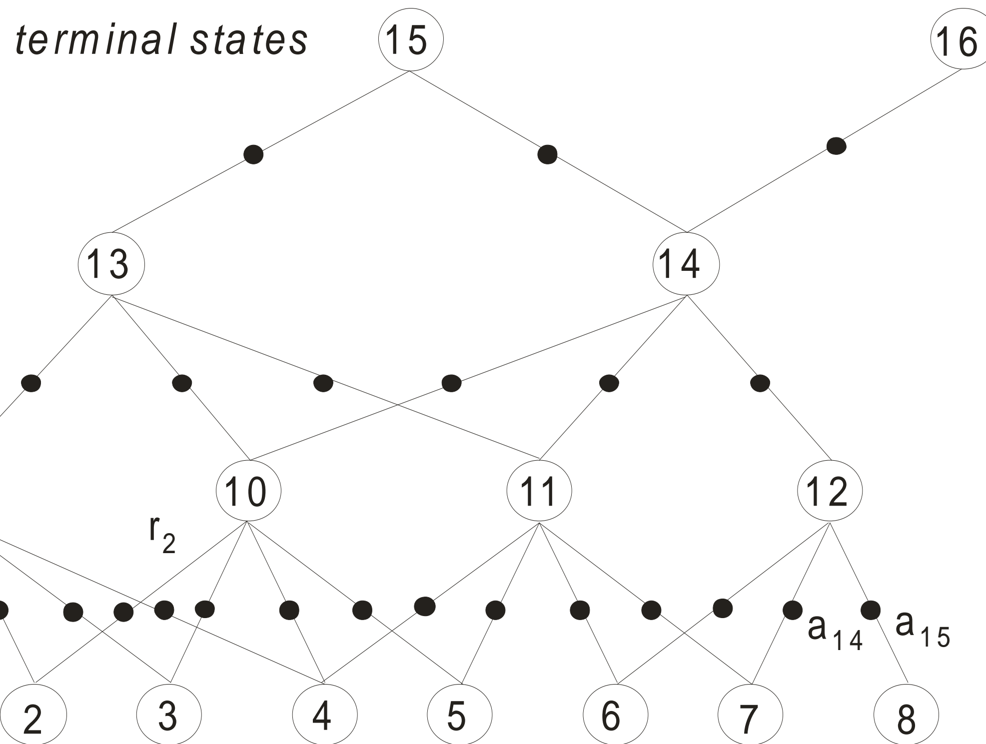
Back to the question: To get the value of a given state, will we have to evaluate ALL states and actions onwards?

There is no unique answer to this! Different methods exist which assign the value of a state by using differently many (weighted) values of subsequent states. We will discuss a few but concentrate on the most commonly used TD-algorithm(s).

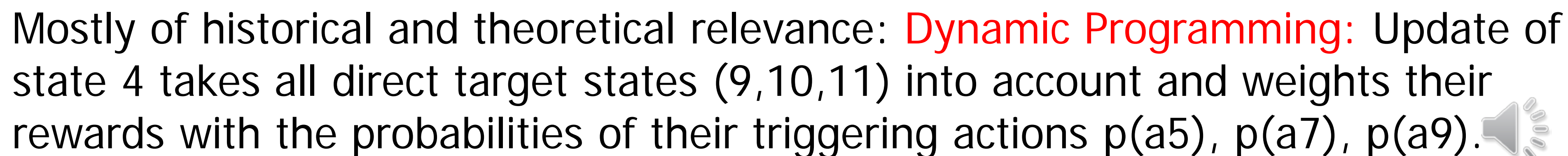
Temporal Difference (TD) Learning

Lets, for example, evaluate just state 4:

Tree backup methods:



Most simplistically and very slow: Exhaustive Search: Update of state 4 takes all direct target states and all secondary, ternary, etc. states into account until reaching the terminal states and weights all of them with their corresponding action probabilities.



Mostly of historical and theoretical relevance: Dynamic Programming: Update of state 4 takes all direct target states (9,10,11) into account and weights their rewards with the probabilities of their triggering actions p(a5), p(a7), p(a9).

