Machine Learning 10-601

Tom M. Mitchell
Machine Learning Department
Carnegie Mellon University

Nov 13, 2017

Today:

- Learning of control policies
- Markov Decision Processes
- Temporal difference learning
- Q learning

Readings:

- Mitchell, chapter 13
- Kaelbling, et al., *Reinforcement Learning: A Survey*

Thanks to Aarti Singh for several slides

Reinforcement Learning: AlphaGo

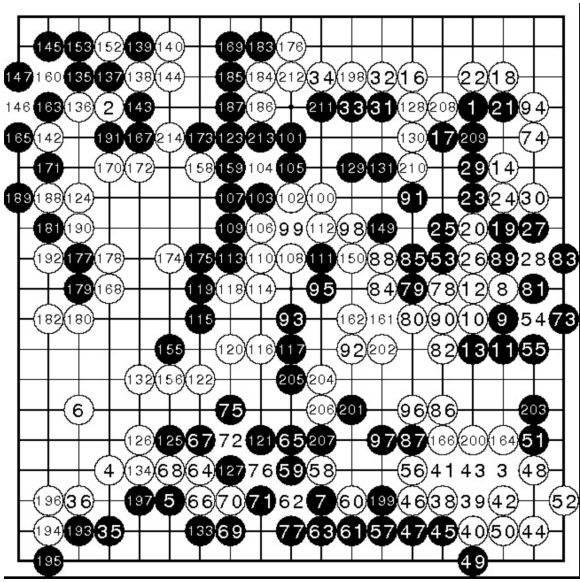
[Deep Mind, 2016]

Learning task:

- chose move at arbitrary board states

Training:

- initially supervised: trained to mimic 30 million expert book moves
- then reinforcement learning: played many games against itself



Algorithm:

- reinforcement learning + neural network
- Oct 2015 version used: 1202 CPUs, 176 GPUs.

Result:

- World-class Go player

Reinforcement Learning: AlphaGo

[Deep Mind, 2017]

ARTICLE

doi:10.1038/nature24270

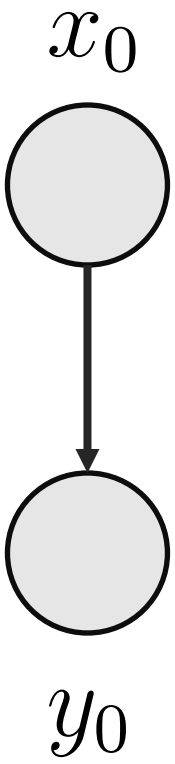
Mastering the game of Go without human knowledge

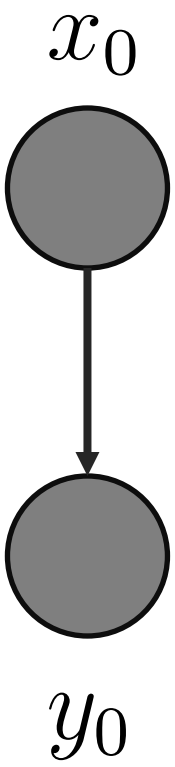
David Silver[1]*, Julian Schrittwieser[1]*, Karen Simonyan[1]*, Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

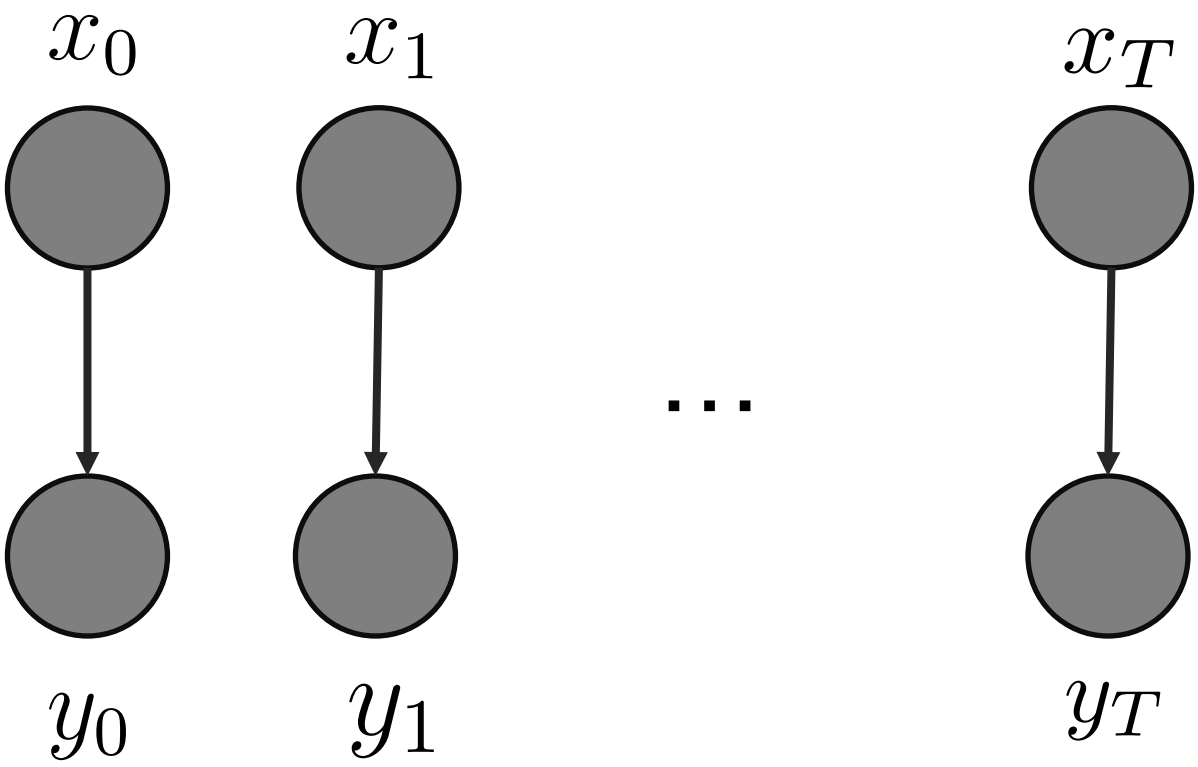
A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here we introduce an algorithm based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game rules. AlphaGo becomes its own teacher: a neural network is trained to predict AlphaGo's own move selections and also the winner of AlphaGo's games. This neural network improves the strength of the tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program AlphaGo Zero achieved superhuman performance, winning 100–0 against the previously published, champion-defeating AlphaGo.

Outline

- Learning control strategies
  - Credit assignment and delayed reward
  - Discounted rewards
- Markov Decision Processes
  - Solving a known MDP
- Online learning of control strategies
  - When next-state function is known: value function $V^*(s)$
  - When next-state function unknown: learning $Q^*(s,a)$
- Role in modeling reward learning in animals

