Reinforcement Learning with Applications of Intelligent Agents

Dr. Michael Thrun

Reinforcement Learning (RL)

- Reinforcement learning can solve problems that are sequential and stochastic, e.g.,
  - Healthcare treatment [Håkansson et al., 2020]
  - Drug sensitvity prediction in cancer by ranking prediction algorithms [Daoud & Freisleben et al., 2020]
  - Taxation [Zheng & Socher et al., 2020 & 2021]

Basic Principles

- Agent learns from interactions with environment
- Feedback is restricted to delayed reward

=> No need for pairs of input and correct output

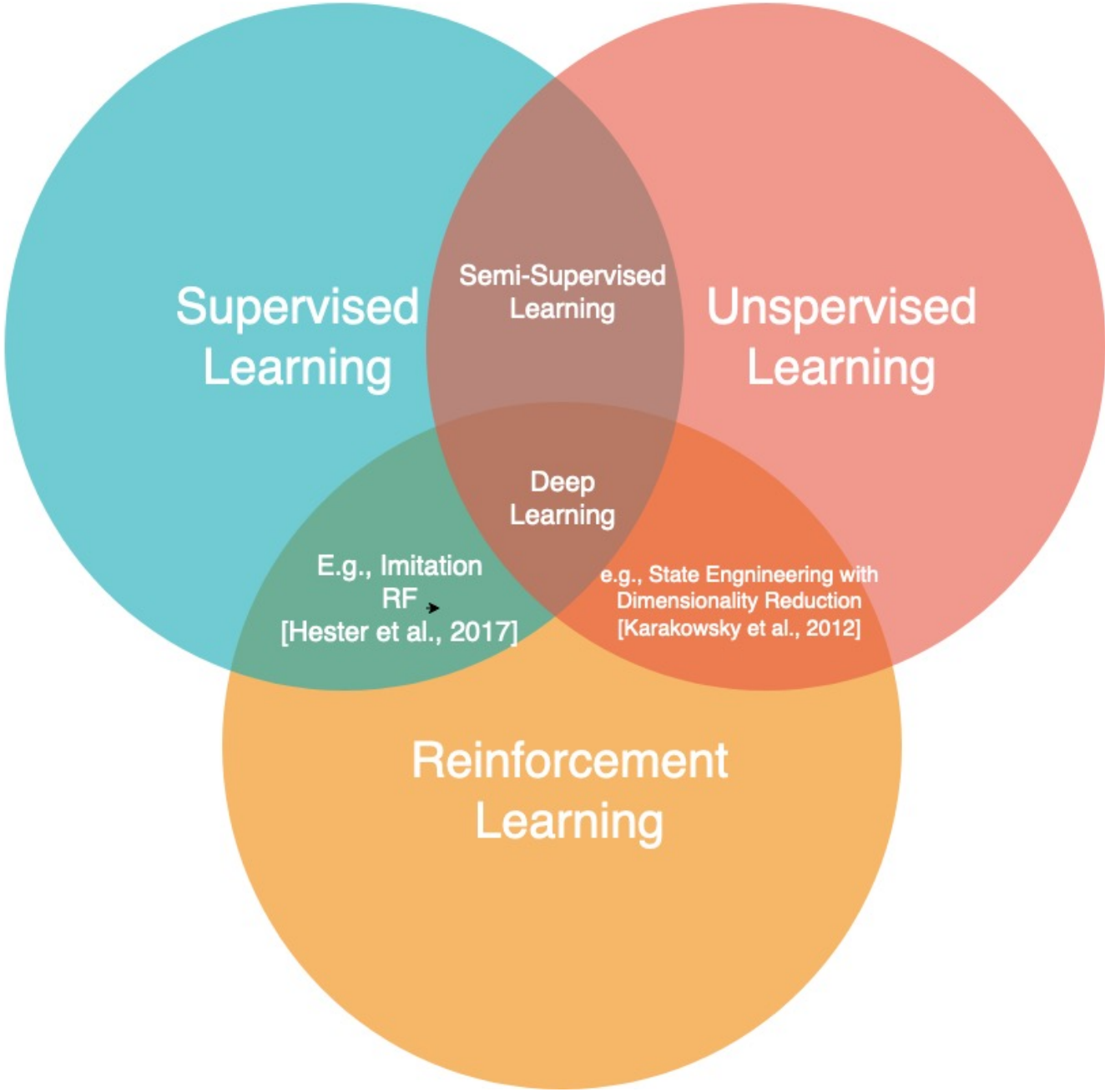
Brief History of Reinforcement Learning (RL)

- RL is based on decades of research in AI and animal learning [Sutton & Barto, 1998], e.g.,
  1. Bellmann's optimal control, dynamic programming (**DP**) and markov decision processes (**MDP**), ~1950
  2. Trial and error learning in animals [Woodworth 1938; Pavlov 1927; Alexander Bain (ca. 1850)]
  3. Temporal difference learning (**TD**) [Minsky 1954; Samuel, 1959]

- Chris Watkins [1989 PhD thesis, Learning from Delayed Rewards] integrated these three reasearch area into a new algorithm called Q-learning

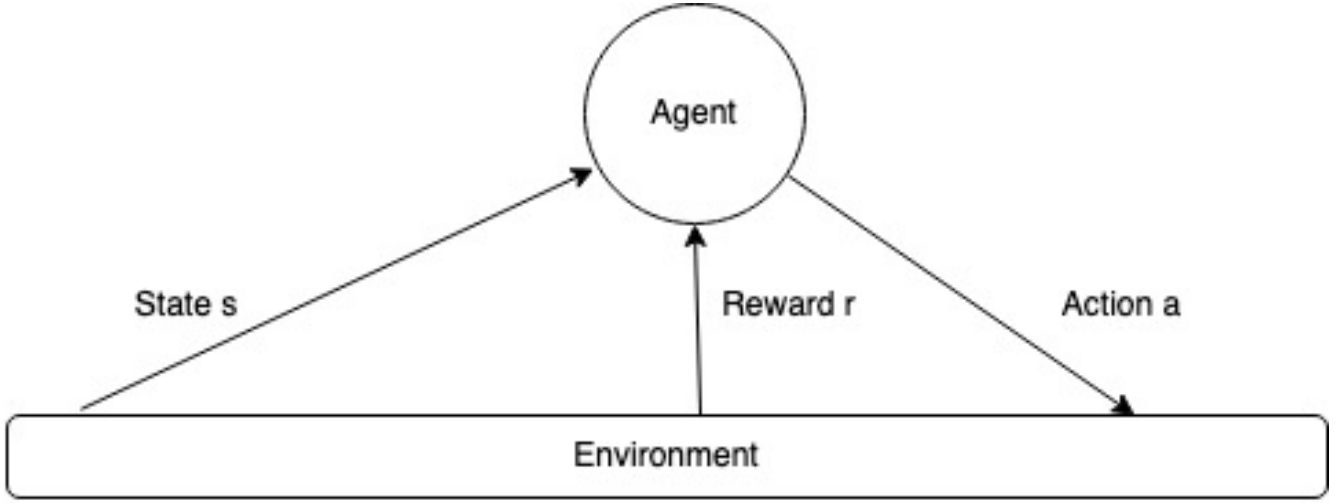
Breakthroughs in RL

- RL concepts from supervised and unsupervised learning
- RL+MLP reached a level similar to the top three human player in the backgammon [Tesauro, 1995]
- RL+Deep learning (DQN) beats humans on atari games [Mnih et al., 2013 & 2015]
- DQN+supervised learning using human knowledge beats grandmaster in GO [Silver et al., 2016]
- DQN+MC beats its 2016 version in GO [Silver et al., 2017]
- DQN+MC beats grandmaster in chess [Silver et al., 2018]
- DQN beats grandmasters in real time strategy game (starcraft 2) [Wang et al., 2021]



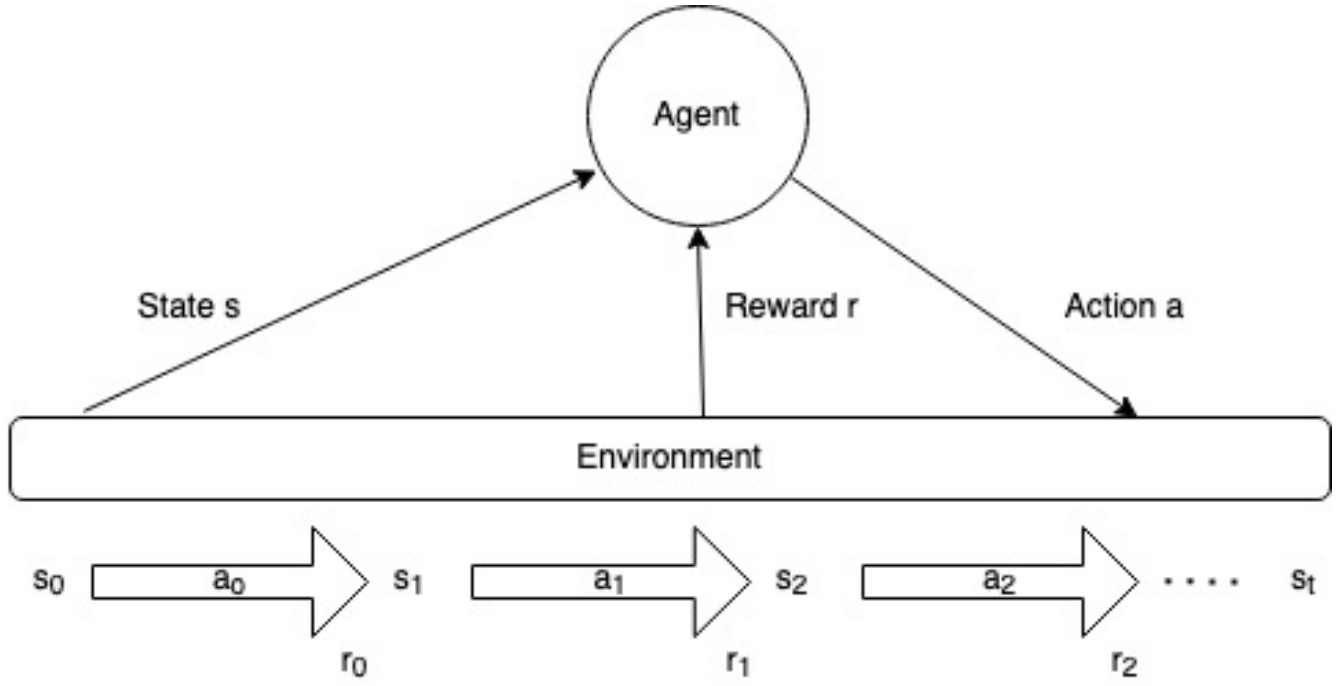
Environment and Agent

- Environment denotes an abstraction of the world that an agent operates within
  - Either real-life situations are constrained to solvable problems using robotics
  - OR environment is a simulation of a task
    - Defined by the tuple (Γ, Ω) where Γ is task space, Ω is distribution that provides the probability of a task $\gamma \in \Gamma$ to occur
- Intelligent agent is instantiation of RL algorithm that “intelligently” interacts with the environment and gain experience



Definition: Rewards R, States S, Actions A, transition probabilities T

- Reward $r \in R$ is a delayed and often sparse and noisy
- State $s_i \in S$ is a representation of the environment at step $i \in \mathbb{N}$ of the agent
- At each state s_i agent performs action $a_i \in A$ that changes the state $s = s_i$ to s_{i+1}
- $T = P(s_{i+1}|s_i)$ are transition probabilities that are unknown in model-free algorithms



Definition: Markov Decision Process (MDP)

- MDP allows to model *tasks* $\gamma \epsilon \Gamma$, i.e., decision making even in situations where outcome are partially random

An MDP is an *task, i.e.,* a 4-tuple (S,A,R,T) that for every state s_i fulfils the Markov property if

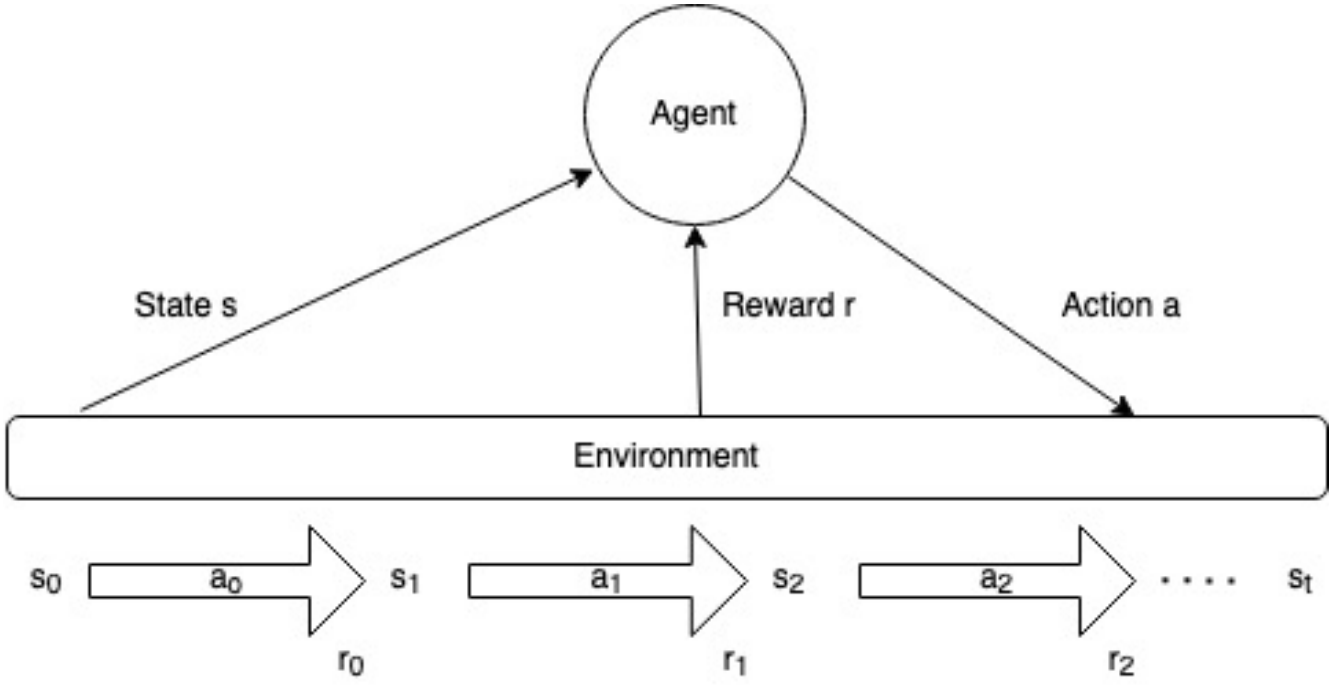
$$P(s_{i+1}|s_i) = P((s_{i+1}|s_1, \dots, s_i)$$

=> Transition from s_i to s_{i+1} is entirely independent of the past

=> effects of an action taken in a state depend only on that state but not prior states

Definitions: Policy π

- Policy is the probability $\pi(a|s)$ of an action given a state
  - $\pi: S \rightarrow A$ is *any* function of choosing an action in a given state
  - Fully defines the behavior of an agent (i.e., all possible actions) in environment
- After learning the optimal behavior is denotated with π^*



Two main approaches to solve the control problem of finding π^*

First

- MDP specifes setup for reinforcement learning

Then

- Policy-based agents will learn policy $\pi(a|s) = \pi(a|s,\vartheta)$ directly by optimizing parameter ϑ

- Value-based agents will learn either action-value function $Q_\pi(s,a)$

  OR state-value function $V_\pi(s)$

  given a behavior policy $\pi(a|s)$

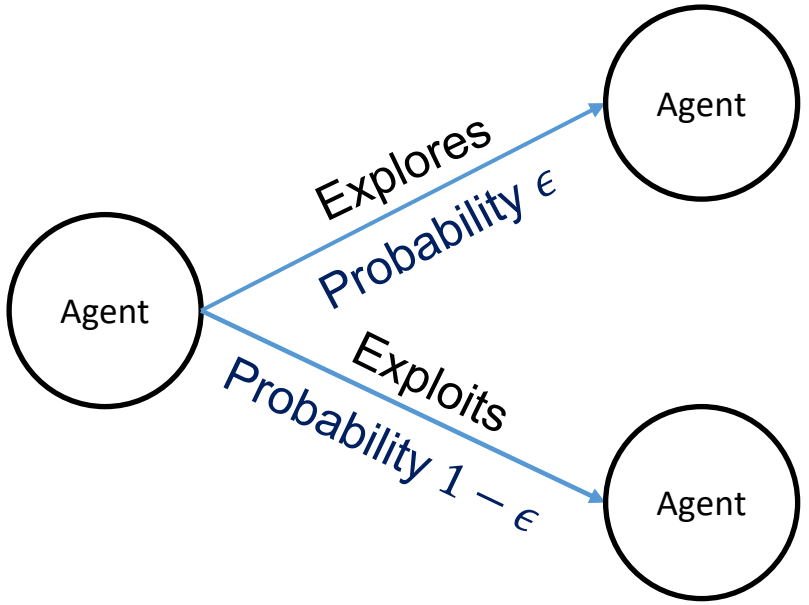
- What is a simple policy of an agent?