Linear backup methods

terminal states 15 16

13 14

C

9 10 11 12

r_1 r_2

a_1 a_2 a_{14} a_{15}

s 1 2 3 4 5 6 7 8

tree.cdr

Full linear backup: Monte Carlo [= TD(1)]: Sequence C (4,10,13,15): Update of state 4 (and 10 and 13) can commence as soon as terminal state 15 is reached.

Linear backup methods

terminal states 15 16

13 14

9 10 11 12

r_1 r_2 A

a_1 a_2 a_{14} a_{15}

s 1 2 3 4 5 6 7 8

tree.cdr

Single step linear backup: TD(0): Sequence A: (4,10)
Update of state 4 can commence as soon as state 10 is reached. This is the most important algorithm.

Linear backup methods

terminal states 15 16

13 14

B C

9 10 11 12

r_1 r_2 A

a_1 a_2 a_{14} a_{15}

s 1 2 3 4 5 6 7 8

tree.cdr

Weighted linear backup: TD(λ): Sequences A, B, C: Update of state 4 uses a weighted average of all linear sequences until terminal state 15.

Why are we calling these methods “backups” ? Because we move to one or more next states, take their rewards&values, and then *move back* to the state which we would like to update and do so!

For the following:

Note: RL has been developed largely in the context of machine learning. Hence all mathematically rigorous formalisms for RL comes from this field.

A rigorous transfer to neuronal model is a more recent development.

Thus, in the following we will use the machine learning formalism to derive the math and in parts relate this to neuronal models later.

This difference is visible from using

STATES s_t for the machine learning formalism and

TIME t when talking about neurons.

Formalising RL: Policy Evaluation with goal to find the optimal value function of the state space

We consider a sequence s_t, r_{t+1}, s_{t+1}, r_{t+2}, . . . , r_T , s_T . Note, rewards occur downstream (in the future) from a visited state. Thus, r_{t+1} is the next *future* reward which can be reached starting from state s_t. The complete return R_t to be expected in the future from state s_t is, thus, given by:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T,$$

where $\gamma \leq 1$ is a discount factor. This accounts for the fact that rewards in the far future should be valued less.

Reinforcement learning assumes that the value of a state V(s) is directly equivalent to the expected return Ep at this state, where p denotes the (here unspecified) action policy to be followed.

$$V(s) = E_\pi\{R_t | s_t = s\}$$

Thus, the value of state s_t can be iteratively updated with:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)]$$

We use α as a step-size parameter, which is not of great importance here, though, and can be held constant.
Note, if $V(s_t)$ correctly predicts the expected complete return R_t, the update will be zero and we have found the final value. This method is called *constant-α Monte Carlo* update. It requires to wait until a sequence has reached its terminal state (see some slides before!) before the update can commence. For long sequences this may be problematic. Thus, one should try to use an incremental procedure instead. We define a different update rule with:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \underbrace{\gamma V(s_{t+1}) - V(s_t)}]$$

This is why it is called TD (temp. diff.) Learning

The elegant trick is to assume that, if the process converges, the value of the next state $V(s_{t+1})$ should be an accurate estimate of the expected return downstream to this state (i.e., downstream to s_{t+1}). Thus, we would hope that the following holds:

$$R_t = r_{t+1} + \gamma V(s_{t+1})$$

Indeed, proofs exist that under certain boundary conditions this procedure, known as *TD(0)*, converges to the optimal value function for all states.

In principle the same procedure can be applied all the way downstream writing:

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

Thus, we could update the value of state s_t by moving downstream to some future state s_{t+n-1} accumulating all rewards along the way including the last future reward r_{t+n} and then approximating the missing bit until the terminal state by the estimated value of state s_{t+n} given as $V(s_{t+n})$. Furthermore, we can even take different such update rules and average their results in the following way:

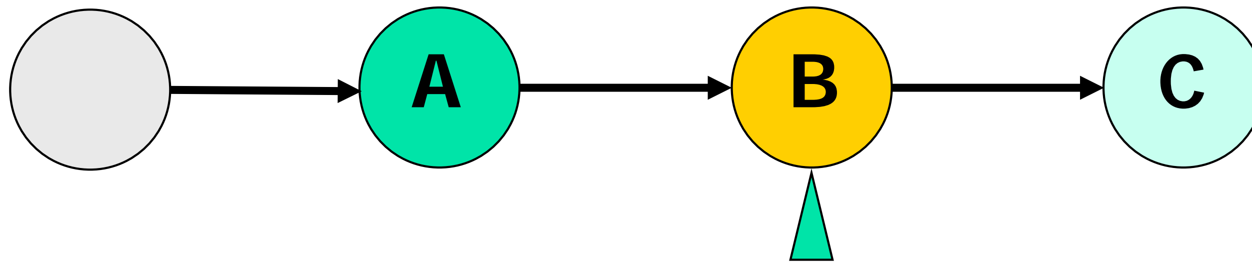
$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$
$$V(s_t) \leftarrow V(s_t) + \alpha [R_t^\lambda - V(s_t)],$$

where $0 \leq \lambda \leq 1$. This is the most general formalism for a TD-rule known as *forward TD(λ)-algorithm*, where we assume an infinitely long sequence.

Linear backup methods

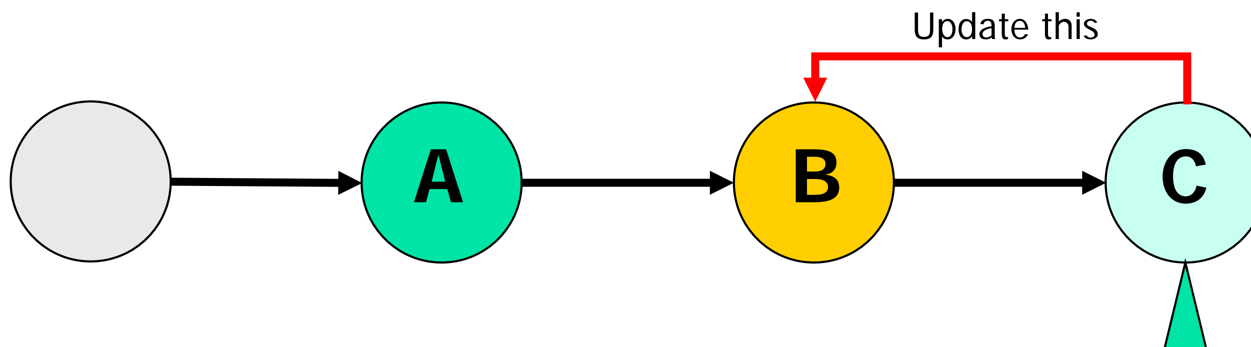
Weighted linear backup: TD(λ): Sequences A, B, C: Update of state 4 uses a weighted average of all linear sequences until terminal state 15.

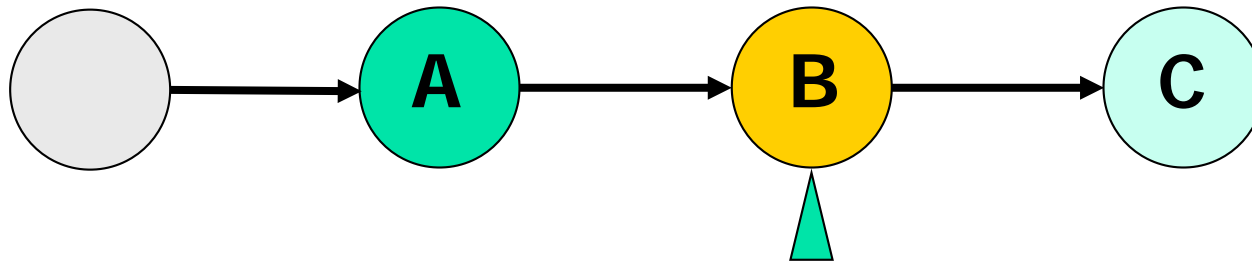
The disadvantage of this formalism is still that, for all $\lambda > 0$, we have to wait until we have reached the terminal state until update of the value of state s_t can commence.
There is a way to overcome this problem by introducing *eligibility traces* (Compare to ISO/ICO before!).



Let us assume that we came from state A and now we are currently visiting state B. B's value can be updated by the TD(0) rule after we have moved on by only a single step to, say, state C. We define the incremental update as before as: $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$

Normally we would only assign a new value to state B by performing $V(s_B) \leftarrow V(s_B) + \alpha\delta_B$, not considering any other previously visited states.