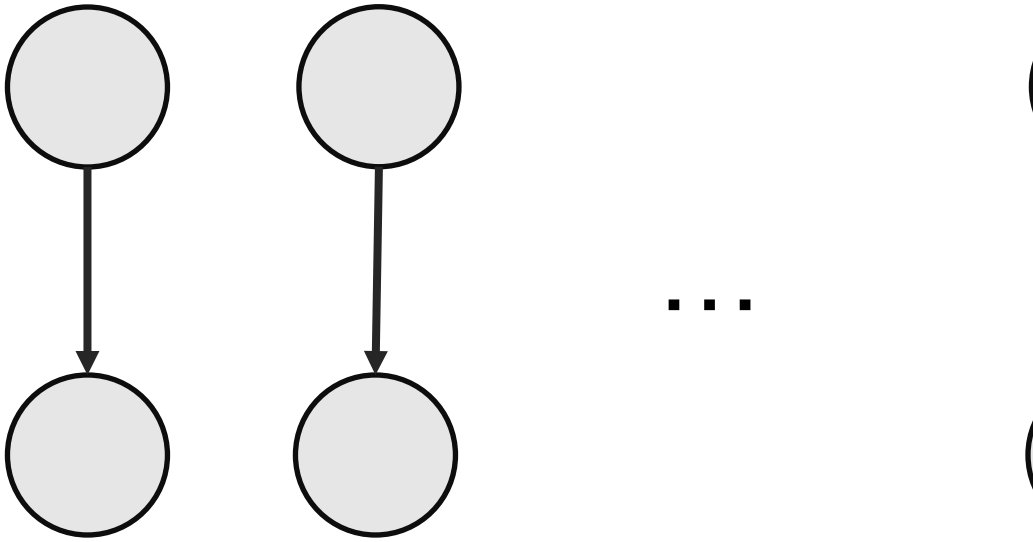

x_0
y_0


x_0
y_0


x_0
x_1
x_T
y_0
y_1
y_T



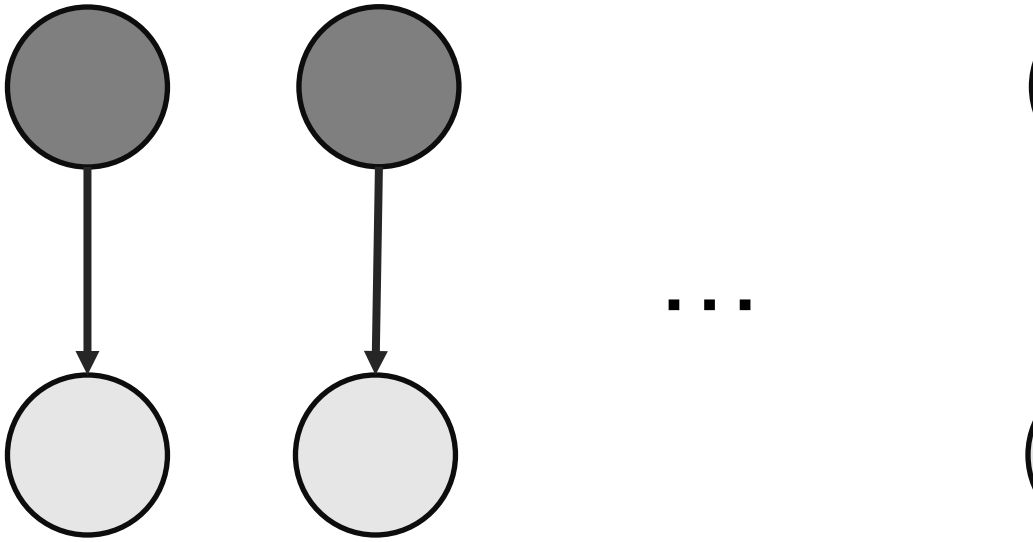
x_0 x_1 x_T



y_0 y_1 y_T

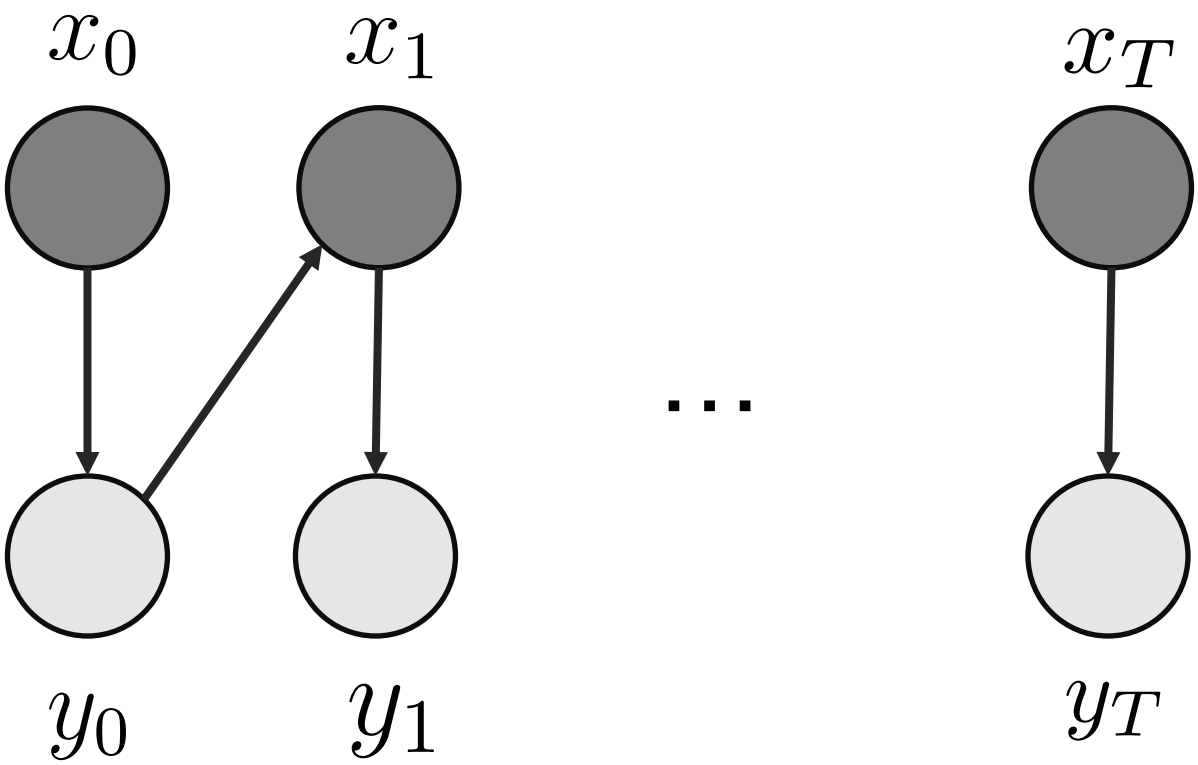


x_0 x_1 x_T

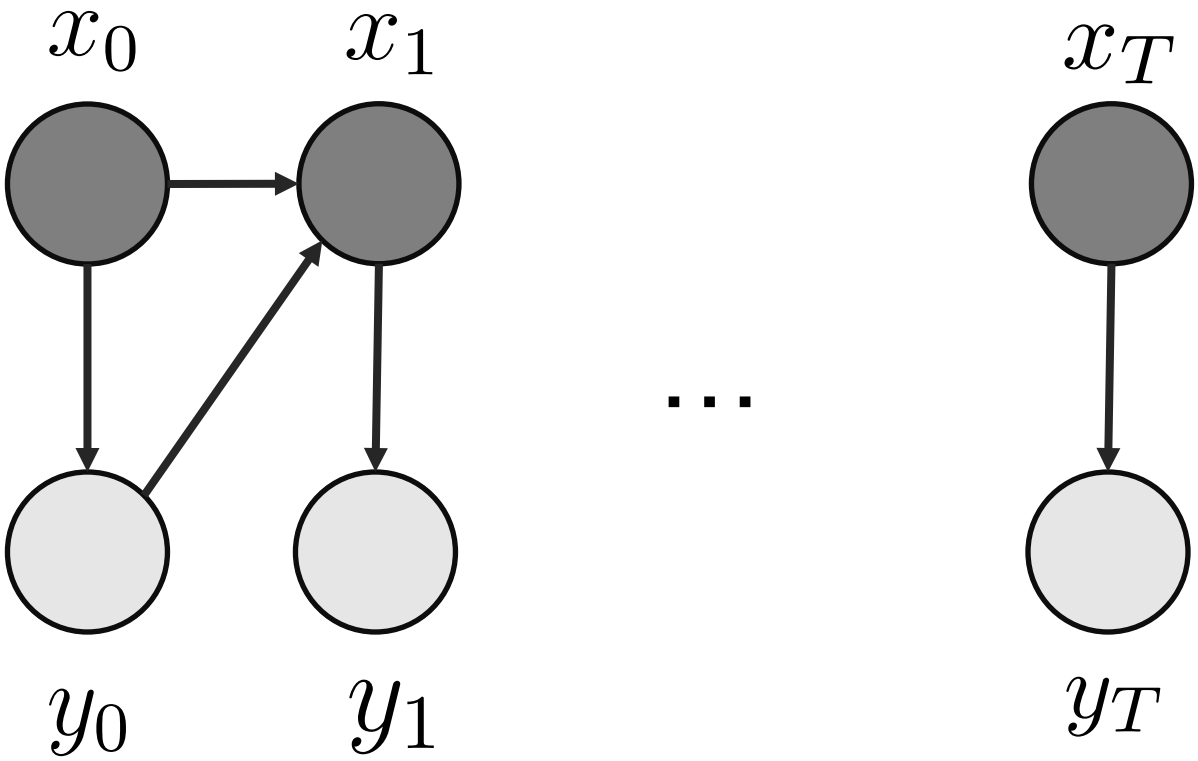


y_0 y_1 y_T