Try and error policy for learning π^* with greedy action selection

Exploration

- Agent tries new non-optimal action to
  - Learn reward
  - Gain better understanding of environment

  $\Rightarrow$ Random action with ϵ, $0 < \epsilon < 1$



Exploitation

- Use current experience of agent to choose optimal action $\pi(s|a) = \begin{cases} 1 \; if \; a = \max Q(s,a) \\ 0 \qquad otherwise \end{cases}$

  $\Rightarrow$ Greedy action with $1-\epsilon$

  $\Rightarrow$ Optimal in the sense that Q_π is maximized for selected a

- Basic improvements are annealing $\epsilon = \epsilon(eppoch)$, UCB and softmax

Definition: State-value and Action-value Functions

Let G be the return and E_π its expected value with regards to the policy of the agent π

- Action-value function $Q_\pi(s, a)$ is defined as

$$Q_\pi(s, a) = E_\pi[G|s, a]$$

  - includes all expected rewards until terminal state s_T given state s and taken action a

- State value function $V_\pi(s)$ is defined as

$$V_\pi(s) = E_\pi[G|s]$$

  - Gives expected returns starting from the state s and going to successor states thereafter

Definition: Return G

- Return is the sum of discounted rewards

$$G = \sum_{i=0}^{T} \gamma^i r_i$$

- „Rewards received *i* steps hence are worth less than rewards received now“ [Watkins & Dayan, 1992]

$$0 < \gamma \leq 1, \mathrm{T} < \infty$$

with T beeing the terminal step leading to the terminal state s_T

- Goal of the agent is to find a policy that maximizes the expected return and, thus, all rewards

Optimal Control: Learn π^* by Bellman Optimality

- Maximizing Q is equal to maximizing the expected return

$$Q_*(s) = arg\max_{\pi} Q_{\pi*}(s,a) = arg\max_{a_i \in A} E_{\pi}[G|s,a]$$

=> Policy obtains the highest reward if agent choses action *a* with highest expected return

- Optimal policy π^* also maximizes state value function $V_{\pi}(s)$ with

$$V_*(\mathrm{s}) = arg\max_{\pi} V_{\pi}(s) \quad \forall s$$

=> Value of the state under an optimal policy must be equal to G for next best action of that state [Sutton & Barto, 1998]:

$$V_*(s) = arg\max_{a_i \in A} Q_{\pi*}(s,a)$$

Q-learning [Watkins, 1989]

- Q-learning learns Q_* directly without modelling transition probabilities

$$Q_*(s) = arg\max_{\pi} Q_{\pi*}(s,a) = arg\max_{a_i \in A} E_{\pi}[G|s]$$

$\Rightarrow$Q learning learns $Q_*(s,a)$ using the Bellman optimality

$\Rightarrow$Agent has to have full experience about future rewards in G

Algorithm of Q learning

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(s,a)$

2: Initialize $Q(s,a) = 0$, for all $s_i \in S$, $a_i \in A$

3: Loop for each episode

4: Initialize environment to provide s

5: do

6: Choose a in state s using π breaking ties randomly

7: Take action a and observe r, s_{i+1}

8: Update Q(s,a) by Eq. 1

9: s= s_{i+1}

10: while s is not terminal

Line 1:
π is used to decide the probability of which action to take, e.g., greedy action selection

Line 2:
Initializes look−up table

Experience of an agent (discrete case)

- Q-learning learns Q_* directly without modelling transition probabilities or expected rewards of the MDP

$$Q_*(s) = arg\max_{\pi} Q_{\pi*}(s,a) = arg\max_{a_i \in A} E_\pi[G|s]$$

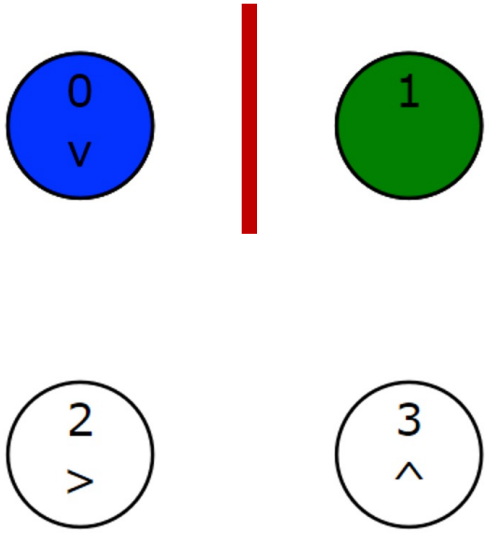
$\Rightarrow$Q learning learns $Q_*(s,a)$ using the Bellman optimality

$\Rightarrow$Agent has to have full experience about future rewards in G

$\Rightarrow$Experience has to be gained by trying out states/action combinations

$\Rightarrow$Experience of agent is stored state-action-table

State/action	a_1=up	a_2=left	a_3=down	a_4=right
Start s_0	0	/	0	0
Terminal s_1	0	0	0	/
s_2	0	0	0	0
s_3	0	0	0	0



Algorithm of Q learning

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(s,a)$

2: Initialize $Q(s,a) = 0$, for all $s_i \in S$, $a_i \in A$

3: Loop for each episode

4: Initialize environment to provide s

5: do

6: Choose a in state s using π breaking ties randomly

7: Take action a and observe r, s_{i+1}

8: Update Q(s,a) by Eq. 1

9: s= s_{i+1}

10: while s is not terminal

Line 3
Episode is the full experiment in the environment until a terminal state s_T

Line 4
Sets also agent to the starting state

Algorithm of Q learning

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(s,a)$

2: Initialize $Q(s,a) = 0$, for all $s_i \in S$, $a_i \in A$

3: Loop for each episode

4: Initialize environment to provide s

5: do

6: Choose a in state s using π breaking ties randomly

7: Take action a and observe r, s_{i+1}

8: Update Q(s,a) by Eq. 1

9: s= s_{i+1}

10: while s is not terminal

Line 6: Agent decides which action to take

Line 7: Agent observes effects of action

Algorithm of Q learning

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(s,a)$

2: Initialize $Q(s,a) = 0$, for all $s_i \in S$, $a_i \in A$

3: Loop for each episode

4: Initialize environment to provide s

5: do

6: Choose a in state s using π breaking ties randomly

7: Take action a and observe r, s_{i+1}

8: Update Q(s,a) by Eq. 1

9: s= s_{i+1}

10: while s is not terminal

Line 8:
Expected return G is calculated as the sum of the current reward and the expected value of the next state and used to update Q in lookup table
How?

Q learning: Value Iteration by Update Rule

- Iterates estimate Q(s,a) by overwriting previous estimate in look up table
  - For non terminal state

$$Q(s_i, a) = Q(s_i, a) + \alpha \left(r + \gamma * \underbrace{arg\max_{a_s \in A} Q(s_{i+1}, a_s)}_{\text{Estimate of future rewards based on current experience}} - Q(s_i, a) \right) \quad \text{Eq. 1}$$

Learning rate
$0 < \alpha \leq 1$

Discount factor
$0 < \gamma \leq 1$

$G_{i:i+1}$ one-step estimated expected return (Bellman equation)

TD error δ_i for one-step ahead Q-learning at step *i* available at time step i+1

  - For terminal state $Q(s_T, a) = r$

Algorithm of Q learning

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(s,a)$

2: Initialize $Q(s,a) = 0$, for all $s_i \in S$, $a_i \in A$

3: Loop for each episode

4: Initialize environment to provide s

5: do

6: Choose a in state s using π breaking ties randomly

7: Take action a and observe r, s_{i+1}

8: Update Q(s,a) by Eq. 1

9: s= s_{i+1}

10: while s is not terminal

- Update is performed indepentently of the policy used by agent

⇒ Update is computed using the best action for a_{i+1}

=> **off-policy** alfgorithm

Algorithm of SARSA $(s_i, a_i r_i . s_{i+1}, a_{i+1})$

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(s,a)$

2: Initialize $Q(s,a) = 0$, for all $s_i \in S$, $a_i \in A$

3: Loop for each episode

4: Initialize environment to provide s

5: do

6: Choose a in state s using π breaking ties randomly

7: Take action a and observe r, s_{i+1}

7b: Choose action a_{i+1} from s_{i+1} using π

8: Update $Q_{a_{i+1}}$(s,a)

9: s= s_{i+1}

10: while s is not terminal

Q-learning is changed & extended by marked lines

Line 7b:

- Suboptimal action a_{i+1} is taken

Line 8:

- Update uses behavior policy to select a_{i+1}

=> **on-policy** alfgorithm

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s_{i+1}.a_{i+1}) - Q(s,a))$$

Q-Learning is TD(0) Learning with Sample Update

Extension of Q-learning can be seen as N-step temporal difference learning (TD)

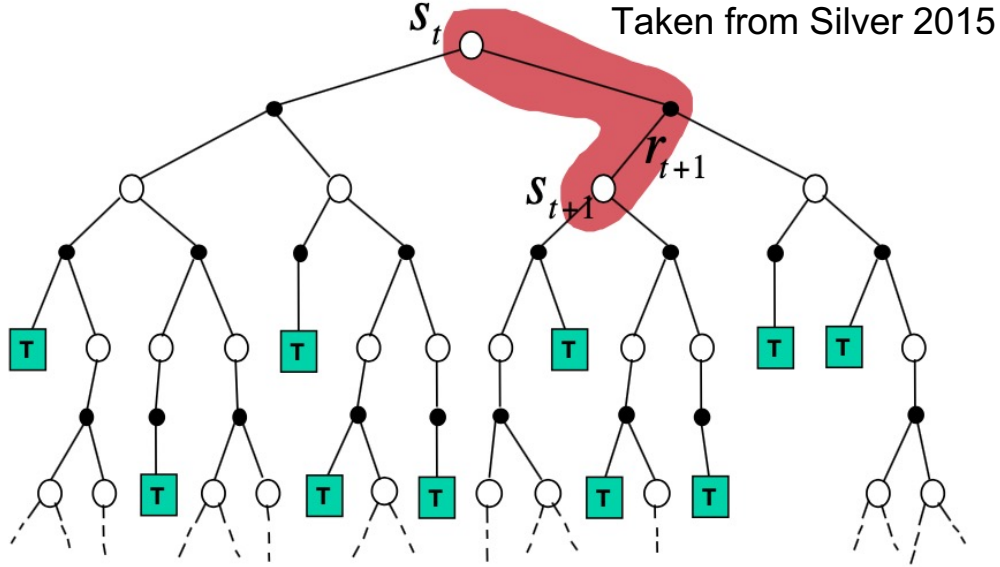
- Agent can buffer the rewards more than one step and take multiple steps forward until updating the lookup table

$$Q_{i:i+1}(s_i, a) = Q(s_i, a) + \alpha\left(r + \gamma * arg\max_{a_s \in A} Q(s_{i+1}, a_s) - Q(s_i, a)\right)$$

$$Q_{i:i+n}(s_i, a) = Q(s_i, a) + \alpha\left(G_{i:i+n} - Q(s_i, a)\right)$$

$$G_{i:i+n} = r + \gamma r_{i+1} + \cdots + \gamma^{n-1} r_{i+n-1} + \gamma^n * arg\max_{a_s \in A} Q(s_{i+n}, a_s)$$

$$0 \leq i \leq T - n$$



Taken from Silver 2015

Improvements of Q-Learning

- Replace look-up table with approximate function

$\Rightarrow$ Agent looks at the current state-action pair and predicts the expected value within an regression method

$\Rightarrow$ BUT: state-action pairs tend to be nonlinear and discontinuous

$\Rightarrow$ Use neural networks [Tesauro, 1995]

$\Rightarrow$ Deep Q learning translates raw observations into actions [Mnih et al.,2013]

Basic Algorithm of Deep Q-learning [Mnih et al., 2013]