Let us assume that we came from state A and now we are currently visiting state B. B's value can be updated by the TD(0) rule after we have moved on by only a single step to, say, state C. We define the incremental update as before as:

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

Normally we would only assign a new value to state B by performing $V(s_B) \leftarrow V(s_B) + \alpha\delta_B$, not considering any other previously visited states. **In using eligibility traces we do something different and assign new values to *all* previously visited states,** making sure that changes at states long in the past are much smaller than those at states visited just recently. To this end we define the eligibility trace of a state as:

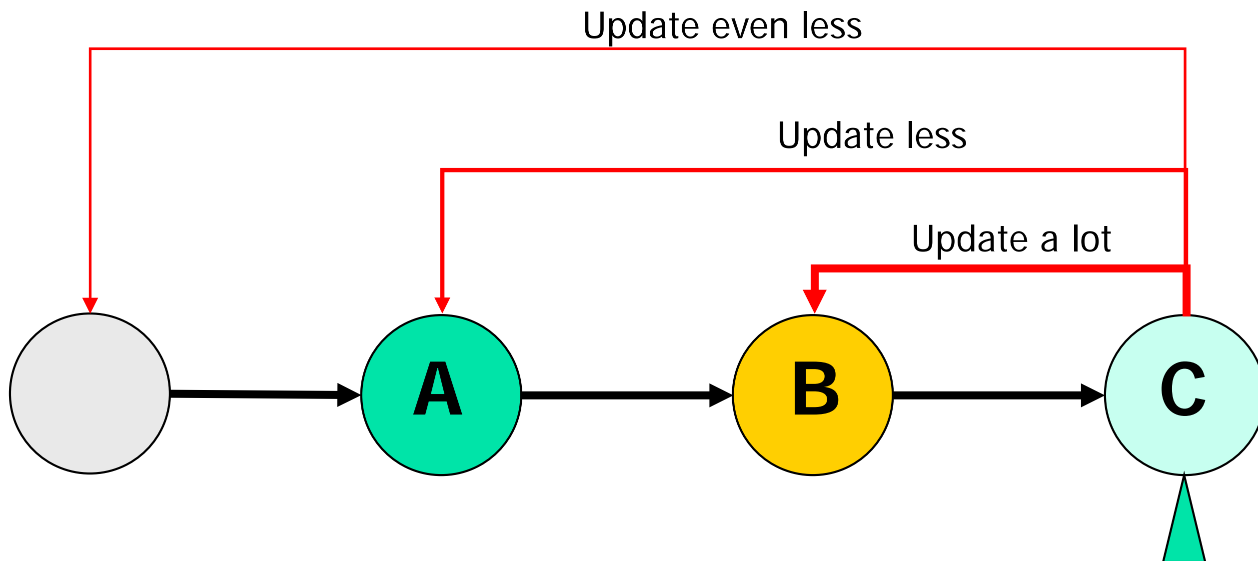
$$\overline{x_t}(s) = \begin{cases} \gamma\lambda x_{t-1}(s) & \text{if} \quad s \neq s_t \\ \gamma\lambda x_{t-1}(s) + 1 & \text{if} \quad s = s_t \end{cases}$$

Thus, the eligibility trace of the currently visited state is incremented by one, while the eligibility traces of all other states decay with a factor of $\gamma\lambda$.

Instead of just updating the most recently left state s_t we will now loop through all states visited in the past of this trial which still have an eligibility trace larger than zero and update them according to:

$$V(s_t) \leftarrow V(s_t) + \alpha\delta_t\overline{x_t}(s)$$

In our example we will, thus, also update the value of state A by $V(s_A) \leftarrow V(s_A) + \alpha\delta_B\,\overline{x_B}(A)$. This means we are using the TD-error δ_B from the state transition B $\rightarrow$ C weight it with the currently existing numerical value of the eligibility trace of state A given by $\overline{x_B}(A)$ and use this to correct the value of state A “a little bit”.