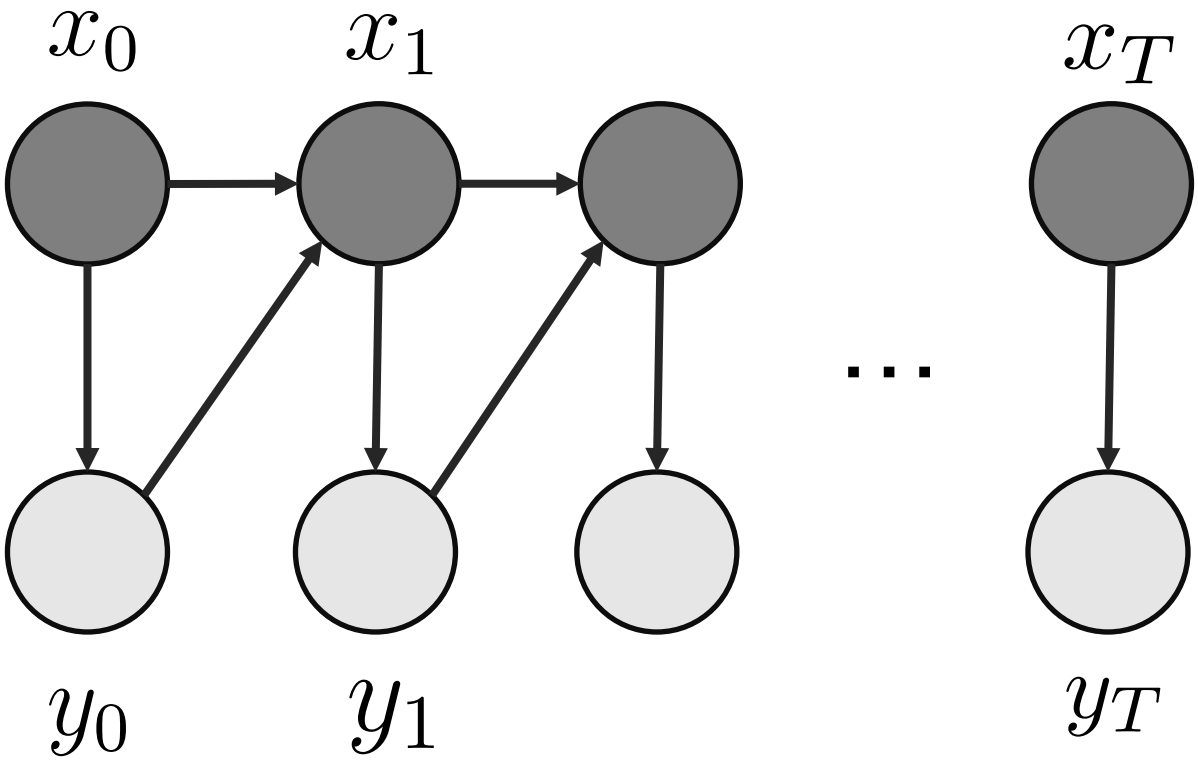


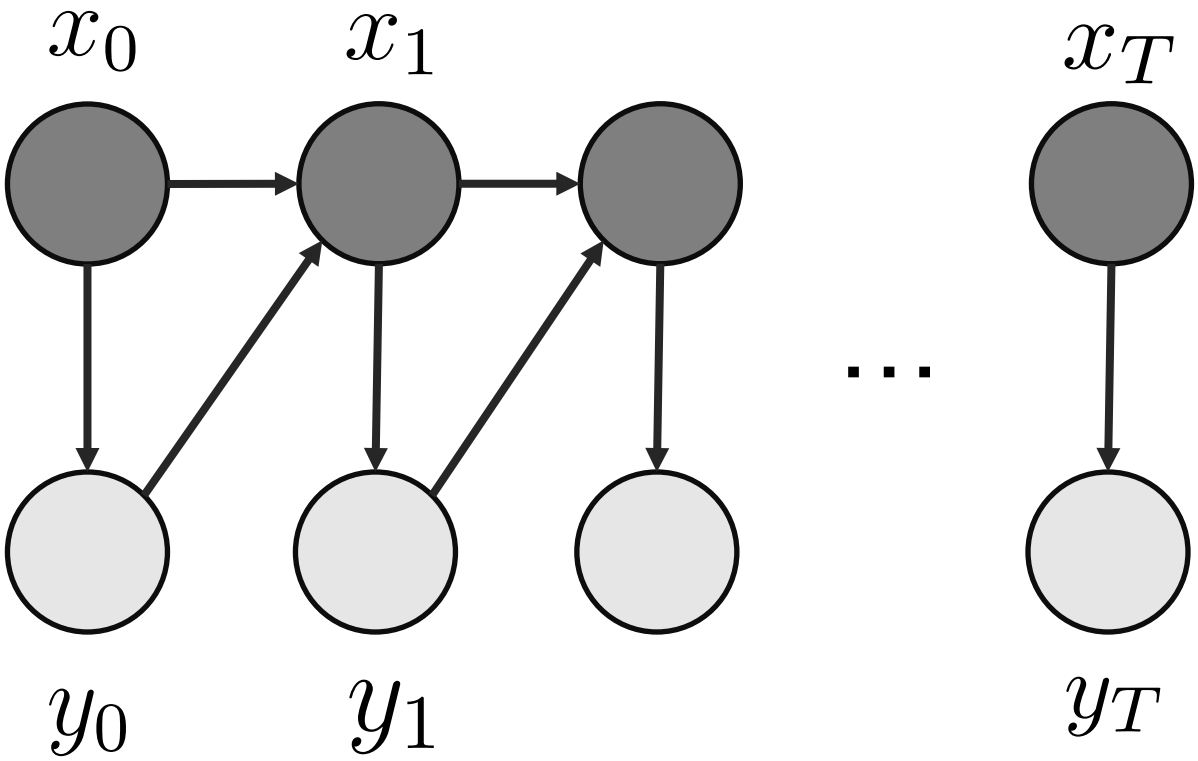

x_0
x_1
x_T
y_0
y_1
y_T




x_0
x_1
x_T
y_0
y_1
y_T

...

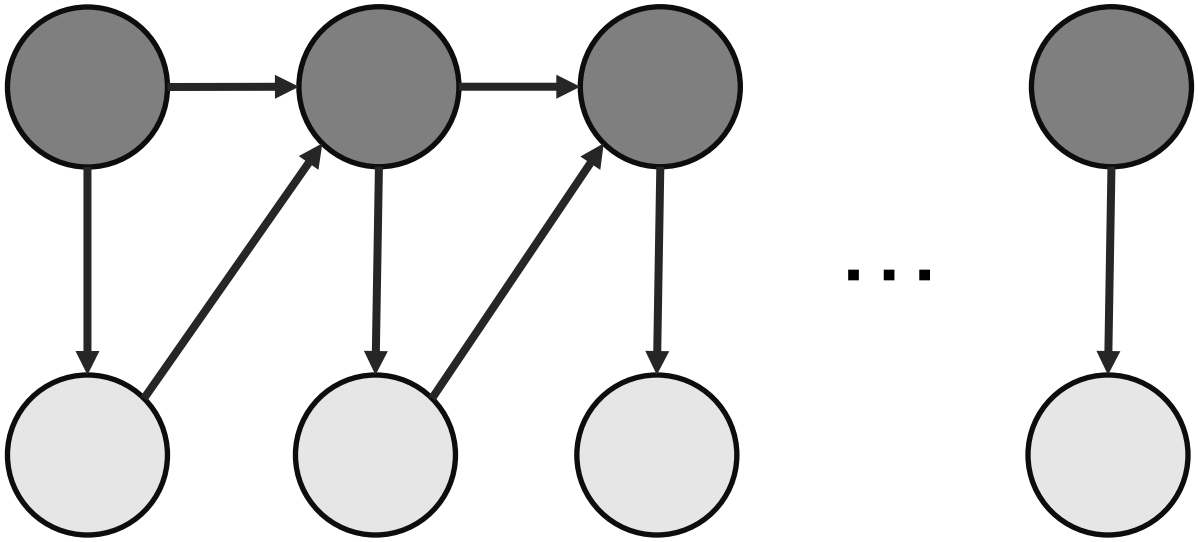

x_0
x_1
x_T
y_0
y_1
y_T


x_0
x_1
x_T
y_0
y_1
y_T



x_0 x_1 x_T

states

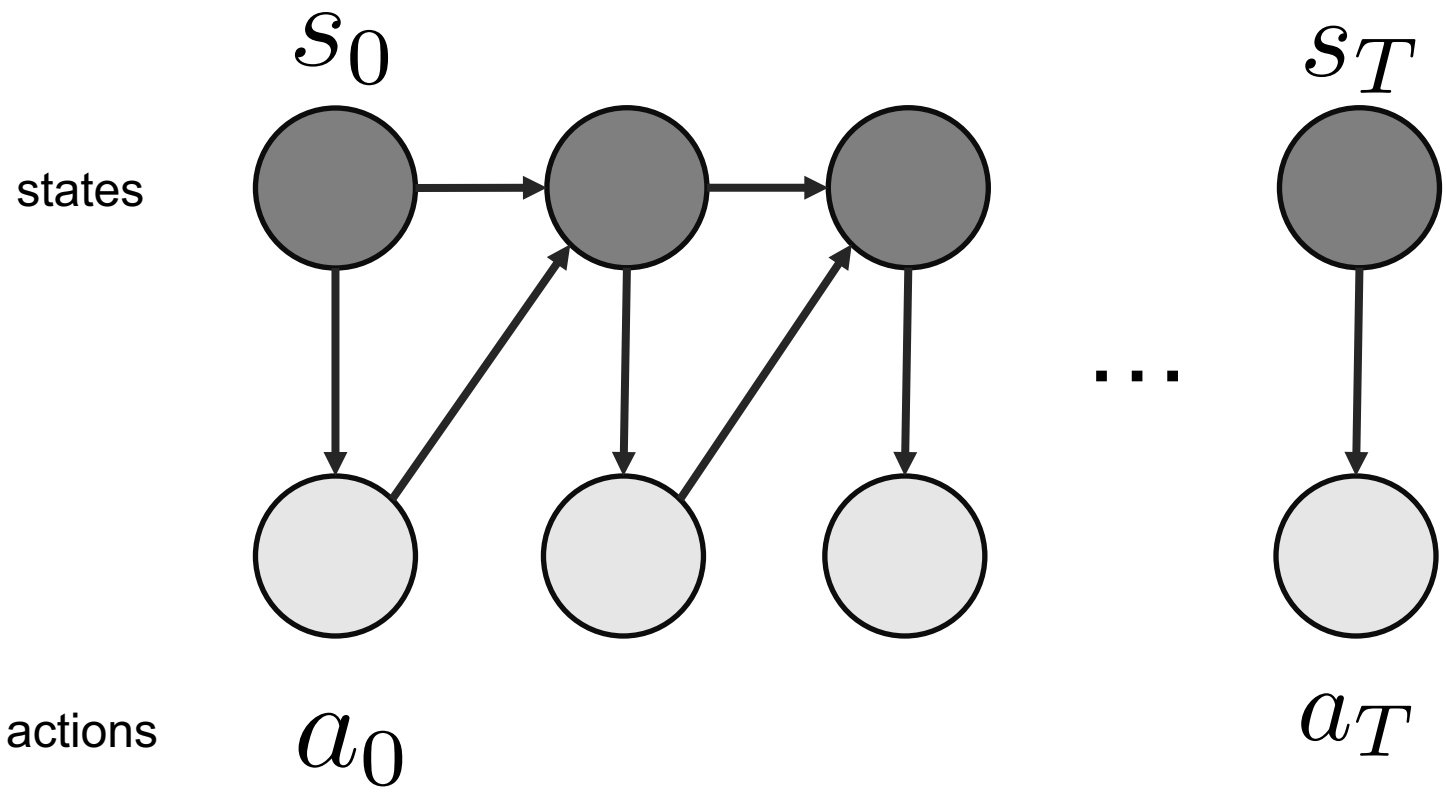


...