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(a, s, \vartheta_i)$

2: Initialize $Q(a, s, \vartheta_i)$ with random weights and replay memory D_N

3: loop for each episode

4: Initialize environment to provide s and preprocessed $\varphi(s)$

5: do

6: Choose a in state s using π breaking ties randomly

a7: Take action a and observe r_i, s_{i+1}

b7: Store transition e_i in D_N

c7: Sample random minibatch of transition $e_j \in D_N$

a8: Set target y

b8: Update $Q(a, s, \vartheta_i)$ by $\nabla_\vartheta L(\vartheta_i)$

9: s= s_{i+1}

10: while s is not terminal

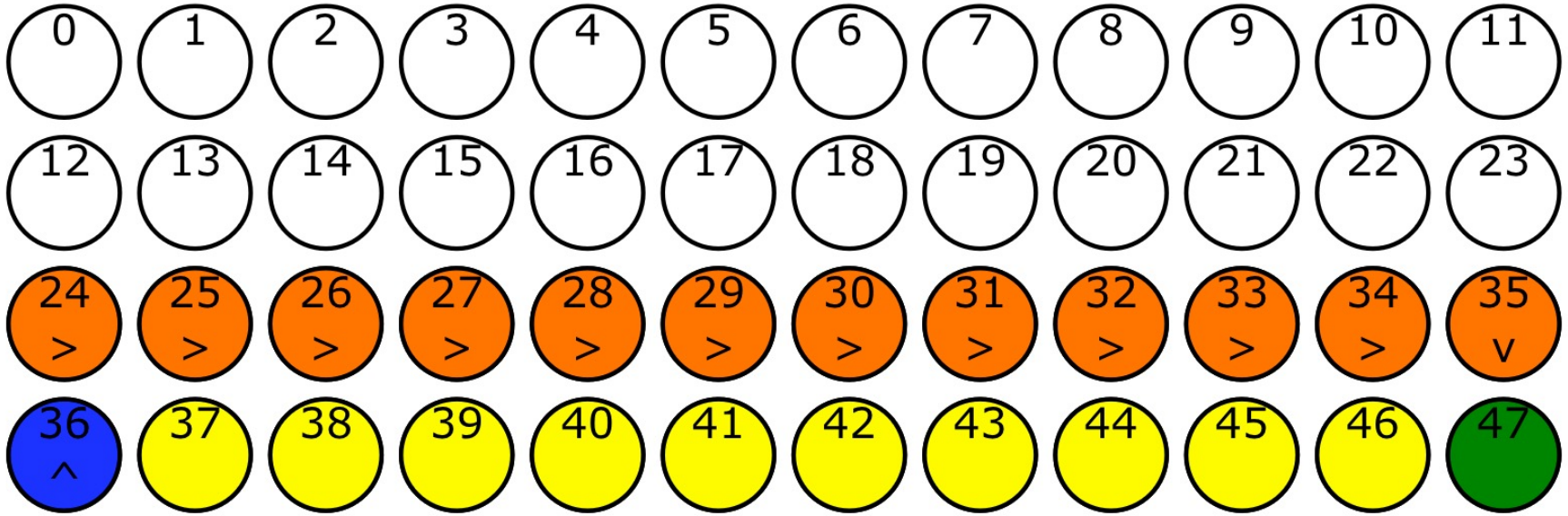
Q-learning is changed & extended by marked lines

Through $\varphi(s)$ the environment definition is more elaborate to exploit standard deep learning

Q-learning is updated by gradient descent

Application: Bypass a cliff

- $s_o, \ldots, s_{47}$ states in white with four actions per state: up, down, right left
- Yellow Cliff has reward r=-200 and stepping off the cliff sends agent back to **blue** start
- No discount with $\gamma = 1$
- Each step has reward r=-1 => maximal reward is -13 (**orange**)
- **Green** goal has reward r=0
- Approximation of π^* will vary depending on selected policy π, parameters and learning rate

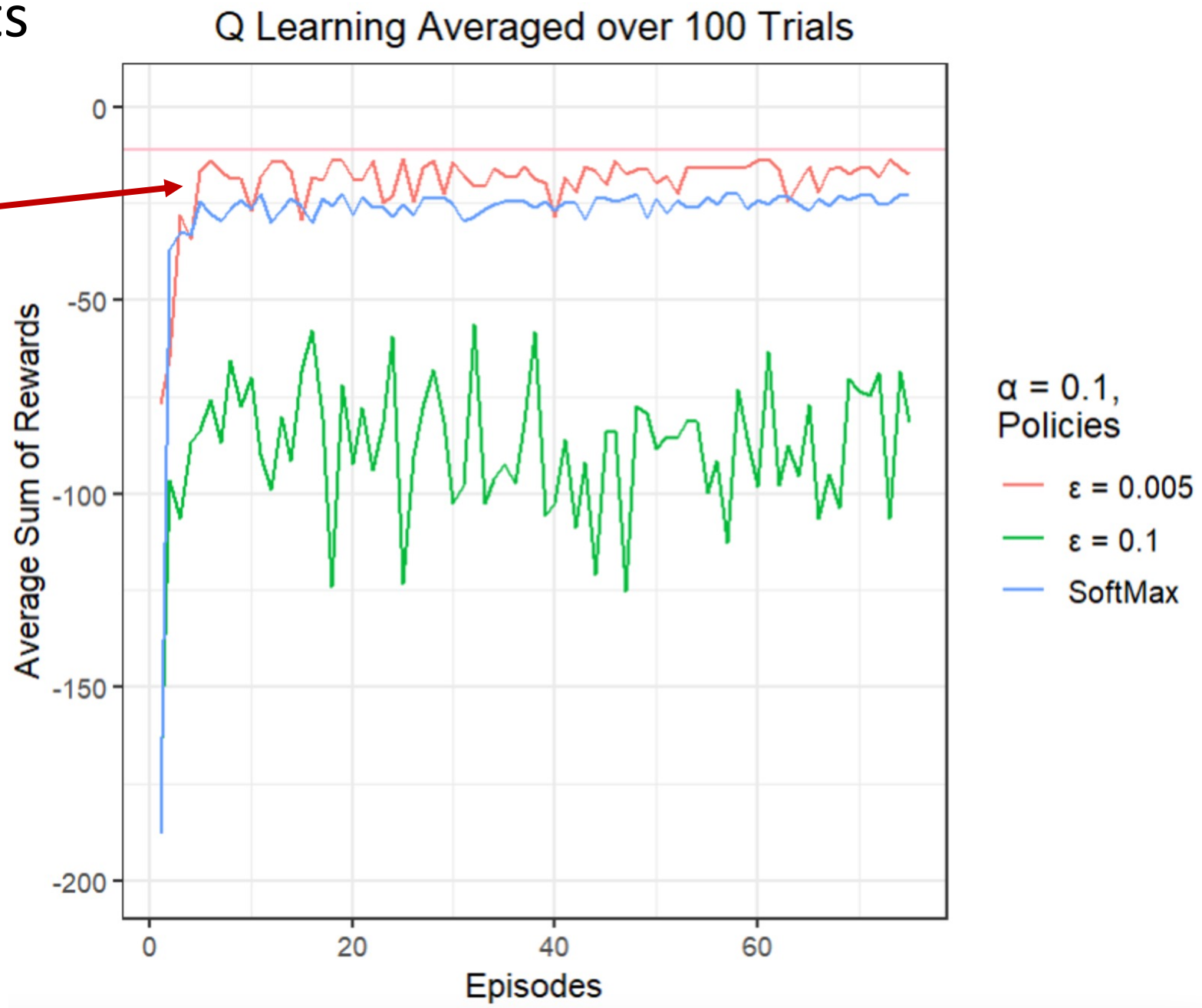


Typical Learning effects

- Agent converges but never stabilizes („chattering“)

  => optimal path is near cliff

  => Chance to be kicked off the cliff by ϵ



Typical Learning effects

- Agent converges but never stabilizes („chattering“)

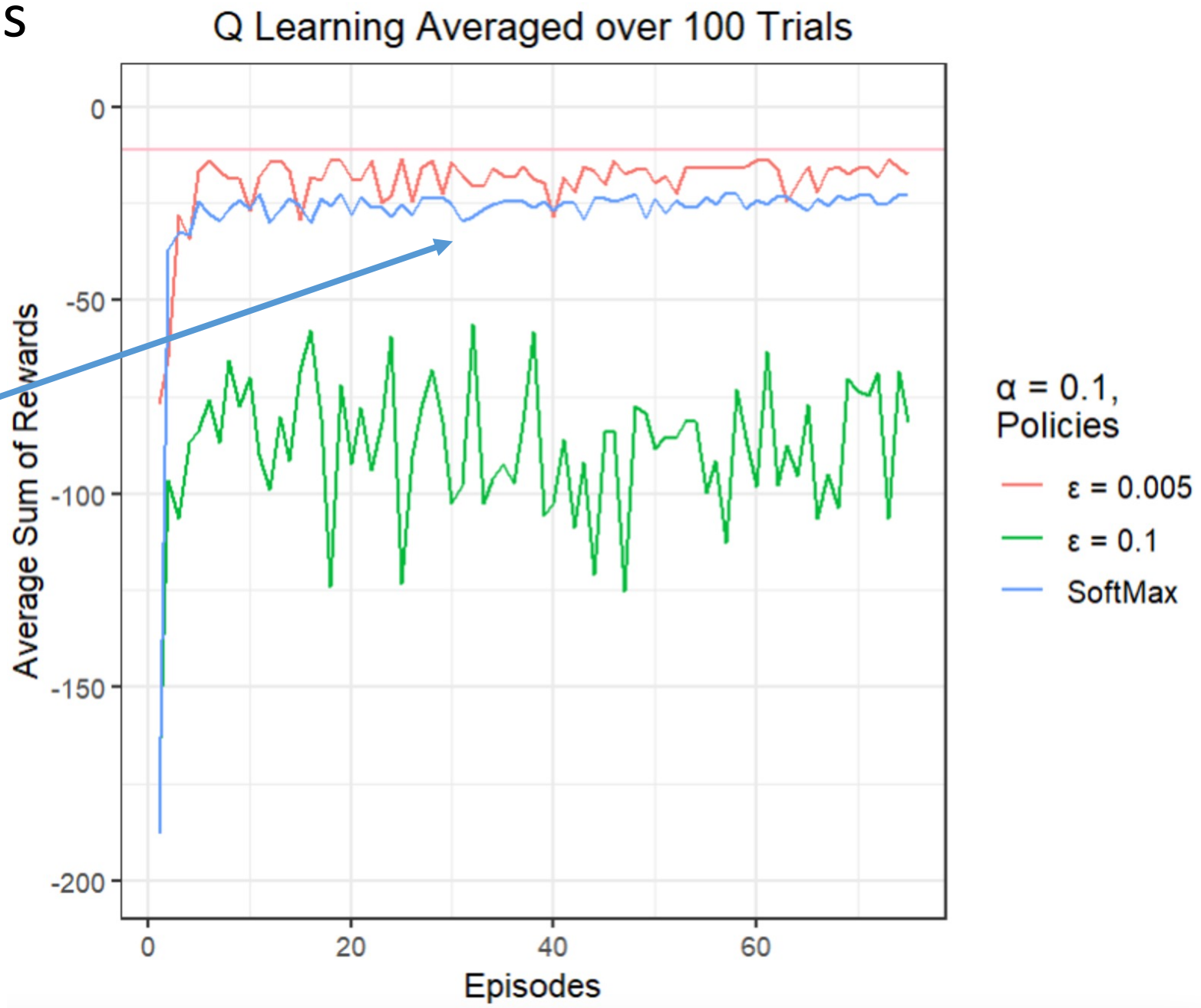
  => optimal path is near cliff

  => Chance to be kicked off the cliff by ϵ

- Policy converges (slightly) below optimum

$\Rightarrow$ Possibility to find local optima

$\Rightarrow$ There may be more than one near optimal policy



Typical Learning effects

- Agent converges but never stabilizes („chattering“)

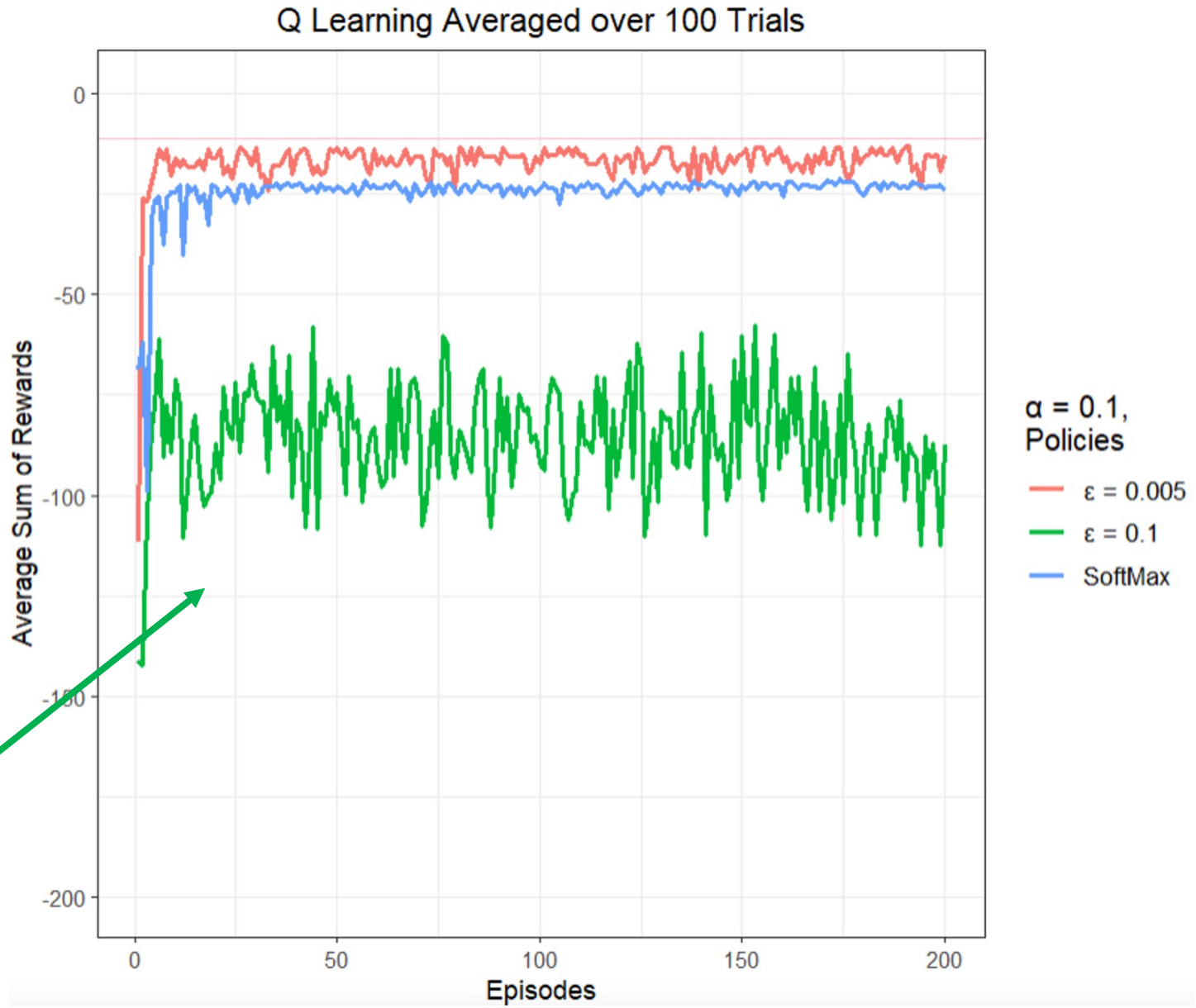
  => optimal path is near cliff

  => Chance to be kicked off the cliff by ϵ

- Policy converges (slightly) below optimum

$\Rightarrow$ Possibility to find local optima

$\Rightarrow$ There may be more than one near optimal policy



Policy never converges

Limitations of Q-Learning I

- *Setting „balance“* between exploration and exploitation is challenging
  - Too much exploration,e.g, “agent jumps of the cliff”
    => “jittered” convergence
  - Too much exploitation, e.g.,“long route is taken”
    => policy converges non-optimally or overfitting