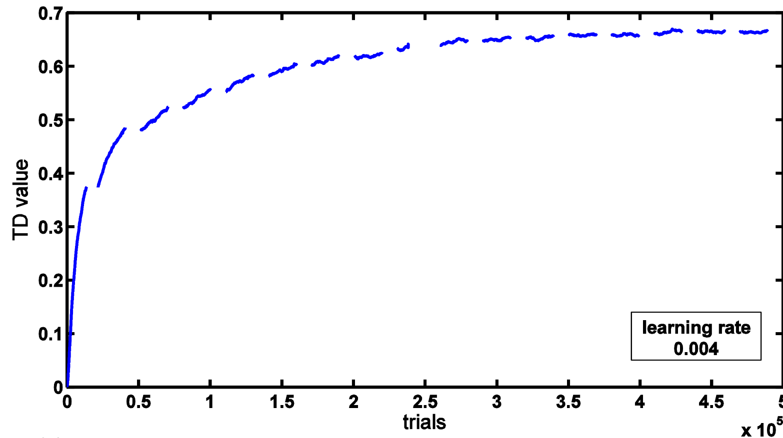


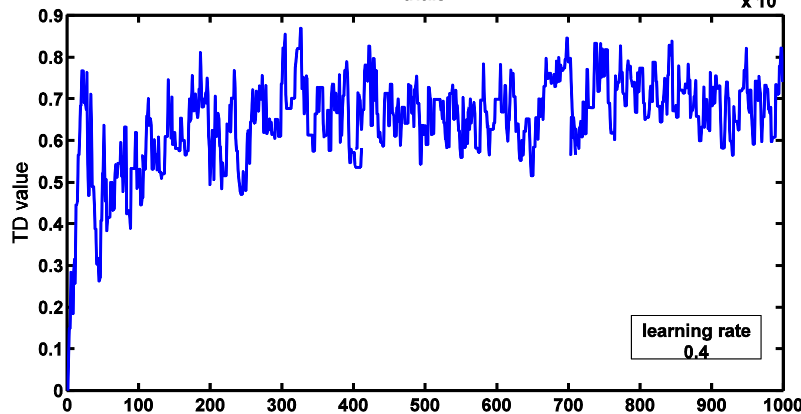
Instead of just updating the most recently left state s_t we will now loop through all states visited in the past of this trial which still have an eligibility trace larger than zero and update them according to:

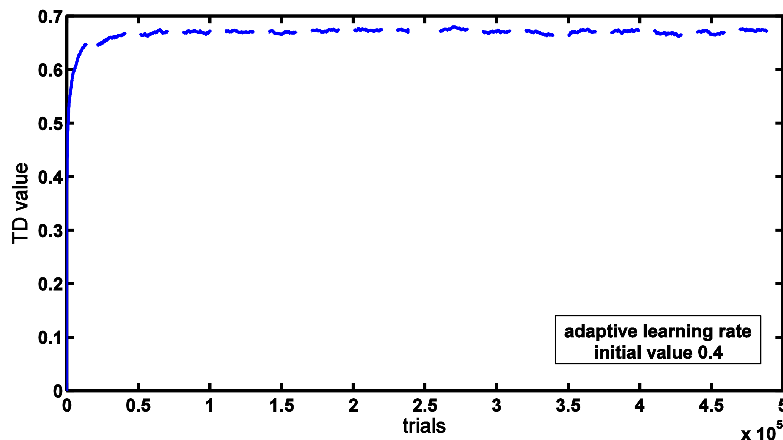
$$V(s_t) \leftarrow V(s_t) + \alpha\delta_t\overline{x_t}(s)$$

In our example we will, thus, also update the value of state A by $V(s_A) \leftarrow V(s_A) + \alpha\delta_B \overline{x_B}(A)$. This means we are using the TD-error δ_B from the state transition B → C weight it with the currently existing numerical value of the eligibility trace of state A given by $\overline{x_B}(A)$ and use this to correct the value of state A "a little bit". **This procedure requires always only a single newly computed TD-error using the computationally very cheap TD(0)-rule, and all updates can be performed on-line when moving through the state space without having to wait for the terminal state.** The whole procedure is known as *backward TD(λ)-algorithm* and it can be shown that it is mathematically equivalent to forward TD(λ) described above.

Rigorous proofs exist the TD-learning will always find the optimal value function (can be slow, though).







TD(0) on a 9x9 grid

Asymptotic convergence of one weight.

Learning Rate 0.004

Learning Rate 0.4

Learning Rate 0.4
and dropping over time.

Next:

Reinforcement Learning, Part B