actions

y_0 y_1 y_T




s_0
s_T
states
...
actions
a_0
a_T



s_0 s_T

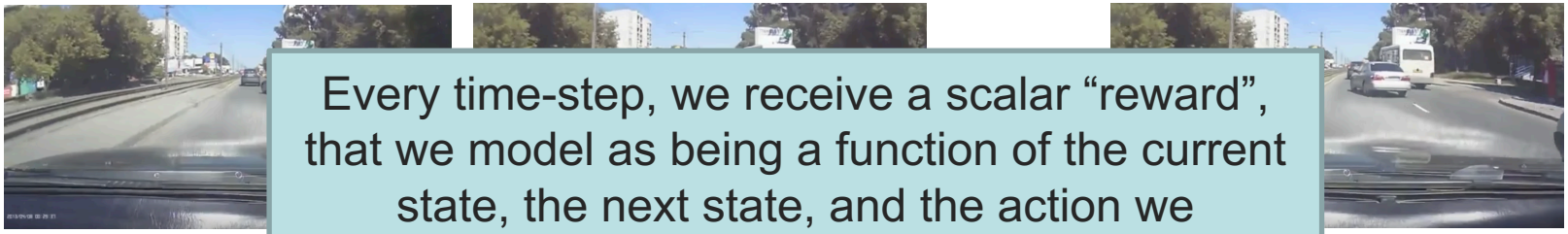
states

$\cdots$

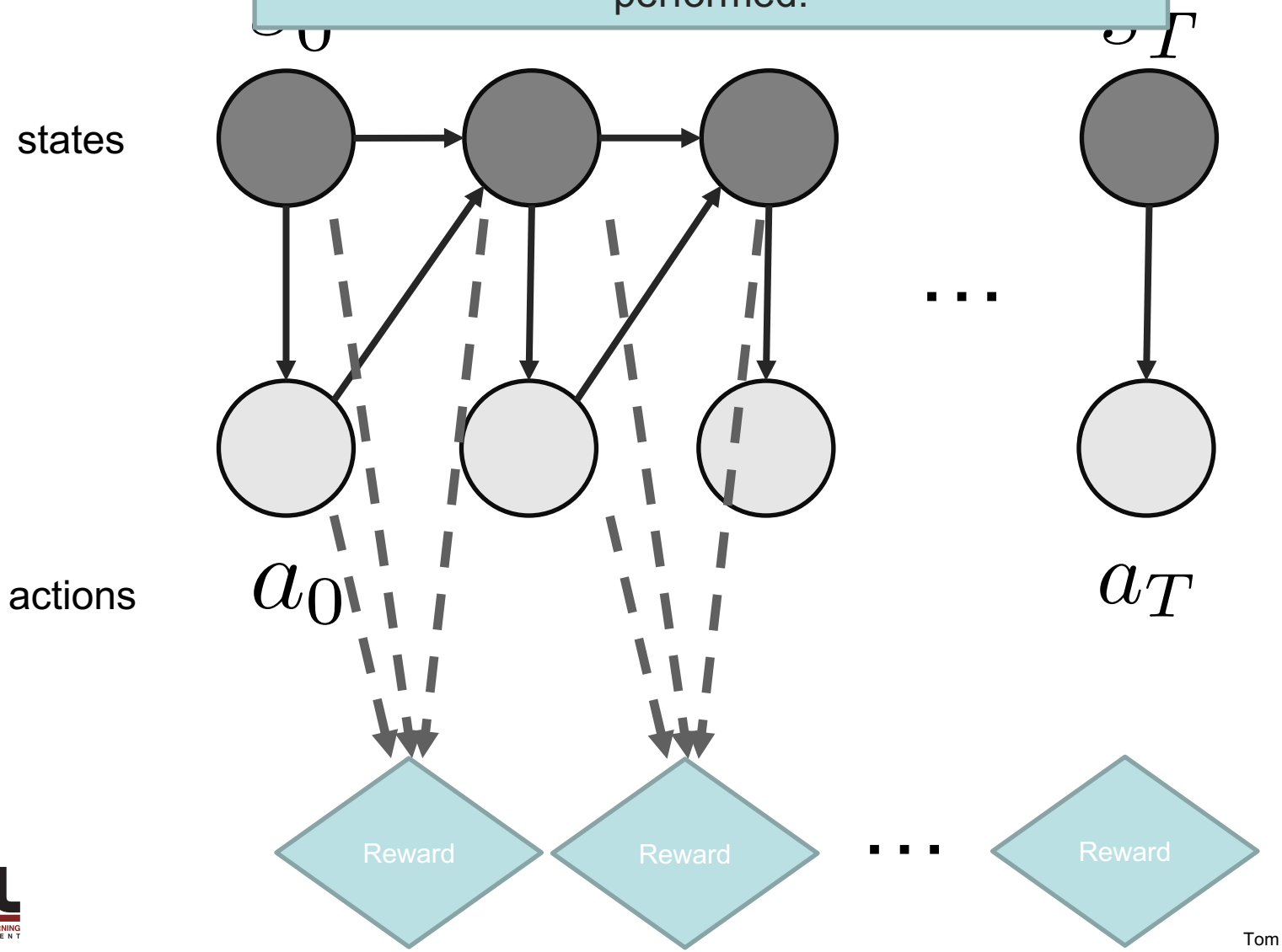
actions

a_0 a_T

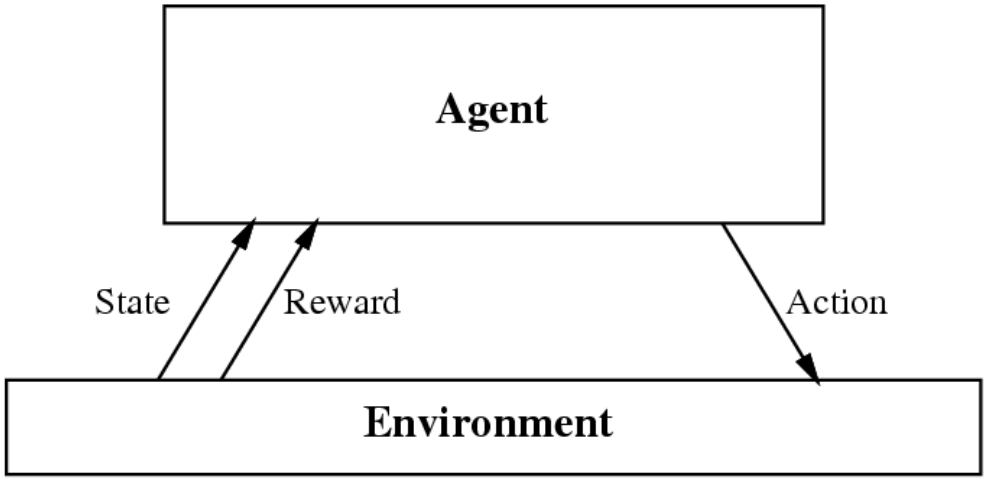
Reward Reward $\cdots$ Reward



Every time-step, we receive a scalar “reward”, that we model as being a function of the current state, the next state, and the action we performed.



Reinforcement Learning Problem

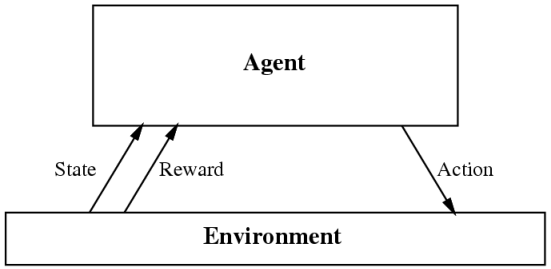


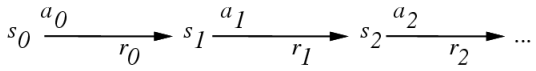
$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$$

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots \text{ , where } 0 \leq \gamma < 1$$

Markov Decision Process = Reinforcement Learning Setting





- Set of states S
- Set of actions A
- At each time, agent observes state $s_t \in S$, then chooses action $a_t \in A$
- Then receives reward r_t , and state changes to s_{t+1}
- Markov assumption: $P(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, ...) = P(s_{t+1} \mid s_t, a_t)$
- Also assume reward Markov: $P(r_t \mid s_t, a_t, s_{t-1}, a_{t-1}, ...) = P(r_t \mid s_t, a_t)$

- The task: learn a policy π: S → A for choosing actions that maximizes

$$E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...] \quad 0 < \gamma \leq 1$$

for every possible starting state s_0

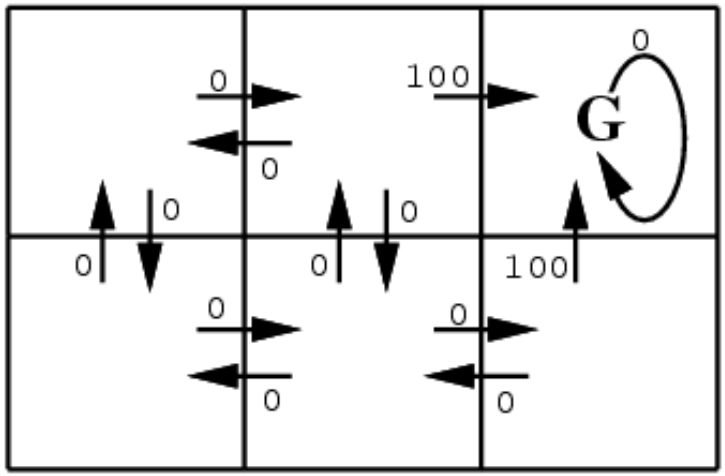
Discount future rewards

Reinforcement Learning Task for Autonomous Agent

Execute actions in environment, observe results, and

- Learn control policy π: S→A that maximizes $\sum_{t=0}^{\infty} \gamma^t E[r_t]$ from every state s $\in$ S

Example: Robot grid world, deterministic reward $r(s,a)$



$r(s,a)$ (immediate reward)

Reinforcement Learning Task for Autonomous Agent

Execute actions in environment, observe results, and

- Learn control policy π: S→A that maximizes $\sum_{t=0}^{\infty} \gamma^t E[r_t]$ from every state s $\in$ S

Yikes!!