- Value-based methods like Q-learning maximize the exploitation part of RL
  - BUT: Exploration is only ensured by external parameter (e.g., ϵ) to be set by user
  - One Solution is learning appropriate *“balance”* via entropy
  - ➢ Maximum entropy RL learns by itself how much exploration it needs to learn appropriately, e.g.,
    - Soft Q-learning [Ziebart et al., 2008]

Demonstrating Overfitting

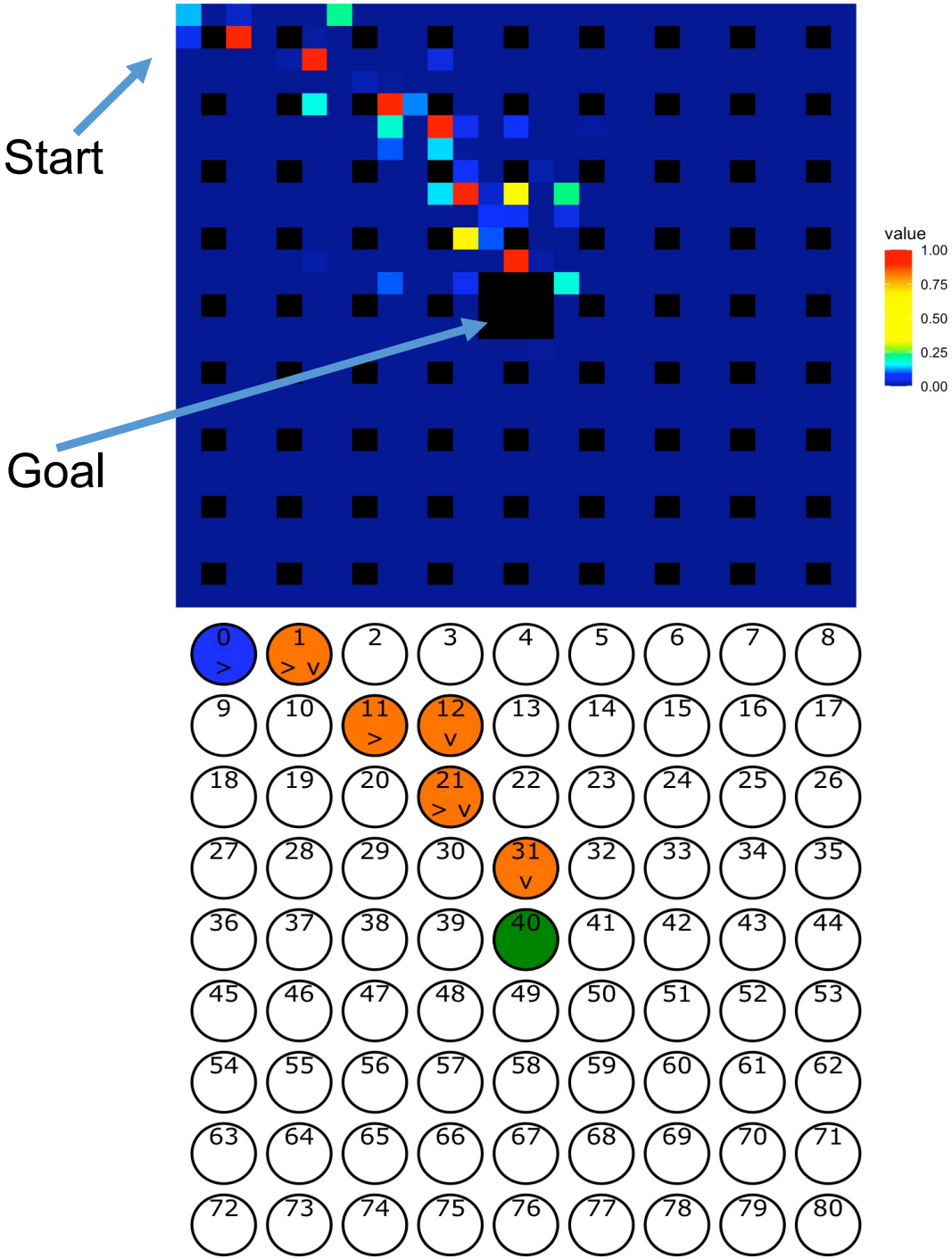
- Grid world with ε greedy exploration
- 8 discrete actions are allowed
  - Diagonal movements are allowed

Lookup table is visualized as follows

- Black=States
- Color=Value of estimated Q
- Position of Pixel=movement position

$\Rightarrow$ Blue= actions & states not sampled by agent so far

$\Rightarrow$ Yellow/Red high action states defining the non-optimal trajectory the agents follows



Q learning with greedy policy

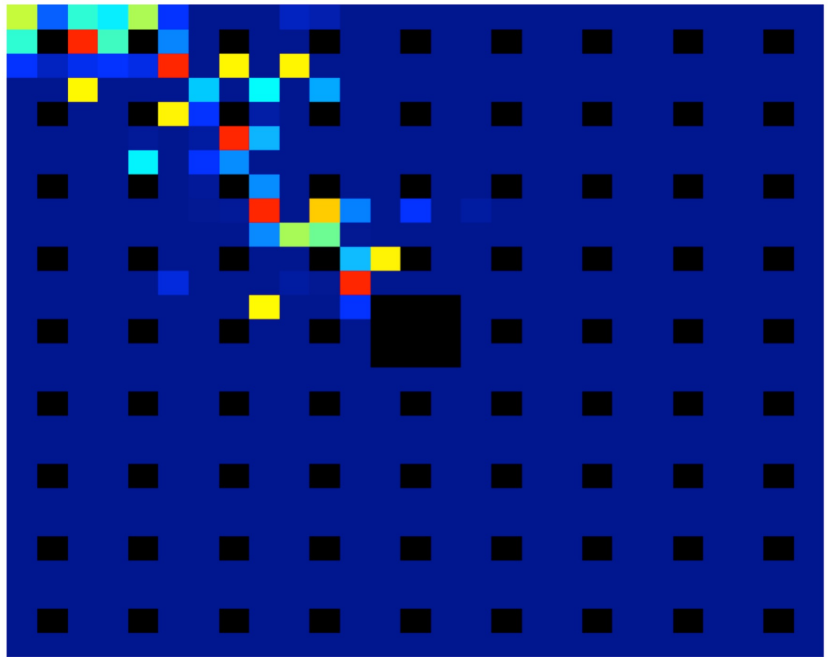
Demonstrating Overfitting

- Process of accidentally building model that works on very specific circumstances, e.g.
  - Policy π^* has uncharted regions near optimal trajectory
  - ➢ Agent did not explore the environment enough
- Modelling actions as a distribution with softmax function results in more explored states

➢ In model changes in start position still enable agent to find goal

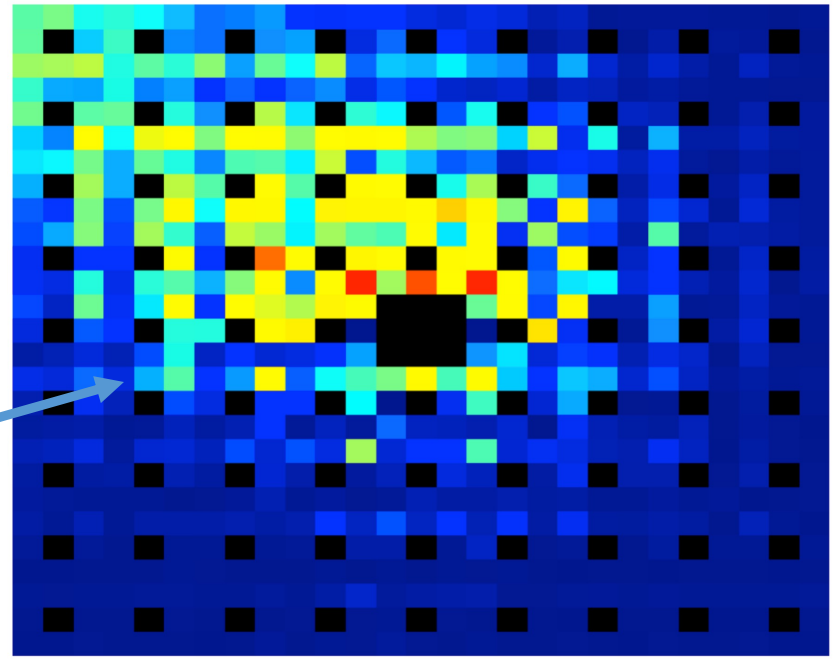
➢ Exploration took 10 times longer yet still left states unexplored

➢ Model is more robust



Q learning with greedy policy

-converged after 10 eppochs lerned



Q-learning with softmax policy

-converged after 100 eppochs lerned

Limitations of Q-Learning II

- Methods using look-up tables like Q-learning struggle when the state or action space is large
  - Agent has to sample a huge number of states and every action to find a good policy
    - Learning becomes very slow
    - Alternative solutions are, e.g., using information loss [Hoof, Neumann & Peters, 2017] or experience replay [Lazaric et al., 2008]

- Appropriate Parameter settings are unknown and depend on environment, rewards and policy
  - How many steps n should the n-step Q-learning take until Q update?
  - Alternatively with eligibility traces the agent can look backward in time, but how to set λ?
  - How should the learning rate α be set?

Q-learning Alternatives

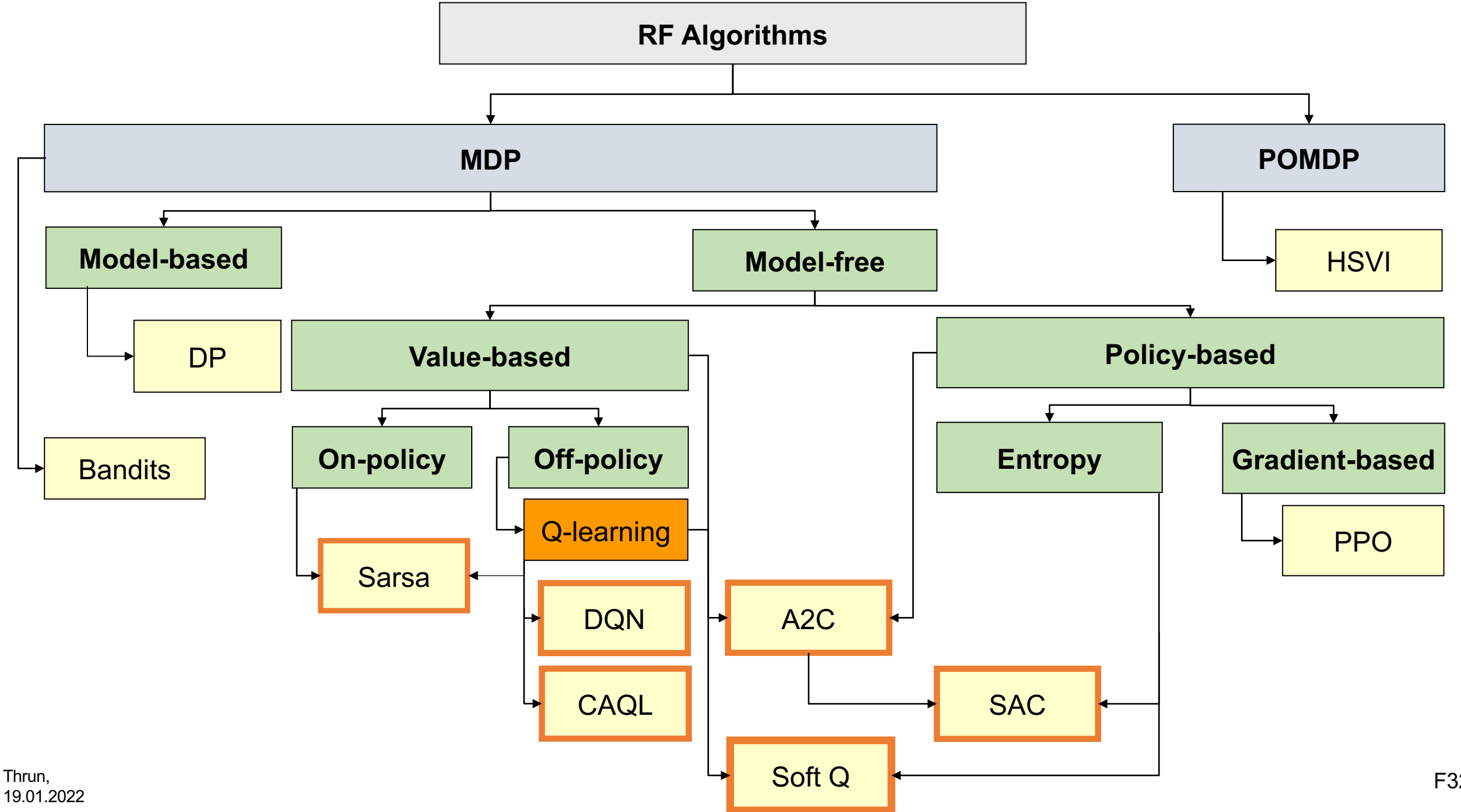
- Major problem can be tackled with an “divide and conquer” approach

  => Hierarchical RL
- One agent learns, otherwise use e.g.,
  - Multi-agent RL (MARL)
  - Proximal Policy Optimization (PPO) [Schulman et al., 2015; Schulman et al. 2017]
- Actions are discrete, otherwise use e.g.,
  - Continous Action Q-Learning (CAQL) [Ryu et al., 2020]
  - Soft Actor-Critic (SAC) [Haarnajoa et al.,2017]
- Environment has a terminal state –> ends after an epsiode
  - Alternative: Continuous Tasks for life-long learning
- If each episode consists only of one time step => multi-armed bandit
- Expectation value of Q is the moving average
  - For example it is multimodal for cliff-wall scenarios!

  => Distributional RL

Using Assumptions about MDP to Structure RL types

- Transition probabilities T are unknown => model-free algorithms
  - Otherwise model-based RL algorithms
- States s are fully observable, otherwise use
  - Algorithms for partially observable MDP (POMDP)
- Agent learns one policy and algorithm is based on one given policy, otherwise
  - Advantage actor critic (A2C) [Barto,Sutton&Anderson, 1983; Kimura & Kobayashi ,1998]
- States S are discrete, otherwise use
  - Function aproximators with deep learning (DQN) [Mnih et al., 2013]
- Q-learning: Agent learns value-based (i.e., Q or V)
  - Alternative: policy-based algorithms learn π directly
- Policy is deterministic, otherwise
  - Stochastic policy learning


RF Algorithms
MDP
POMDP
Model-based
Model-free
HSVI
DP
Value-based
Policy-based
Bandits
On-policy
Off-policy
Entropy
Gradient-based
Q-learning
PPO
Sarsa
DQN
A2C
CAQL
SAC
Soft Q

Summary

- RL problems are sequential contrary to ML
- Policy is responsible for mapping representation of state to action
  - Vast majority of RL literature is dedicated to finding improved ways of defining policies or improving some theoretical guarantees
- Balance exploitation vs. exploration

  $\Rightarrow$ Many algorithms provide elaborate solutions (e.g. SAC)
- Consider varying the environment to help algorithms generalize (not to overfit)
- Some part of the MDP can always be modeled by supervised or unsupervised ML aproaches, e.g.,
  - Advantage: Deep learning does not require time-consuming state engineering
  - Disadvange: There is no proof that RL will converge in this case!

Practical Challenges

- Learning is the bottleneck of RL because it costs time or/and money

=> Constrain problem definition (and thus the exploration space) until a valid MDP can be defined

  - Depending on MDP select RL algorithm using Ockham's razor
  - ➢ Start simple from straighforward Q-learning and then select "addons"
    - Very vast number of literature is about some combinations of such addons

- BEWARE: RL aims to find a policy that maximizes the expected return irrespective of consequences

  ⇒ Rewards should be as near as possible to the application (e.g., € in business decisons)

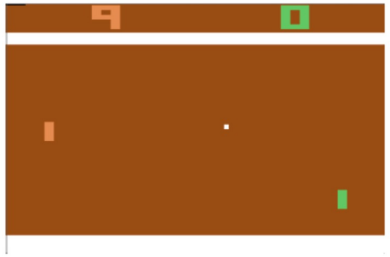
  ⇒ Hypothesis is that all goals can be described in such a way!

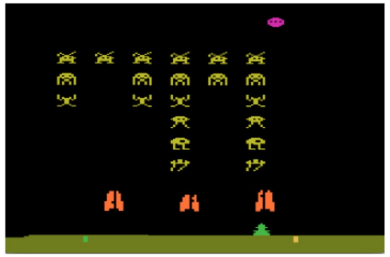
Sources

Literature, Code & Slide accessible in

https://github.com/Mthrun/RLwithAppl_IntAgents/

Deep Q Learning on Atari Games





- Reward is current game score
- Actions are defined by game with a probability
- Environment is Atari emulator
  - Agent observes image (s=vector of pixels) from emulator representing the current screen
    - 4 frames are preprocesses $\varphi(s)$ and combined to one input to Q-function
- $Q(a, s, \vartheta) \approx Q(a, s)$ is the function approximator, ϑ are weights of the deep learning network with loss function

$$L_i(\vartheta_i) = E_\pi \left[\left(y_i - Q(a, s, \vartheta_i) \right)^2 \right]$$

With the target

$$y_i = E_\pi \left[r + \gamma * arg\max_{a_s \in A} Q(s_{i+1}, s, \vartheta_{i-1}) \right]$$

Where weights of previous iteration are fixed

=> BUT Target is in contrast to ML not fixed

Replay memory

- Problem: Supervised ANN algorithms assume I.I.D
- BUT: RL has highly corelated sequence of states and data distribution changes as agent learns
- Solution: Use replay memory D [Lohng-ji, 1993] that takes sample of previos transitions e

$$e_i = (\varphi_i, a_i, r_i, \varphi_{i+1})$$
$$D_N = \{e_1, \dots e_N\}, N = 10^6$$

- Sampling technique samples depending on the action, e.g.,
  - Action is move right than samples are dominated by right-hand side

- Mnih et al, 2013 claims that this smoothes out learning and avoides divergence and chattering

Basic Algorithm of Deep Q-learning

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(a, s, \vartheta_i)$

2: Initialize $Q(a, s, \vartheta_i)$ with random weights and replay memory D_N

3: loop for each episode

4: Initialize environment to provide s and preprocessed $\varphi(s)$

5: do

6: Choose a in state s using π breaking ties randomly

a7: Take action a and observe r_i, s_{i+1}

b7: Store transition e_i in D_N

c7: Sample random minibatch of transition $e_j \in D_N$

a8: Set target y

b8: Update $Q(a, s, \vartheta_i)$ by $\nabla_\vartheta L(\vartheta_i)$

9: s= s_{i+1}

10: while s is not terminal

Q-learning is changed & extended by marked lines

Through $\varphi(s)$ the environment definition is more elaborate to exploit standard deep learning

Q-learning is updated by gradient descent

Basic Algorithm of Deep Q-learning

1: **Input** is a policy $\pi(a|s)$ that uses the action-value function $Q(a, s, \vartheta_i)$

2: Initialize $Q(a, s, \vartheta_i)$ with random weights and replay memory D_N

3: loop for each episode

4: Initialize environment to provide s and preprocessed $\varphi(s)$

5: do

6: Choose a in state s using π breaking ties randomly

a7: Take action a and observe r_i, s_{i+1}

b7: Store transition e_i in D_N

c7: Sample random minibatch of transition $e_j \in D_N$

a8: Set target y

b8: Update $Q(a, s, \vartheta_i)$ by $\nabla_\vartheta L(\vartheta_i)$

9: s= s_{i+1}

10: while s is not terminal

Most important knack is to add replay memory D_N

Replay memory

- Stores given variables for a specific iteration
- Takes sample for computing a8-b8

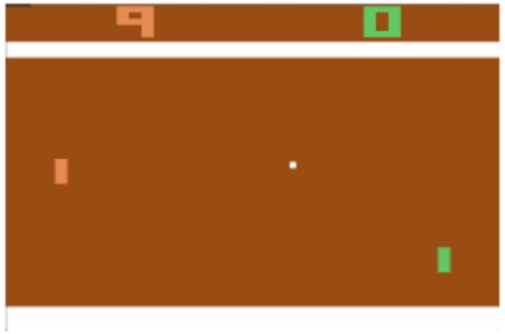
Results Deep Q Learning on Atari Games [Mnih et al., 2013]

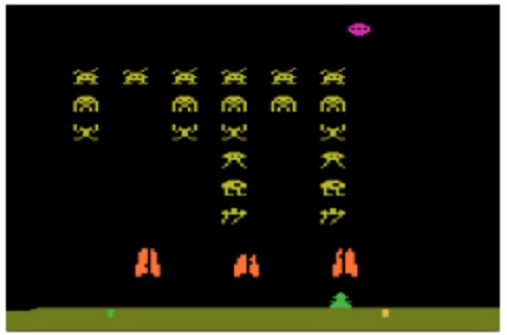
- Total average reward by running ϵ greedy policy 4 Million frames
- DQN outperforms standard RL with elobarate manual feature engineering for states
- DQN outperforms even human expert in three games

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690

$\Rightarrow$ Deep learning allows a automated feature extraction given a suitable problem for the ANN architecture

- Using 6 prior extensions of Q-learning rainbow DQN extended DQN [Hessel et al., 2017]






Convergence Proofs

- Q-learning converges [Watkins & Dayan, 1992], so long as
  - All actions are repeatedly sampled in all states
  - Action-values are represented discretely
- Basic n-step Q-learning converges [Tsitsiklis & Van Roy, 1999]
- Given policies of continouus states policies must approximate actions
  - Linear approximations are always convergent [Sutton & Barto, 1998]
  - Non-parametric policies converge [Sutton & Barto, 1998, Hoof, Neumann & Peters, 2017]
  - Non-linear approximations (e.g. neural networks) have no guarantees to converge [Silver 2015, Winder, 2020]
    - In ML, Ben-David et.al.,2019 proofs that learnibility is undecidable

[Silver,2015] claims...

- "TD does not follow the gradient of any objective function "
  - "This is why TD can diverge when off-policy or using non-linear function approximation "

- Gradient TD follows true gradient of projected Bellman eror

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Algorithm	Table Lookup	Linear	Non-Linear	
Monte-Carlo Control	✓	(✓)	✗	(chattering)
Sarsa	✓	(✓)	✗	
Q-learning	✓	✗	✗	
Gradient Q-learning	✓	✓	✗	

UCB and Softmax function

Softmax

- It is often better to choose the action based on the current estimates of the distribution of rewards
  - Rather than returning a single action, the agents returns probabilities of each action weighted by expected rewards of each states
    - Action with highest preference h has the highest probability of beeing selected

$$\pi(s|a,\vartheta) = \frac{e^{h(s,a,\vartheta)}}{\sum_b e^{h(s,b,\vartheta)}}, h \in \mathbb{R} \text{ preference for a state-action pair}$$

- Another option is upper confidence bound (UCB)
  - Agent does not explore randomly but explores specifically states it has not seen before
  - UCB adds bonus to each action for inadequately sampled states

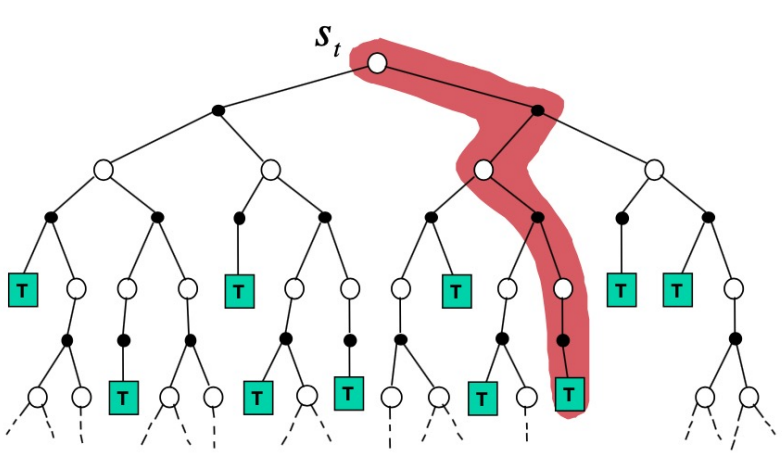
Monte-Carlo (MC) vs Dynamic Programming DP approaches

- MC is modell-free and samples
  - MC techniques choose a single action but have to simulate full episode
- MC learns directly from experience but needs complete episodes to learn

  => terminal state required in MDP
- DP is model-based as it requires all transition probabilities and does not sample
  - DP methods look one step ahead and iterate over all actions

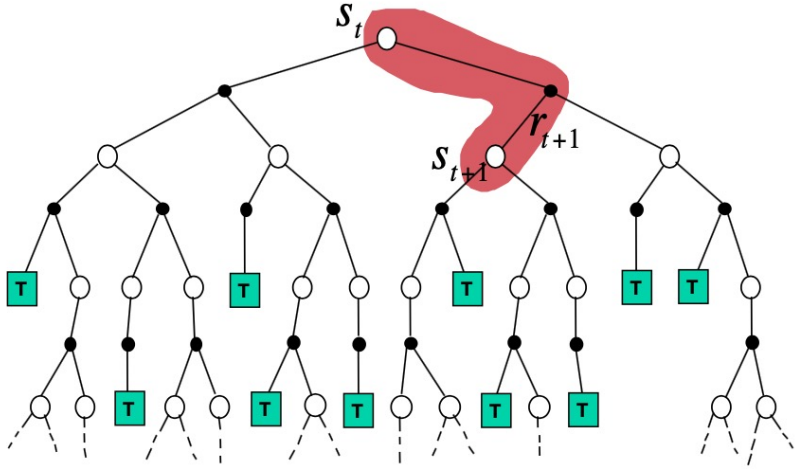
MC

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t - V(S_t)\right)$$



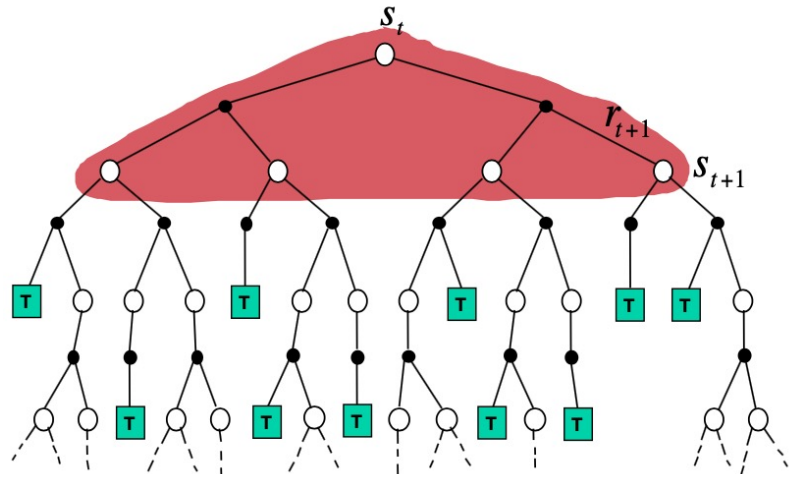
TD

$$V(S_t) \leftarrow V(S_t) + \alpha\left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$



DP

$$V(S_t) \leftarrow \mathbb{E}_\pi\left[R_{t+1} + \gamma V(S_{t+1})\right]$$



Taken fron Silver 2015

Benchmarking Algorithms – Policy Gradient (PG) vs Q-Leanring

- Research showed that soft Q-learning algorithms performed equivalent to policy grading algorithms, e.g., [Schulman et al., 2018]
  - Q learning increases the value of an action
  - Policy gradients increase the probabiliy of selecting an action
    - Entropy-based policy gradients boost the probability of an action proportional to a function f(a)

=> Both solve the same problem

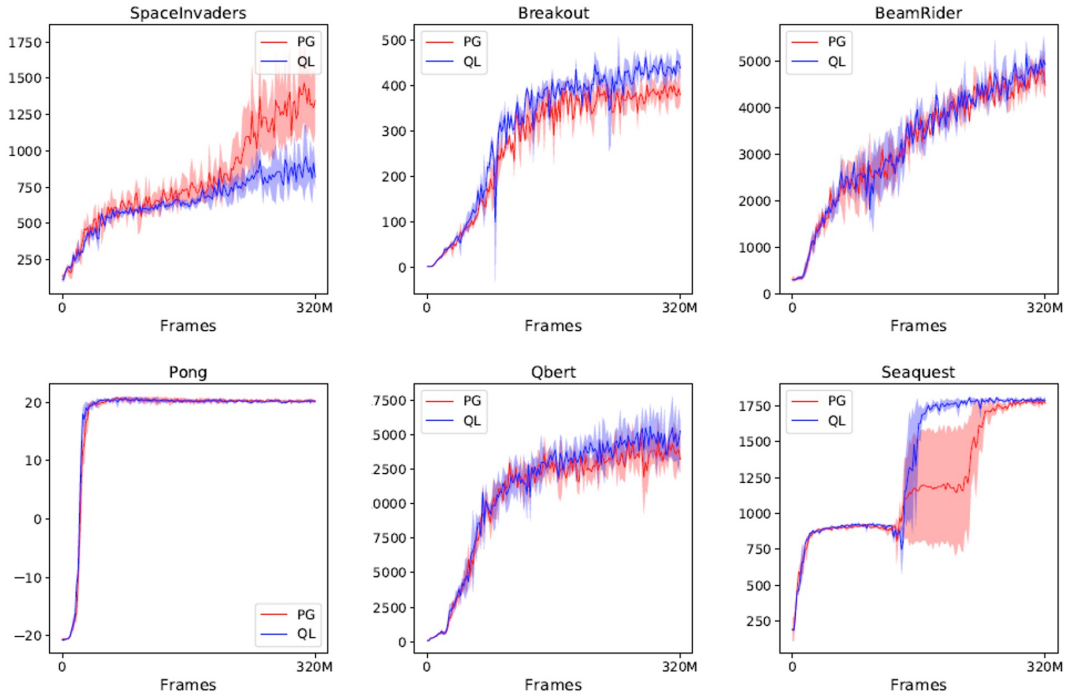


Figure 3: Atari performance with policy gradient vs Q-learning update rules. Solid lines are average evaluation return over 3 random seeds and shaded area is one standard deviation.