- Function to be learned is π: S→A
- But training examples are not of the form <s, a>
- They are instead of the form < <s,a>, r >

Reinforcement Learning Task for Autonomous Agent

Suppose that we run only a finite number of steps forward, from *t=0* to *t=T-1*. The **maximum total reward** starting from s_0 at *t=0* and going all the way to *T-1* can then be written as:

$$TR_{0\ldots T-1}(s_0) = \max_{a_0} \left\{ \sum_{s_1 \in S} P(s_1 \mid s_0, a_0)[R(s_1, s_0, a_0) + TR_{1\ldots T-1}(s_1) \right\}$$

* Suppose that future rewards are not discounted (since we are going a finite amount of steps in the future, the total reward will be finite

$TR_{0\ldots T-1}(s_0)$ stands for maximum total reward from time *t=0* to time *t=T-1* When we start at some initial state s_0

Reinforcement Learning Task for Autonomous Agent

Suppose that we run only a finite number of steps forward, from *t=0* to *t=T-1*. The total reward, starting in state s_0 is then:

Which action should we choose at step 0 to maximize total reward until the end (*T* steps)?

Probability of moving to next state

$$TR_{0\ldots T-1}(s_0) = \max_{a_0} \left\{ \sum_{s_1 \in S} P(s_1 \mid s_0, a_0)[R(s_1, s_0, a_0) + TR_{1\ldots T-1}(s_1) \right\}$$

Max over actions in *t=0*

Immediate reward from that transition

Total maximum reward from next step to the end *T*

Reinforcement Learning Task for Autonomous Agent

In general, we can express the maximum total expected reward by following the best sequence of actions from any step *t* to *T-1* (the end)

$$TR_{t\ldots T-1}(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + TR_{t+1\ldots T-1}(s_{t+1}) \right\}$$

Max over actions in time step *t*

Probability of moving to next state

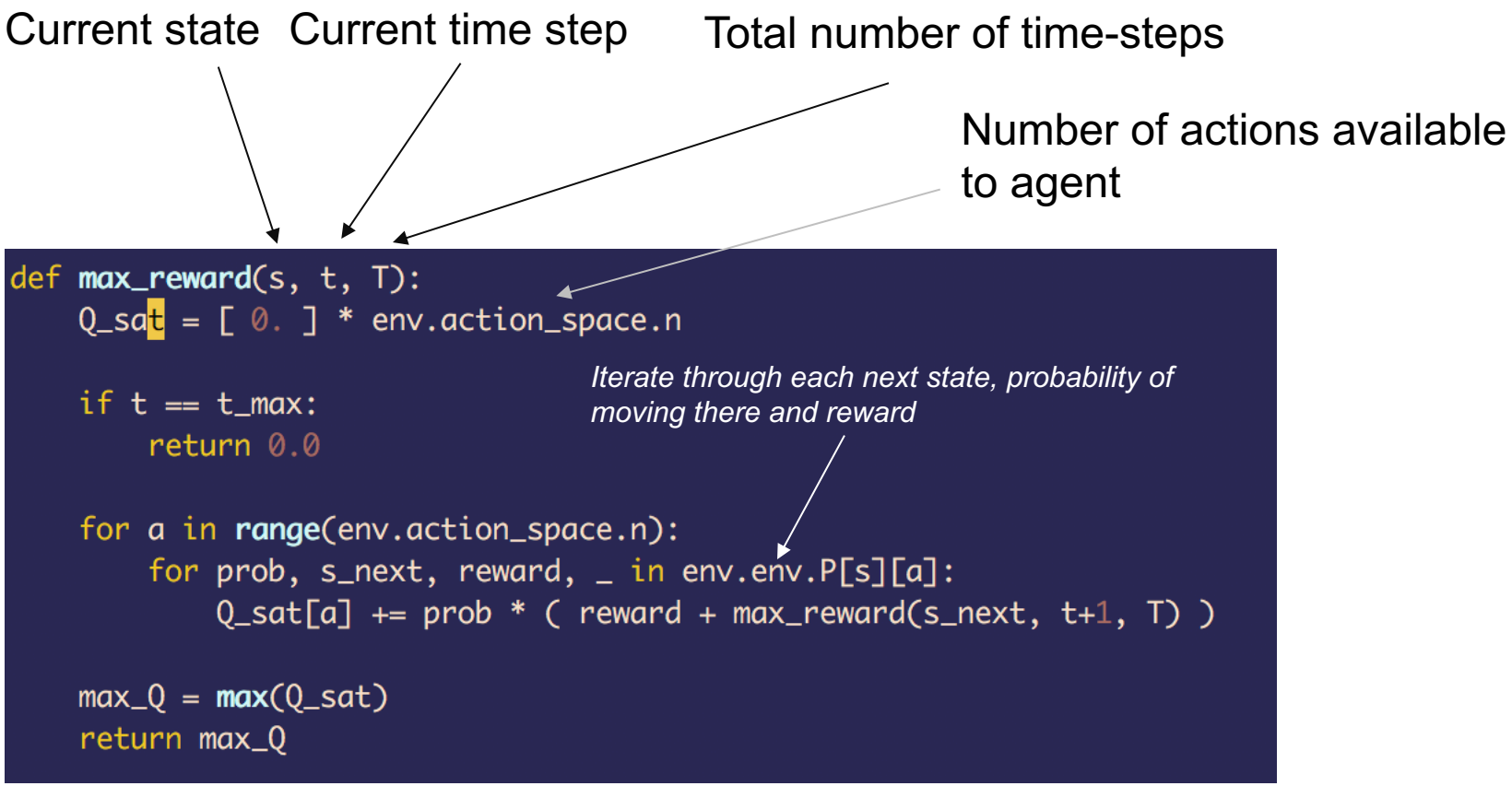
Immediate reward from that transition

Total maximum reward from next step to the end

Notice that the maximum total reward is actually the **expected maximum total reward** since the transitions from state to state are stochastic and we are computing an expectation over possible state transitions

Reinforcement Learning Task for Autonomous Agent

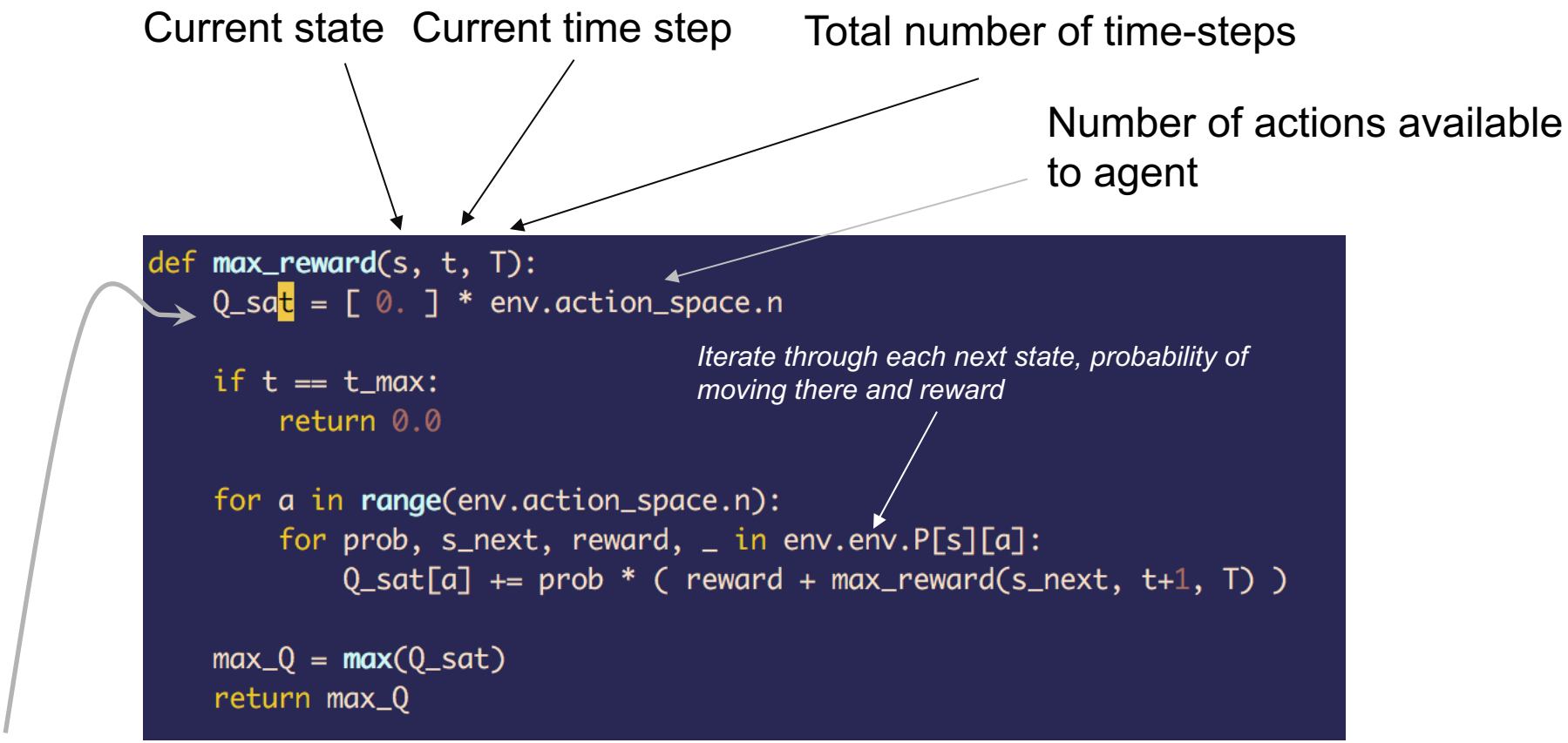
Naïve implementation




```
def max_reward(s, t, T):
    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    return max_Q
```

Reinforcement Learning Task for Autonomous Agent

Naïve implementation




```
def max_reward(s, t, T):
    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    return max_Q
```

Expected reward from now to to the end of taking action *a* in this time step

Reinforcement Learning Task for Autonomous Agent

What's the problem with this implementation?

```
def max_reward(s, t, T):
    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    return max_Q
```

Iterate through each next state, probability of moving there and reward

Reinforcement Learning Task for Autonomous Agent

What's the problem with this implementation?