Benchmarking for continous control taks by [Winter, 2020]

- Challenge: Researchers mostly publish plots and Winter had to read the Sum of expected returns
- Result: SAC and PPO perform over a significant number of studies similar

Table 7-1. Performance results from continuous control benchmarks for SAC and PPO

Environment	SAC[a]	SAC[b]	SAC†[c]	SAC-auto*[d]	PPO[e]	PPO†[f]	SAC-auto[g]	PPO[h]
MuJoCo Walker	3475	3941	3868	4800	3425	3292		
MuJoCo Hopper	2101	3020	2922	3200	2316	2513		
MuJoCo Half-cheetah	8896	6095	10994	11000	1669			
MuJoCo Humanoid	4831		5723	5000	4900	806		
MuJoCo Ant	3250	2989	3856	4100				
MuJoCo Swimmer			42		111			
AntBulletEnv-v0							3485	2170
BipedalWalker-v2							307	266
BipedalWalkerHardcore-v2							101	166
HalfCheetahBulletEnv-v0							3331	3195
HopperBulletEnv-v0							2438	1945
HumanoidBulletEnv-v0							2048	1286
InvertedDoublePendulumBulletEnv-v0							9357	7703
InvertedPendulumSwingupBulletEnv-v0							892	867
LunarLanderContinuous-v2							270	128
MountainCarContinuous-v0							90	92
Pendulum-v0							-160	-168
ReacherBulletEnv-v0							18	18
Walker2DBulletEnv-v0							2053	2080

[a] Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor" (*https://oreil.ly/JjjwB*). ArXiv:1801.01290, August.

[b] Wang, Tingwu, Xuchan Bao, Ignasi Clavera, Jerrick Hoang, Yeming Wen, Eric Langlois, Shunshi Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. 2019. "Benchmarking Model-Based Reinforcement Learning" (*https://oreil.ly/9ZBoU*). ArXiv: 1907.02057, July.

[c] Wang, Che, and Keith Ross. 2019. "Boosting Soft Actor-Critic: Emphasizing Recent Experience without Forgetting the Past" (*https://oreil.ly/eigQt*). ArXiv:1906.04009, June.

[d] Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, et al. 2019. "Soft Actor-Critic Algorithms and Applications" (*https://oreil.ly/s3apP*). ArXiv:1812.05905, January.

[e] Dhariwal, Prafulla, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. OpenAI Baselines (*https://oreil.ly/9642Z*). GitHub Repository.

[f] Logan Engstrom et al. 2020. "Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO" (*https://oreil.ly/pOPdK*). ArXiv:2005.12729, May.

[g] Raffin, Antonin. (2018) 2020. Araffin/RI-Baselines-Zoo. Python. *https://oreil.ly/H3Nc2*

[h] Raffin, Antonin. (2018) 2020. Araffin/RI-Baselines-Zoo. Python. *https://oreil.ly/H3Nc2*

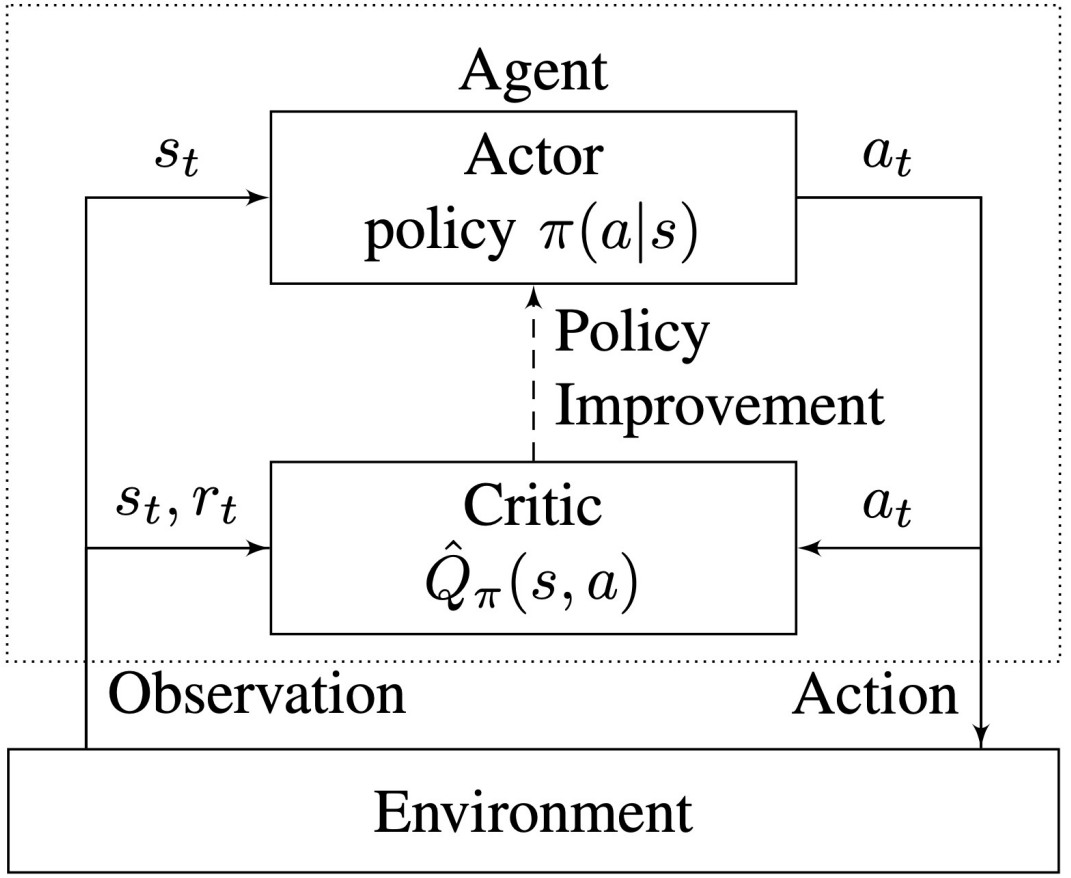
Policy-based algorithms I: Advantage Actor Critic (A2C)

- Basic idea published in [Barto,Sutton&Anderson, 1983] and refined in [Kimura & Kobayashi ,1998]
- Actor refers to the learned policy
- Critic refers to the learned value function with parameters w
- Policy $\pi(a|s) = \pi(a|s, \vartheta)$ is a probability function differentiable w.r.t. its parameters ϑ

=> Continous actions are easy to implement

- Policy gradient update rule is defined by

$$\vartheta = \vartheta + \alpha\gamma^i B\nabla_\vartheta \ln \pi(a|s, \vartheta)$$



Source: [Kämmerer, 2019] cites [Kimura & Kobayashi ,1998]

Policy-based algorithms I: Advantage Actor Critic (A2C)

- Policy gradient update rule is defined by
$$\vartheta = \vartheta + \alpha_\vartheta \gamma^i B_w \nabla_\vartheta \ln \pi(a|s,\vartheta)$$
- $B = G$
- => reinforce algorithm & no baseline

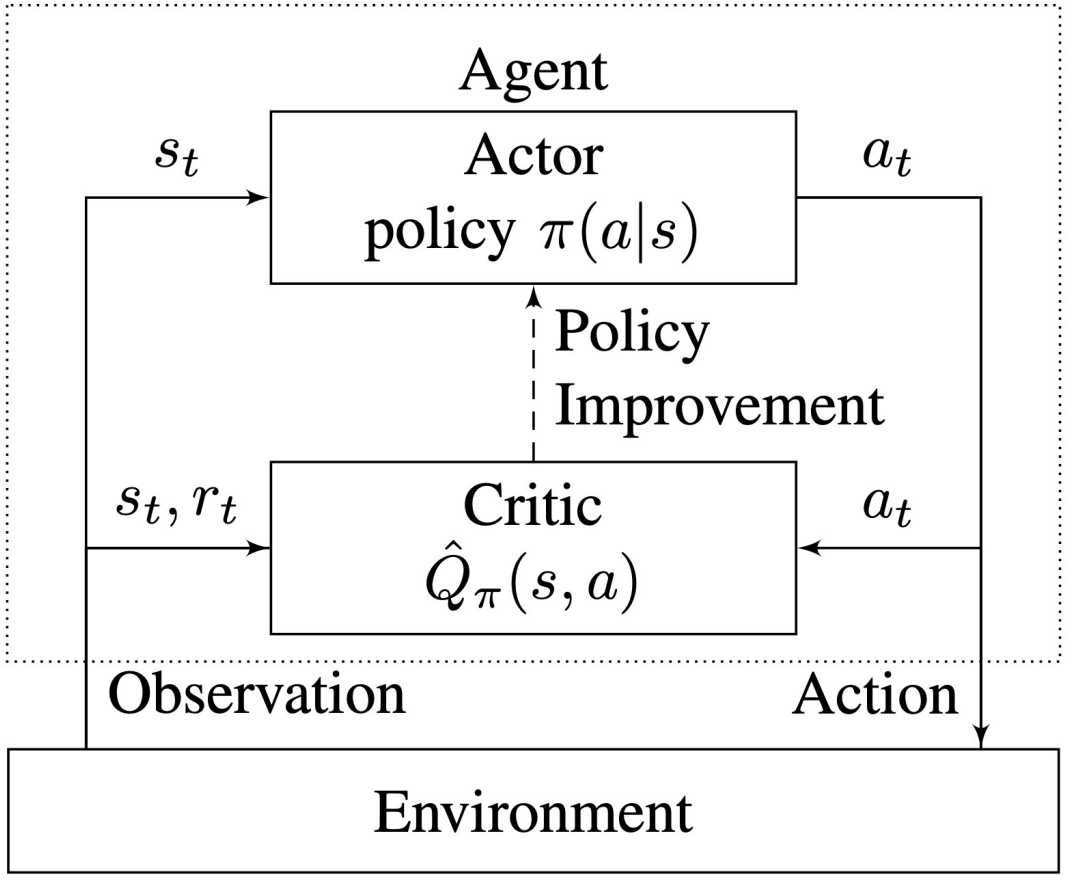
- $B_w = \delta_{t:t+1} = G_{t:t+1} - V(s,w)$

$$G_{t:t+1} = r + \gamma V(s_{i+1}, w)$$

=> reinforce with baseline

- $B_w = \delta_{t:t+1}$ *I

=> n-step actor critic (A2C)



Source: [Kämmerer, 2019] cites [Kimura & Kobayashi ,1998]

Algorithm of A2C

1: **Input** is a policy $\pi(a|s,\boldsymbol{\vartheta})$ and state value function V(s,w)

2: Initialize ϑ_i and w

3: loop for each episode

4: Initialize environment to provide s and I=1

5: do

6: Choose a in state s using π breaking ties randomly

a7: Take action a and observe r_i, s_{i+1}

b7: $\delta_{t:t+n} = G_{t:t+n} - V(s,w)$

c7: $w = w + \alpha_w \delta_{t:t+n} \nabla_w V(s,w))$

b8: Update ϑ using baseline B_w

c8: I= γI

9: s= s_{i+1}

10: while s is not terminal

Q-learning is changed & extended by marked lines

Learns both policy and ist state-value function on ever single step

- Actor updates the policy in the direction suggested by the critic and takes best action based on that policy in b8
- Baseline prediction in b7 critizes the consequences of the action and is used to update the policy (*critic*)
- Critic alters the probability of actions to be taken by δ

=> off-policy algorithm

Advantage Function $\mathcal{A}$

- Given a predicted state-value function which is the expectation (average over all actions of a state, one can represent the indivodual actions relative to the average

$$\mathcal{A}_\pi = Q_\pi(s,a) - V_\pi(s,a)$$

- If advantage function $\mathcal{A}_\pi$ is positive means that action taken by agent is more optimal
- If $\mathcal{A}_\pi$ is negative, decrease the probability of the action to be taken next time

- For policy-based algorithms using the temporal difference error of Eq. 1

$$\mathcal{A}_\pi = r_{i+1} + \gamma Q_\pi(s_{i+1}, a_{i+1}) - V_\pi(s,a)$$

- When agent is following optimal policy, proof in [Bhatnagar et al., 2007]

$$= r_{i+1} + \gamma V_\pi(s_{i+1}, a_{i+1}) - V_\pi(s,a)$$

Advantage Actor Critic (A2C)

- Objective function J is based on the expected rewards and model parameters ϑ

$$\mathrm{J}(\vartheta) = E_\pi[G|s,a]$$

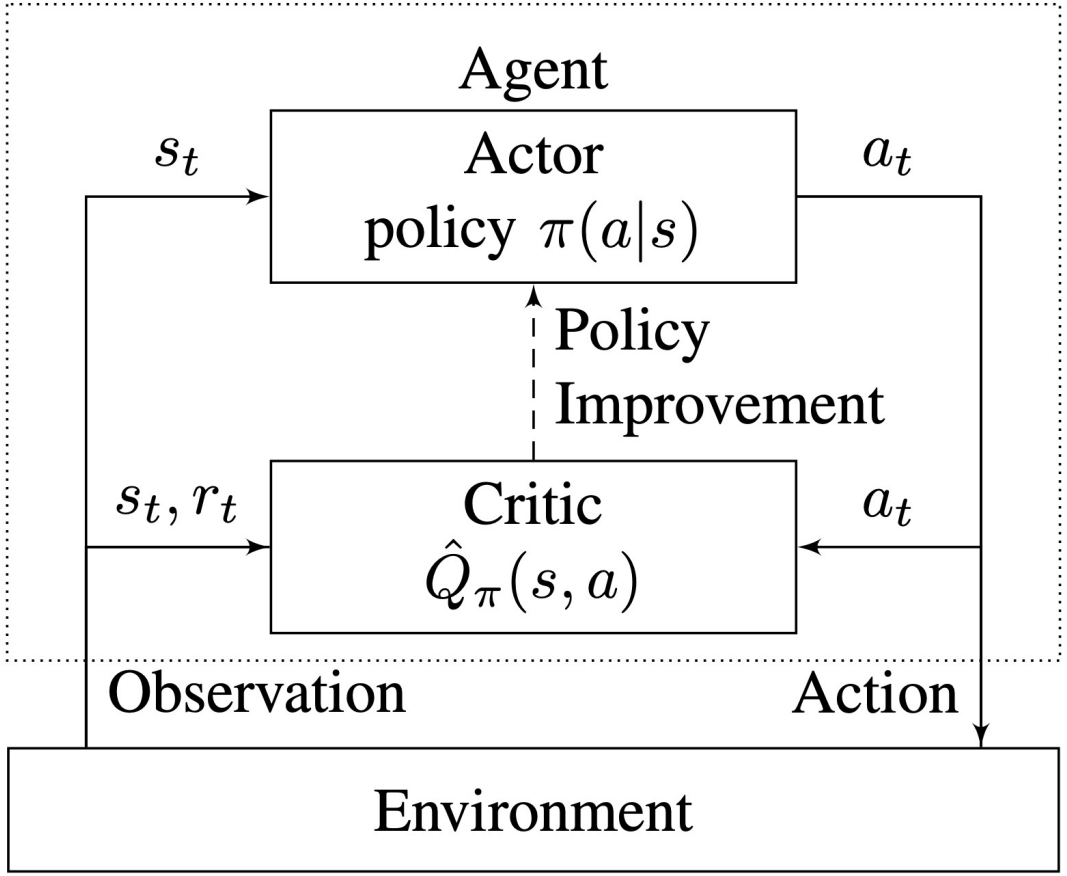
Parameters are optimized with update rule

$$\vartheta = \vartheta + \alpha\left(\nabla_\vartheta \mathrm{J}(\vartheta)\right)$$

With [Sutton&Barto, 1998 p.325]

$$\nabla_\vartheta \mathrm{J}(\vartheta) = \nabla_\vartheta E_\pi[G|s,a] = \nabla_\vartheta \sum_{a\in A} Q(s,a)\pi(a|s,\vartheta)$$

$$= \cdots = E_\pi[G\nabla_\vartheta ln(\pi(a|s,\vartheta)|s,a]$$



Source: [Kämmerer, 2019] cites [Kimura & Kobayashi 1998]

Soft Q-Learning [Ziebart et al., 2008]

- Using Shannon entropy $H\big(\pi_\vartheta(s)\big) = -\sum_{a\in A} \pi_\vartheta(s,a) \log_b \pi_\vartheta(s,a)$ the maximum entropy objective function is

$$\pi * = \underset{\pi}{\operatorname{argmax}} \sum_{s\epsilon S, a\epsilon A} E[r + \beta H(\pi(a|s))]$$

- Large values of hyperparameter β promote nondeterministic actions and hence more exploration
- Beware: in deterministic problems with one optimal trajectory this may not work well
- BUT: Learning a stochastic policy forces the agent to learn **many** optimal solutions to the same problem

$$\mathrm{J}(\vartheta) = E_\pi[r + \beta H]$$

$$\pi(s|a)) \sim e^{Q_{soft}(s,a,)/\beta}$$

$$Q_{soft}(s,a) = E\big[r + \gamma E[V_{soft}(s_{i+1})])\big]$$

$$V_{soft}(s) = E[Q(s,a) + \beta \log(\pi(a|s))]$$

- β=0 is the classifical Q-Learning
- Main drawback is that several actions have to be sampled in the next state in order to estimate its current soft V-value, what makes it hard to implement in practice

Soft Actor-Critic (SAC) algorithm [Haarnajoa et al.,2017]

- Objective Function for critic

$$J_Q(\vartheta) = E_\pi\left[\left(\frac{1}{2}Q_{soft}(s,a) - \left(r + \gamma E\left[V_{soft}(s_{i+1})\right]\right)\right)^2\right]$$

- Objective Function for actor

$$J_V(\varphi) = E_\pi\left[\beta \log\left(\pi_\varphi(a|s)\right) - Q_{soft,\vartheta}(s,a)\right]$$

- Policy parameters csn be learned by directly minimizing the expected KL-divergence
- Given, *Z* as a partition function to normalize the softmax

$$J(\sigma)_\pi = \mathbb{E}_{s_t \in \mathcal{D}}[D_{\mathrm{KL}}(\pi_\theta(s,\cdot)|\frac{\exp Q_\psi(s_t,\cdot)}{Z(s_t)})]$$

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors $\psi, \bar{\psi}, \theta, \phi$.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$
  end for
end for

Soft Actor-Critic (SAC)

- SAC claims to Outperforms PPO [Haarnajoa et al.,2017]

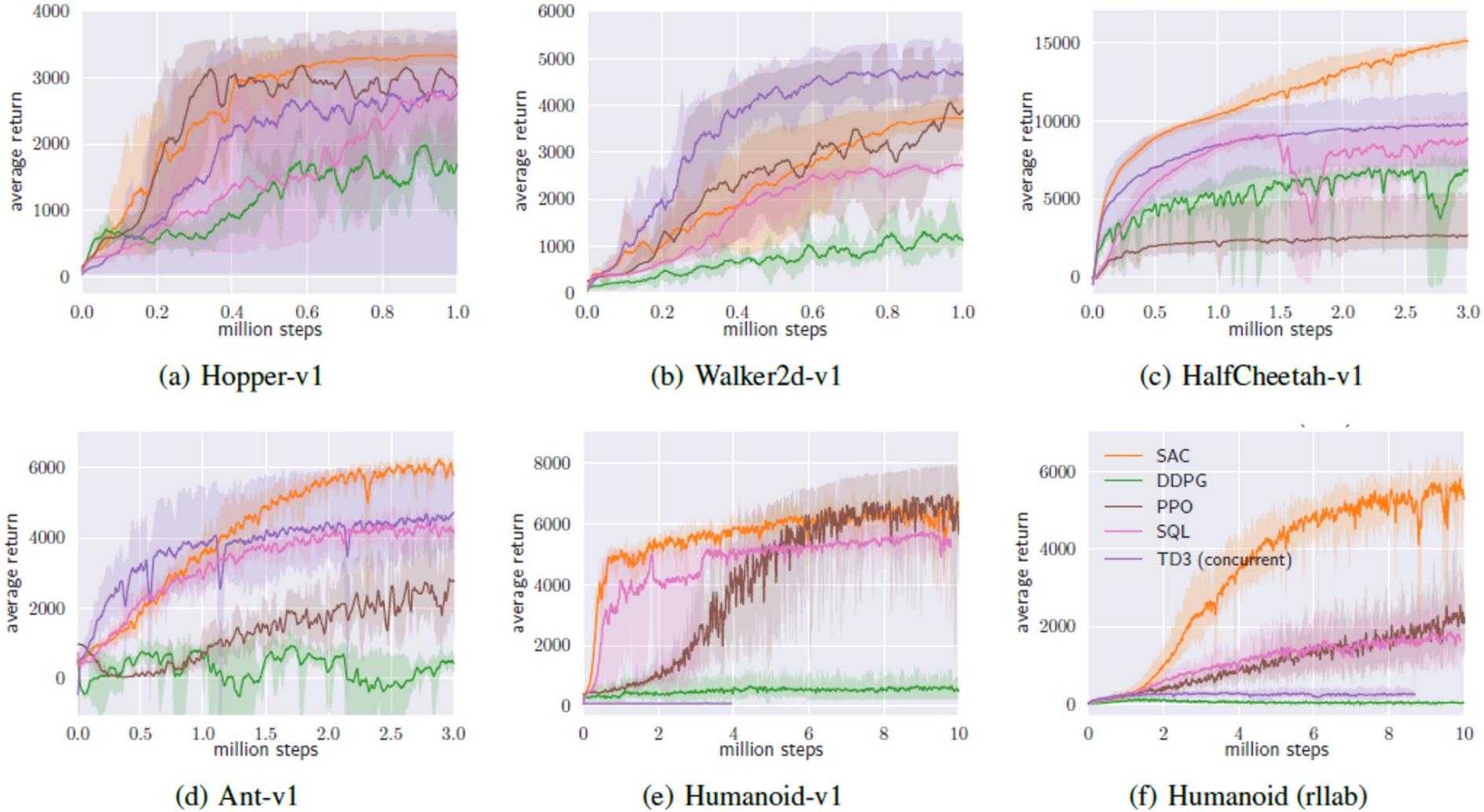


Figure 1. Training curves on continuous control benchmarks. Soft actor-critic (yellow) performs consistently across all tasks and outperforming both on-policy and off-policy methods in the most challenging tasks.

Outlook policy-based algorithms: Proximal Policy Optimization (PPO)

- Published in [Schulman et al., 2015, PMLR] and extended [Schulman et al. 2017 ArXiv]:
- Objective function *J* is based on policy and advantage function

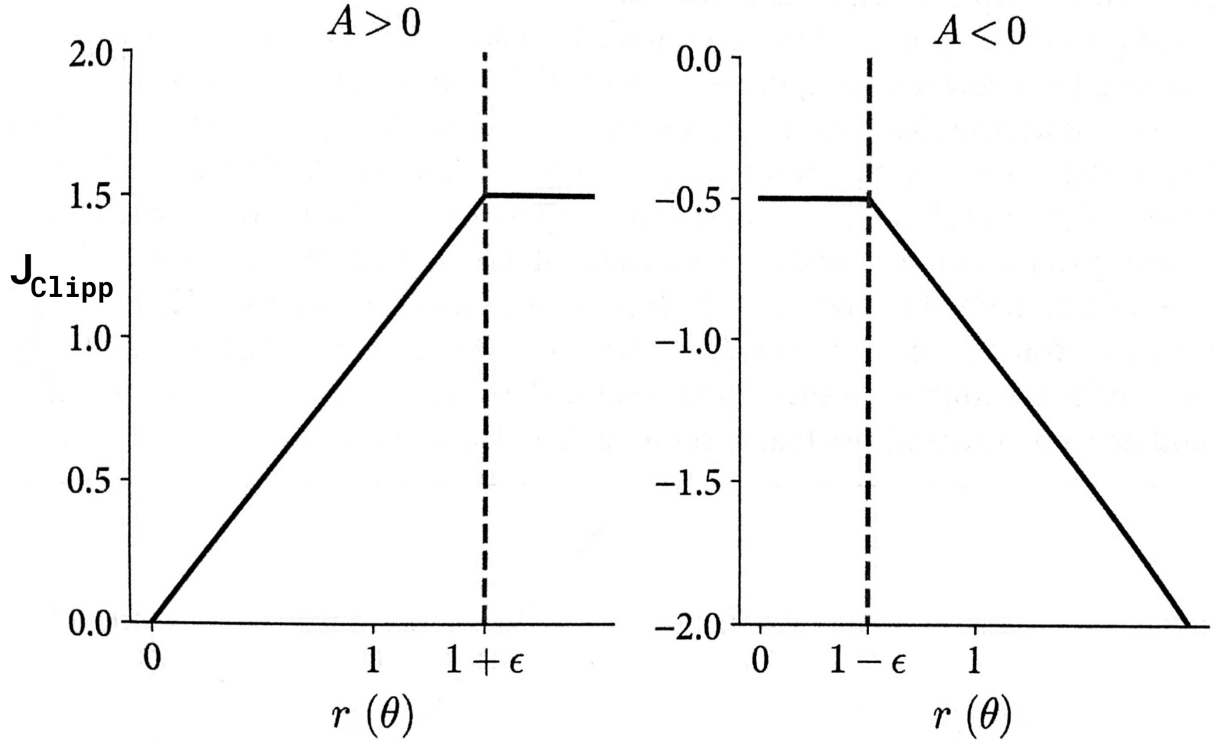
$$\mathrm{J}(\vartheta) = E_{\pi}\left[r((\vartheta)A_{\pi,old}(s,a)\right] \text{ with } \mathrm{r}(\vartheta) = \frac{\pi_{\vartheta}(a|s)}{\pi_{\vartheta,old}(a|s)}$$

- Without a limitation on difference between ϑ,old and ϑ, maximization of $J(\vartheta)$ would lead to instability with extremely large parameter updates and big policy ratios

$$\Rightarrow J_{clipp}(\theta) = \mathbb{E}[\min(r(\theta)A_{\pi,old}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)A_{\pi,old}(s,a))]$$

Outlook policy-based algorithms II: Proximal Policy Optimization (PPO)

$$J_{clipp}(\theta) = \mathbb{E}[\min(r(\theta)A_{\pi,old}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)A_{\pi,old}(s,a))]$$



- Function *clip* clips the ratio to be no more than 1+ϵ and no less than 1−ϵ.
- Objective function of PPO takes the minimum one between original value and clipped version

=> loses motivation for increasing the policy update to extremes for better rewards

PPO Objective function and algorithm

$$J_{clipp+VF+S}(\theta) = E\left[J_{clipp} + C_1 L_{VF} + C_2 H(\pi_\vartheta, s)\right]$$

With $C_1 L_{VF}$=$\left(V_\vartheta(s) - V_{target}\right)^2$, and H the entropy defined by [Mnih et al., 2016]

- H improves the exploration by discouraging premature convergence to suboptimal deterministic policies
- C_1, C_2 hyperparameters
- V_{target} is the prediction target of a neural netowork

Source: [Schulman et al. 2017 ArXiv]:

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

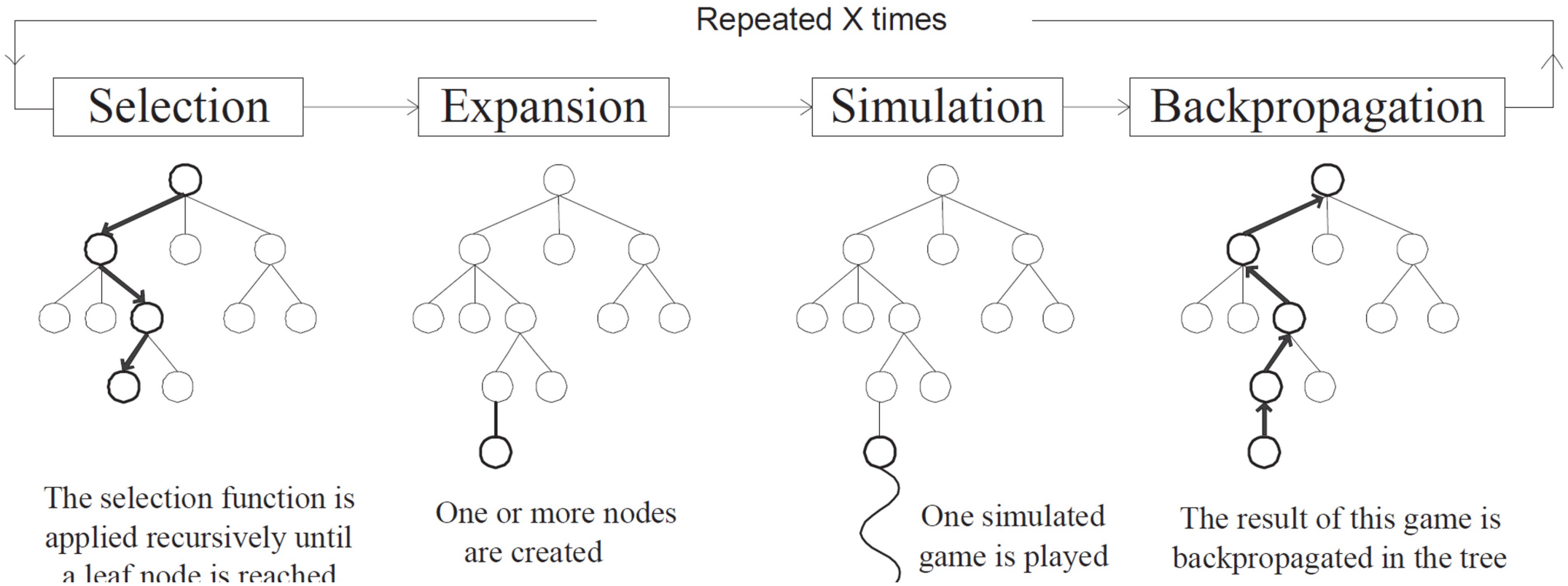
Application of AlphaGO zero has three key components [Silver at al., 2017]