Exponential run-time with *T!*

```
def max_reward(s, t, T):
    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    return max_Q
```

Iterate through each next state, probability of moving there and reward

Reinforcement Learning Task for Autonomous Agent

What's the problem with this implementation?

TR function (*max_reward*) gets called multiple times with exact same parameters!

```
def max_reward(s, t, T):
    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    return max_Q
```

Iterate through each next state, probability of moving there and reward

Reinforcement Learning Task for Autonomous Agent

What's the problem with this implementation?

Cache results!

```
cache = {}
def max_reward(s, t, T):
    if (s,t) in cache:
        return cache[s,t]

    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    cache[s,t] = max_Q
    return max_Q
```

Return result if computed before

Save results into a table

Reinforcement Learning Task for Autonomous Agent

What's the problem with this implementation?

Cache results! **Dynamic Programming**.

```
cache = {}
def max_reward(s, t, T):
    if (s,t) in cache:
        return cache[s,t]

    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    cache[s,t] = max_Q
    return max_Q
```

Return result if computed before

Save results into a table

Reinforcement Learning Task for Autonomous Agent

What's the problem with this implementation?

Cache results! **Dynamic Programming**.

What is the runtime?

```
cache = {}
def max_reward(s, t, T):
    if (s,t) in cache:
        return cache[s,t]

    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    cache[s,t] = max_Q
    return max_Q
```

Return result if computed before

Save results into a table

Reinforcement Learning Task for Autonomous Agent

What's the problem with this implementation?

Cache results! **Dynamic Programming**.

What is the runtime?

```
cache = {}
def max_reward(s, t, T):
    if (s,t) in cache:
        return cache[s,t]

    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    cache[s,t] = max_Q
    return max_Q
```

Return result if computed before

Save results into a table

Reinforcement Learning Task for Autonomous Agent

What's the problem with this implementation?

Cache results! **Dynamic Programming**.

How to recover optimal action sequence?

```
cache = {}
def max_reward(s, t, T):
    if (s,t) in cache:
        return cache[s,t]

    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    cache[s,t] = max_Q
    return max_Q
```

Return result if computed before

Save results into a table

Reinforcement Learning Task for Autonomous Agent

A little about our simulation world.

FrozenLake world in OpenAI Gym environment

```
env = gym.make('FrozenLake-v0')
env.reset()
```

(S)tarting position

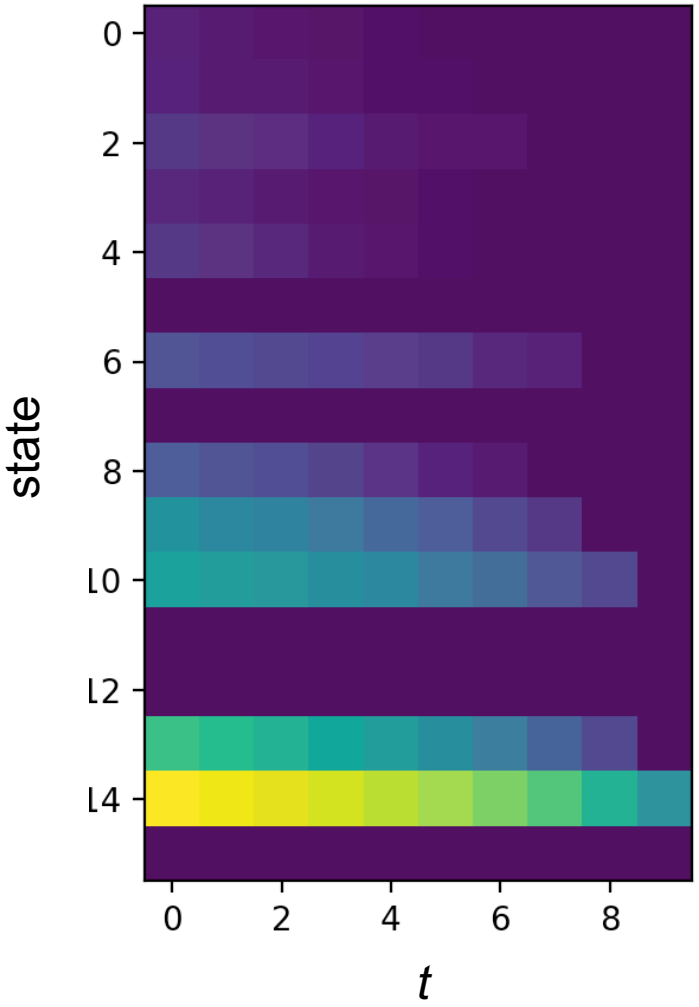
```
SFFF
FHFH
FFFH
HFFG
```

(H)ole
(get stuck in the hole)

(G)oal position
(receive reward only if you get here)

Reinforcement Learning Task for Autonomous Agent

Let's visualize **Maximum Expected Total Reward** as a function of time *t* (position in the sequence) and state

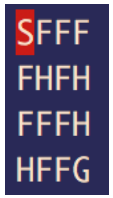


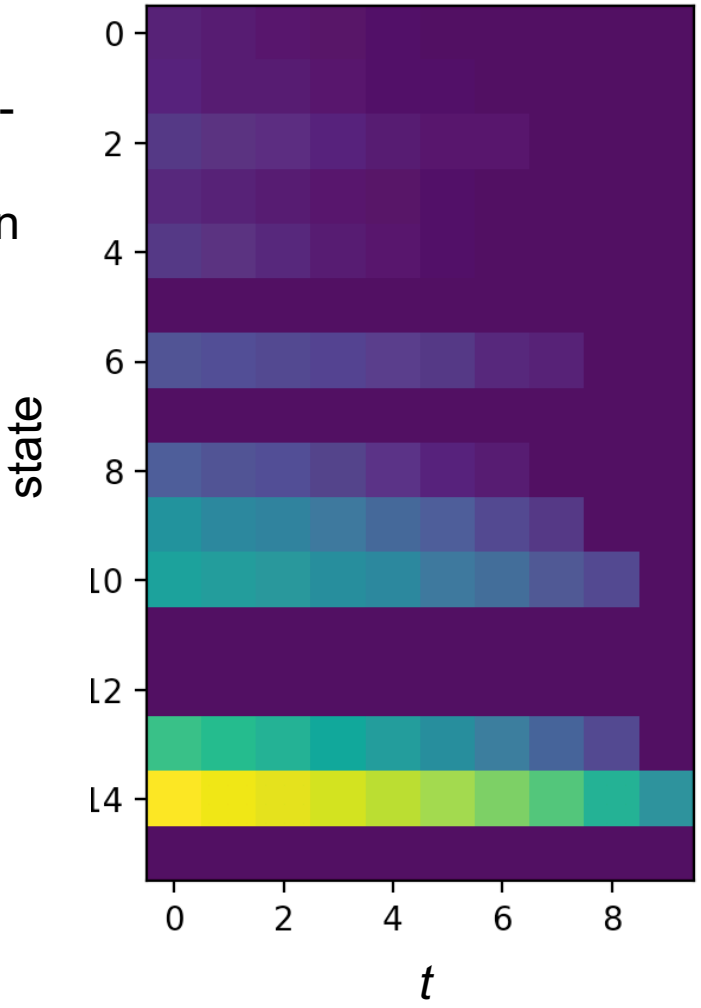
Maximum Total Expected Reward
From *t* to the end (*T-1)*
where we set *T = 10*

Reinforcement Learning Task for Autonomous Agent

Let's visualize **Maximum Expected Total Reward** as a function of time *t* (position in the sequence) and state

Each state on the y-axis represents a cell in the simulation (there are 16 cells)



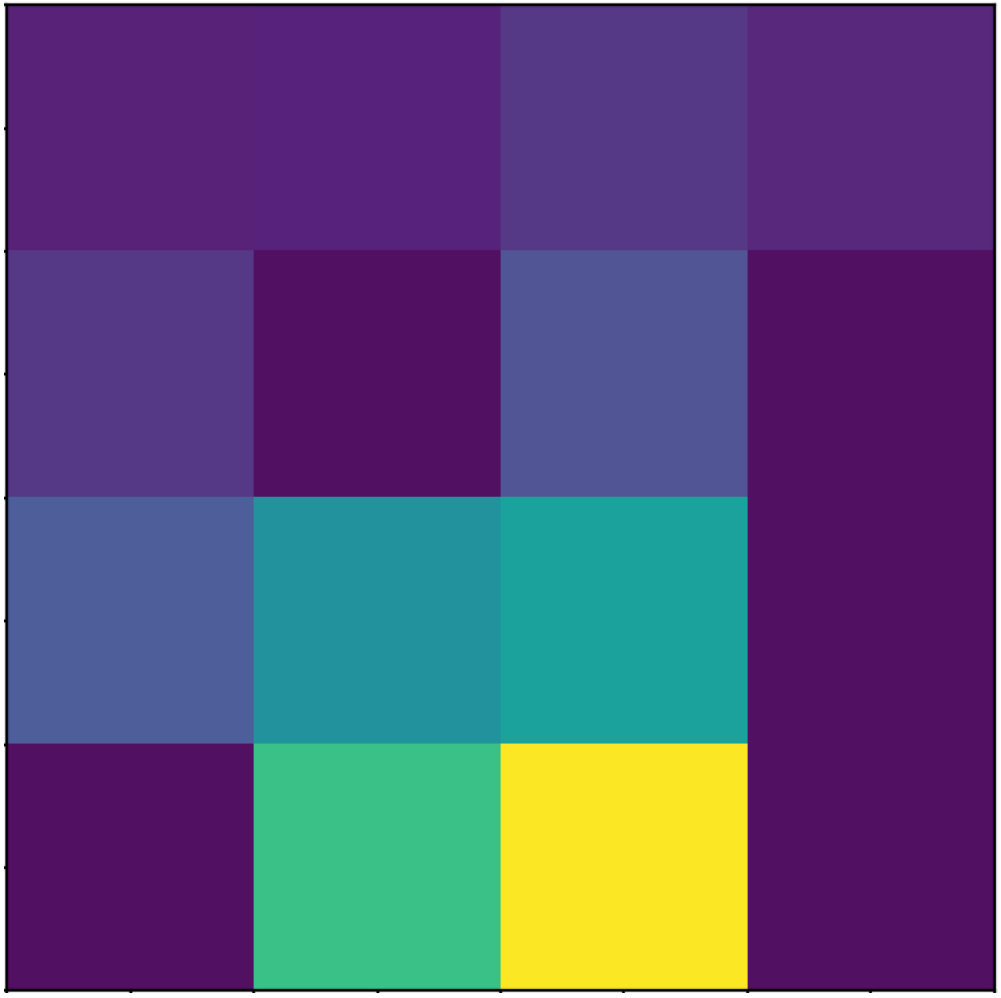


Maximum Total Expected Reward
From *t* to the end (*T-1)*
where we set *T = 10*

Reinforcement Learning Task for Autonomous Agent

Let's reshape the states back into the grid, and visualize the **Maximum Total Expected Reward** for each time step

SFFF
FHFH
FFFH
HFFG

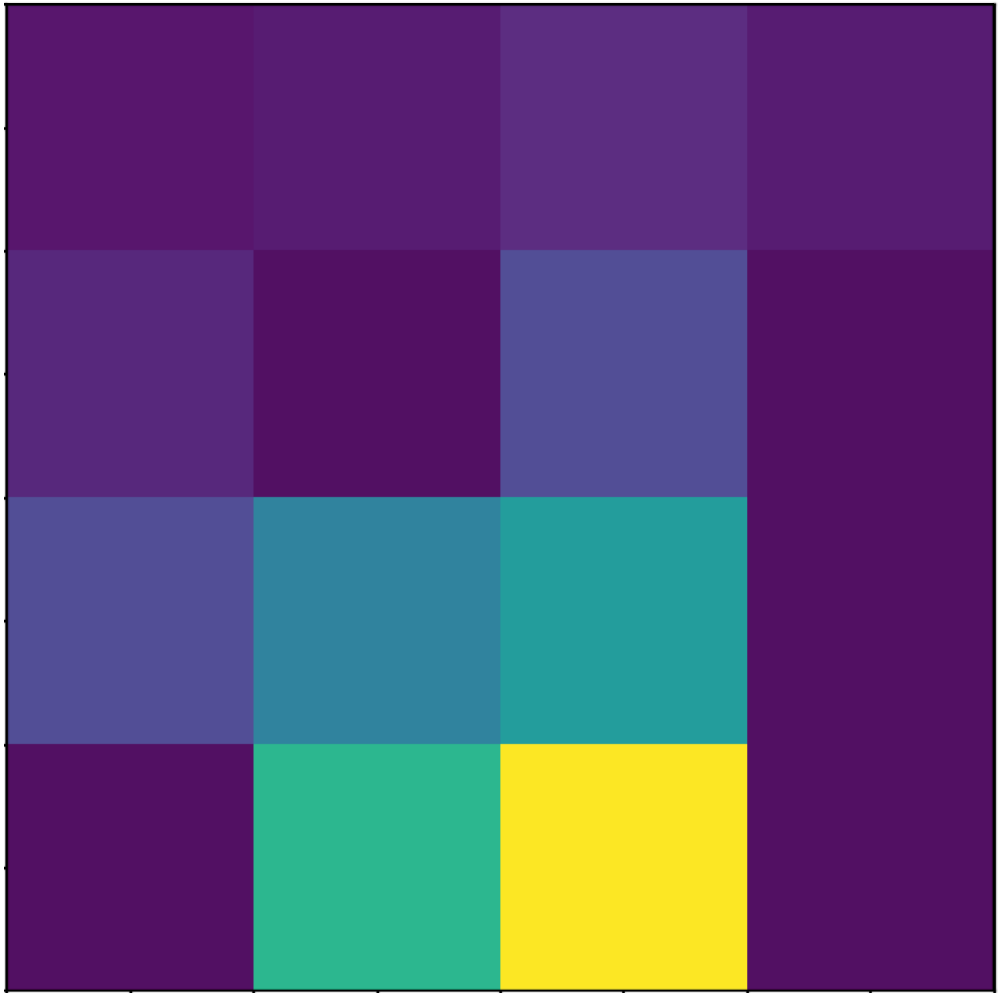


First time step

Reinforcement Learning Task for Autonomous Agent

Let's reshape the states back into the grid, and visualize the **Maximum Total Expected Reward** for each time step

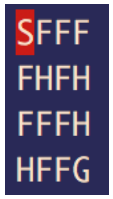
SFFF
FHFH
FFFH
HFFG

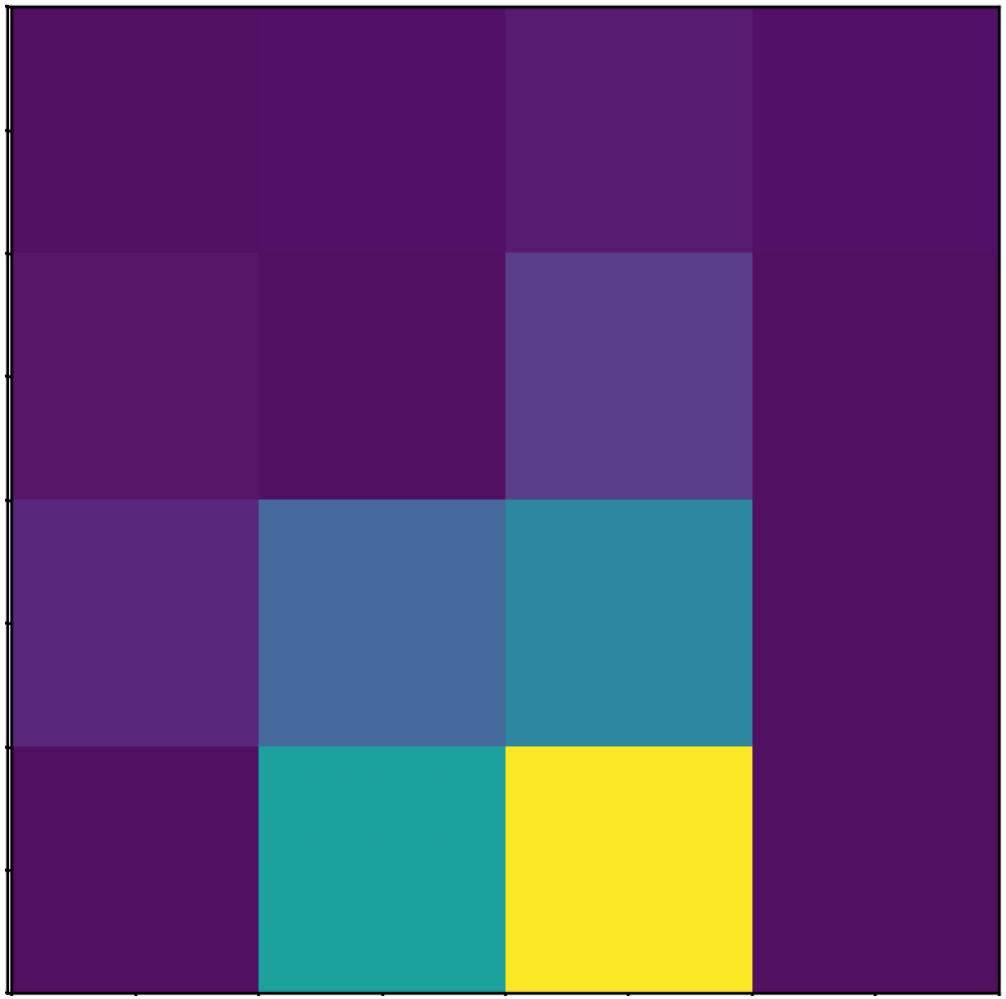


Intermediate time step

Reinforcement Learning Task for Autonomous Agent

Let's reshape the states back into the grid, and visualize the **Maximum Total Expected Reward** for each time step