1. Self-play generates training data in each time step i (s_i, π_i, z_i),
   - z_i beeing the game winner from perspective of current step
   - Two player play and are continually evalauted by policy evaluation
   - (s_i, π_i, z_i) is taken form best-performing player at the moment
2. Monte-Carlo-Search-Tree (MCTS) output probabilties $\pi_i = a_{\vartheta_{i-1}}(s_i)$ if playing each move by each player given neural network parameters ϑ and root position s_i
   - Tree search is restricted to either 19*19*2=722 moves or loss of game
3. Training data is used in supervised learning to train parameters ϑ from data sample uniformly
   - Minimizing the error between predicted value v and the self-play winner z
   - Maximize the similarity between the neural network move probabilities p to the search probabilities π_i

   $(p, v) = f(\vartheta)$ and $loss = (z-v)^2 - \pi_T \log(p + c\|\vartheta\|^2)$, c controlling parameter to prevent overfitting

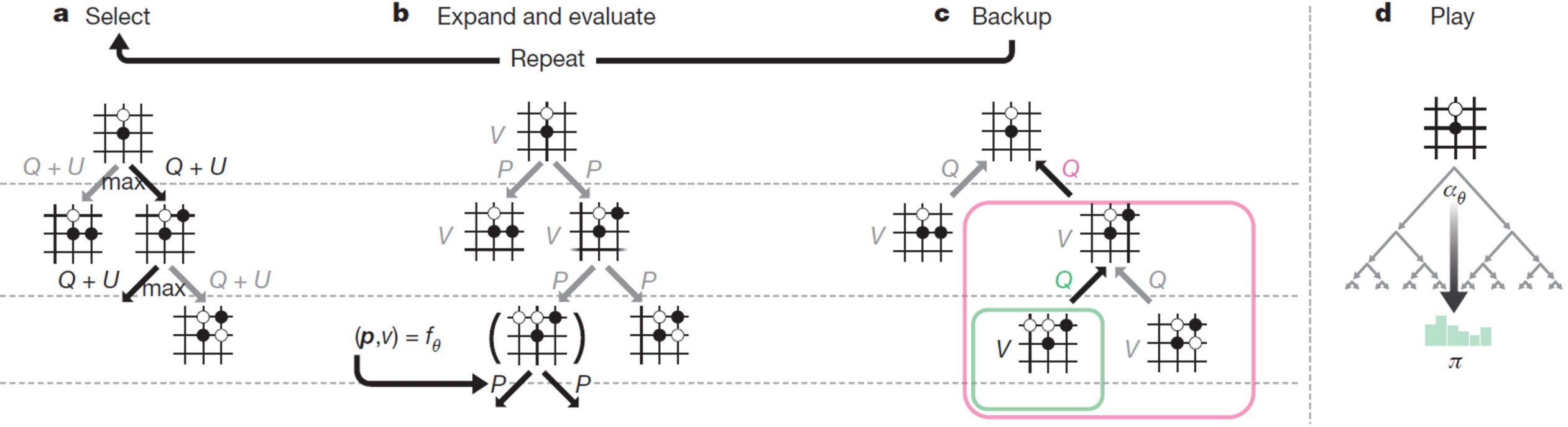
Monte-Carlo-Search-Tree (MCTS)

- MCTS successively focuses multiple simulations at the current state by extending the intial portions of trajectories that have received high evaluations from earlier simulations
- MCTS executes as many iterations as possible before an action needs to be selected
- MCTS incrementaly builds a tree whose root node represents the current state
- Each iteration consists of the four operations: Selection, Expansion, Simulation and Backup/Backpropagation [Chaslot et al., 2008]



Monte-Carlo-Search-Tree (MCTS) in adapted [Silver at al., 2017]

- Each simulation traverses tree by selecting edge with maximum action value Q, plus an upper confidence bound U that depends on stored prior probability P & visit count N for that edge which is incremented once traversed
- Leaf node is expanded and the associated position s is evaluated by neural network $(P(s, \cdot), V(s)) = f\_\vartheta(s)$
  - Vector of P values are stored in the outgoing edges from s
- Action value Q is updated to track mean of all evaluations V in the subtree below that action
- Once the search is complete, search probabilities $\boldsymbol{\pi}$ are returned, proportional to $N1/\tau$, where N is the visit count of each move from the root state and τ is a parameter controlling temperature



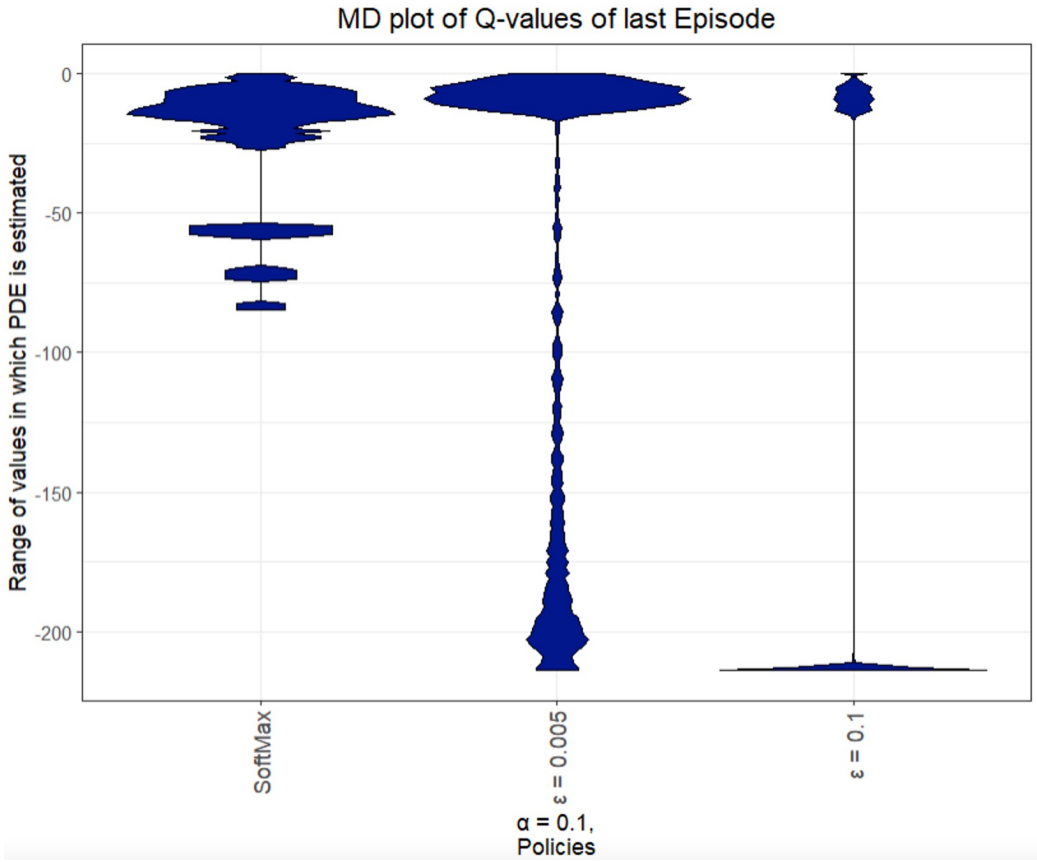
AlphaGO Zero Results

- Using 64 GPU workers and 19 CPU servers AphaGOZero learns for 40 days 500.000 games
- Components calculate in parallel using specific computation structures of Google cloud build for performaning efficiently on these algorithms
- Thereafter it has enough experience in GO to beat prior AlphaGO(s) that were combined with human experience
  - Prior AlphaGO outperformed grandmaster in GO in 2016 [ilver et al., 2016].

Analysis of Update Rule

$$Q_{t:t+1}(s_i, a)$$
$$= Q(s_i, a) + \alpha \left(r + \gamma * \underset{a_s \in A}{arg\max}\, Q(s_{i+1}) - Q(s_i, a) \right)$$
$$= (1 - \alpha)\, Q(s_i, a) + \alpha \left(r + \gamma * \underset{a_s \in A}{arg\max}\, Q(s_{i+1}, a_s) \right)$$

Q values of the 1000 Episode show multimodalities
=> Assumption is not correct



Is the exponential moving average (e.g., see [Awheda/Schwartz,2013])

$$\bar{x}_{n+1} = (1 - \alpha)\bar{x}_n + \alpha x_{n-1}$$

$$Q_*(s) = \underset{a_i \in A}{arg\max}\, E_\pi[G|s]$$

=> Computation of $Q_*(s)$ assumes approximation by exponential moving average!

Improvements of Q-Learning

- In basic algorithm the next best action is chosen

=> Intial bad update can create loop in which agent keeps making supoptimal decision

  - Solution: Use two look-up tables [Hasselt, 2010]

- Expected returns can be noisy
  - Solution: Buffer the rewards and update action value function after agent has visited the state a certain number of times [Strehl at al.,2006]
- Combine on- and off-policy Q learning by intruducing new parameter σ [Sutton & Barto, 2018]

Extensions Q Learning II: eligibility trace

- Use tracer to update Q table backwards [Singh & Sutton, 1996]

After line 7:

- Temporarily calculate one step error $\delta_{t:t+1}$
- Iterate $z(s,a)$= $z(s,a)$+1

Change Line 8 to:

- Add eligibility trace:

For each $s \in S$ and for each $a \in S$

$$Q(s_i,a) = Q(s_i,a) + \alpha\left(G_{t:t+n} - Q(s_i,a)\right) * \text{z(s,a)}$$

$$z(s,a) = \lambda\, z(s,a), 0 < \lambda < 1$$

Difference between Sampling Updates and Expected Update

In determinisct environment there is no difference

- Sampling Update is used in Q learning with $Q_{i:i+1}(s_i, a) = Q(s_i, a) + \alpha\left(r + \gamma * arg\max_{a_s \in A} Q(s_{i+1}, a_s) - Q(s_i, a)\right)$
- Expected Updates is used in TD(0) with $Q_{i:i+1}(s_i, a) =$

$$\sum_{s_{i+1}, r} P(s_{i+1}, r|s.a)\left(r + \gamma * arg\max_{a_s \in A} Q(s_{i+1}, a_s) - Q(s_i, a)\right)$$

- BUT: In stochastic environment given s,a many next states are possible making sampling update comutationally cheaper but influenced by the sampling error
- In general there are 8 backup strategy of updating Q and V possible
  1. Update V or Q
  2. Update on or off-policy
  3. Expected updates or sample updates

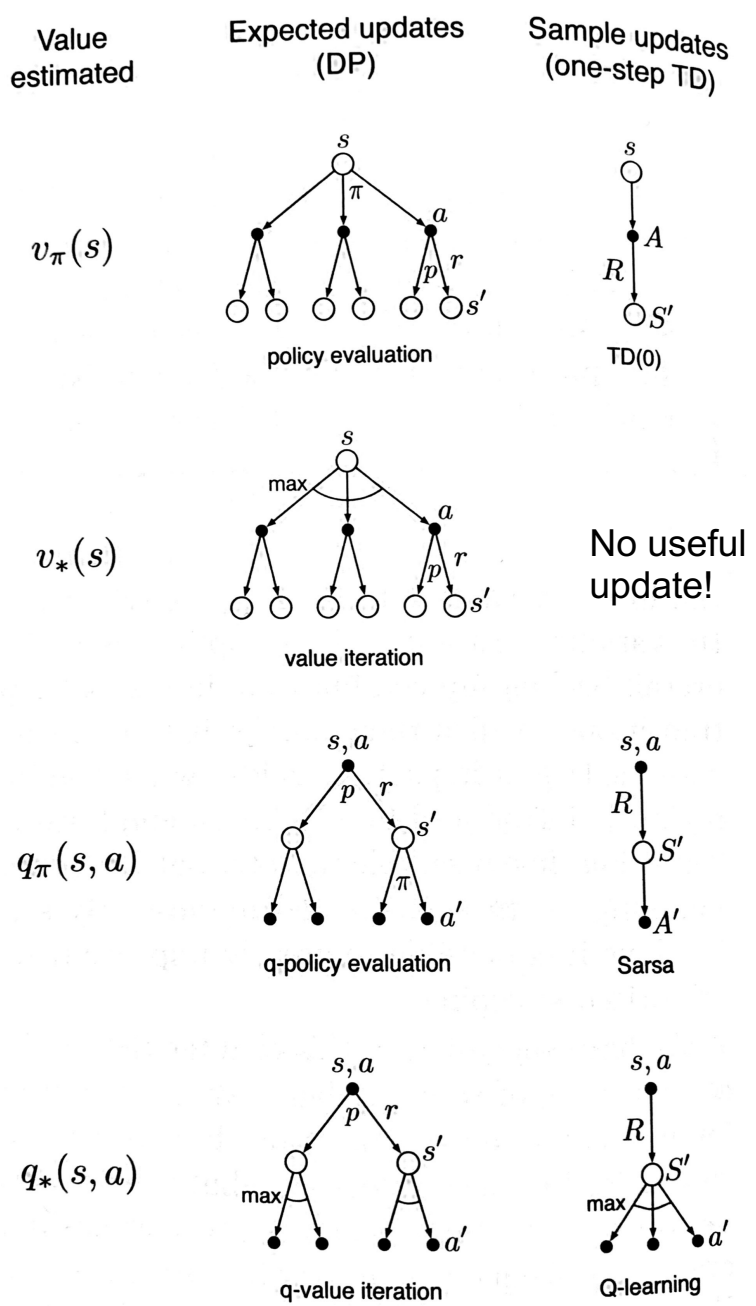


Figure 8.6: Backup diagrams for all the one-step updates considered in this book.

Source: [Sutton & Barto, 1998]

Bellman Equation

$Let\, s = \mathrm{s_t}$ be the current state, G the discounted return then the Bellman equation defines a recursive relationship between values with

$$V_\pi(s) = E_\pi[G_t|s] = E_\pi\left[\sum_{i=0}^{T} \gamma^i r_i \,\middle|\, s\right]$$

$$= E_\pi\left[\sum_{i=0}^{T} \gamma^i r_{t+i} \,\middle|\, s\right] = E_\pi\left[r_t + \sum_{i=1}^{T} \gamma^i r_{t+i} \,\middle|\, s\right]$$

$$= E_\pi[r_t + \gamma(1 * r_{t+1} + \gamma r_{t+2} + \cdots)|s]$$

$$= E_\pi\left[r_t + \gamma \sum_{i=0}^{T} \gamma^i r_{t+1+i} \,\middle|\, s\right]$$

$$= E_\pi[r_t + \gamma G_{t+1}|s] = \ldots = r_t + \gamma * E_\pi[G_{t+1}|s]$$

$$= r_t + \gamma * V_\pi(s_{t+1})$$

Bellman Equation

$Let\, s = \mathrm{s_t}$ be the current state, G the discounted return then the Bellman equation defines a recursive relationship between values with

$$V_\pi(s) = E_\pi[G_t|s] = r_t + \gamma * V_\pi(s_{t+1})$$

$\Rightarrow$ We have an optimal substructure and overlapping suproblems that can be solved recursively

$\Rightarrow$ Dynamic programming can be used in generalized policiy iteration (GPI)

- $\pi_o \xrightarrow{Evaluation} V_{\pi_0} \xrightarrow{Iteration} \pi_1 \xrightarrow{Evaluation} V_{\pi_1} \xrightarrow{Iteration} \pi_2 \ldots . \pi *$

=> model-based RF algorithms if transition probabilities T given