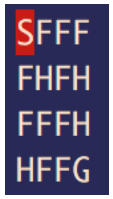


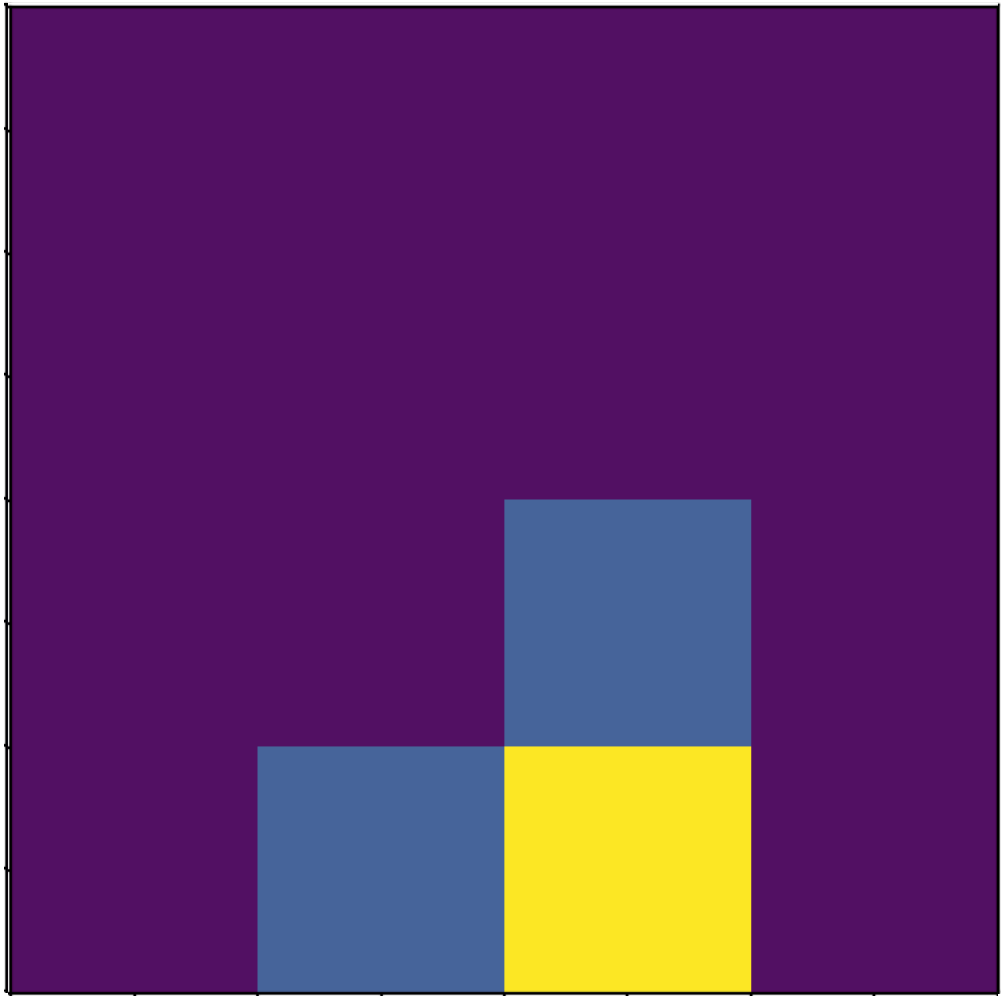


Intermediate time step

Reinforcement Learning Task for Autonomous Agent

Let's reshape the states back into the grid, and visualize the **Maximum Total Expected Reward** for each time step



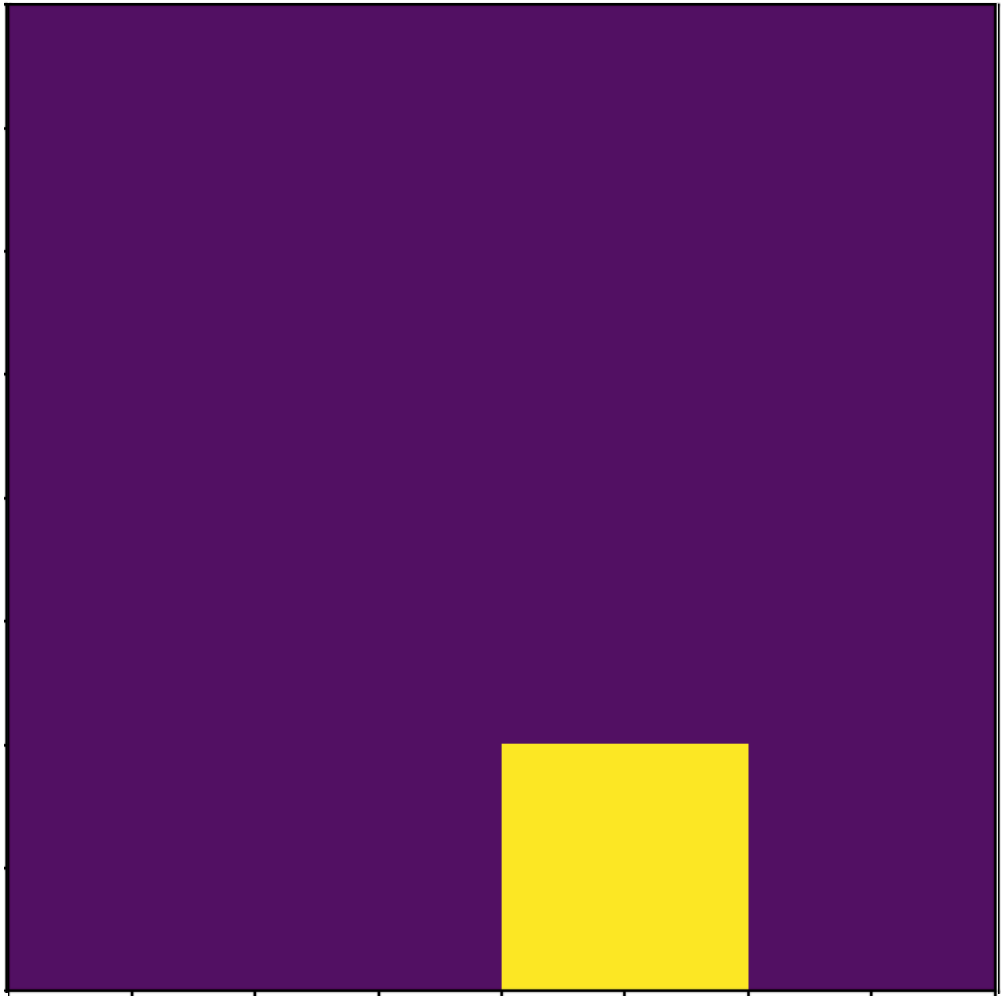


Intermediate time step

Reinforcement Learning Task for Autonomous Agent

Let's reshape the states back into the grid, and visualize the **Maximum Total Expected Reward** for each time step

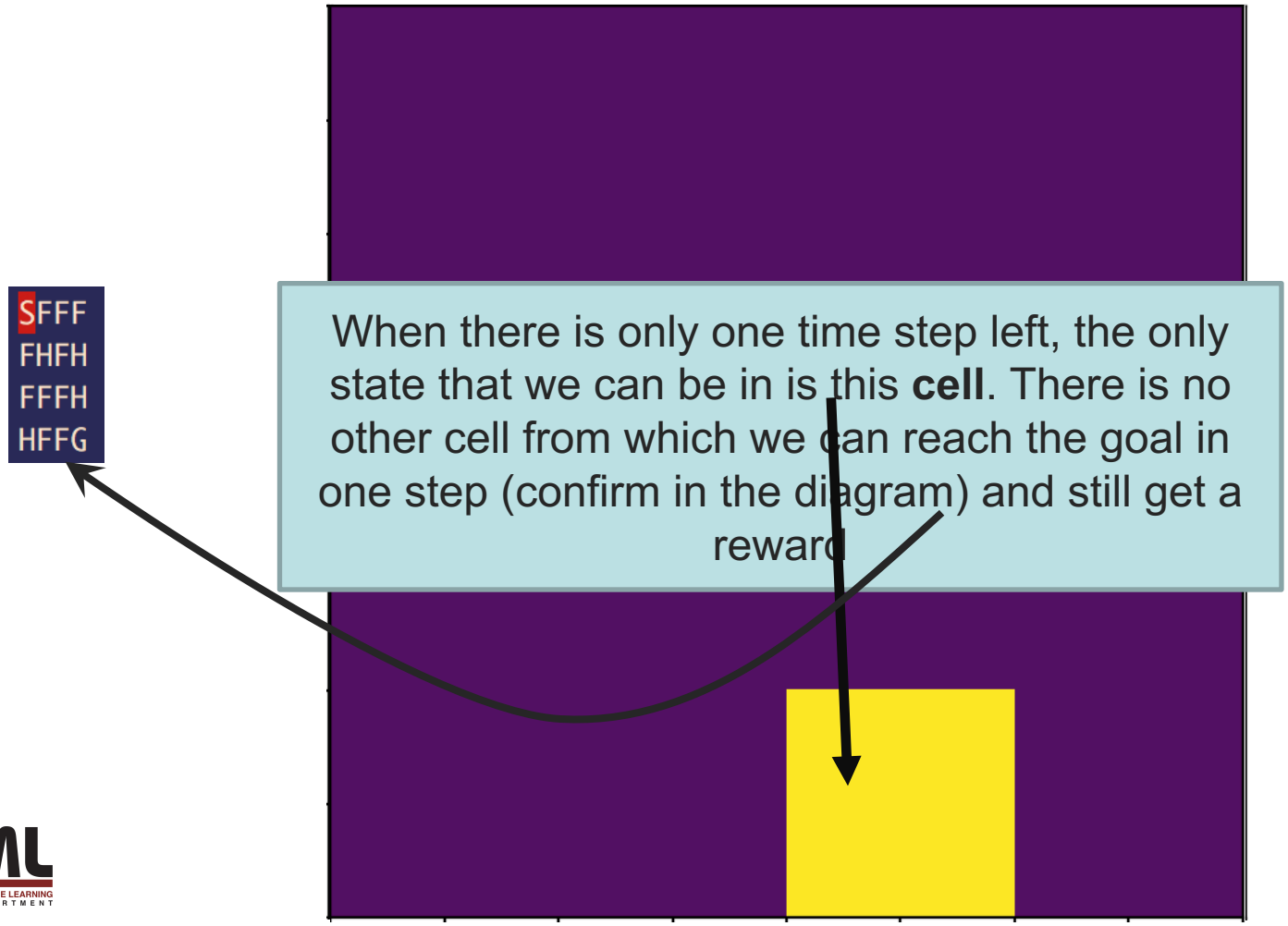
SFFF
FHFH
FFFH
HFFG



Last time-step

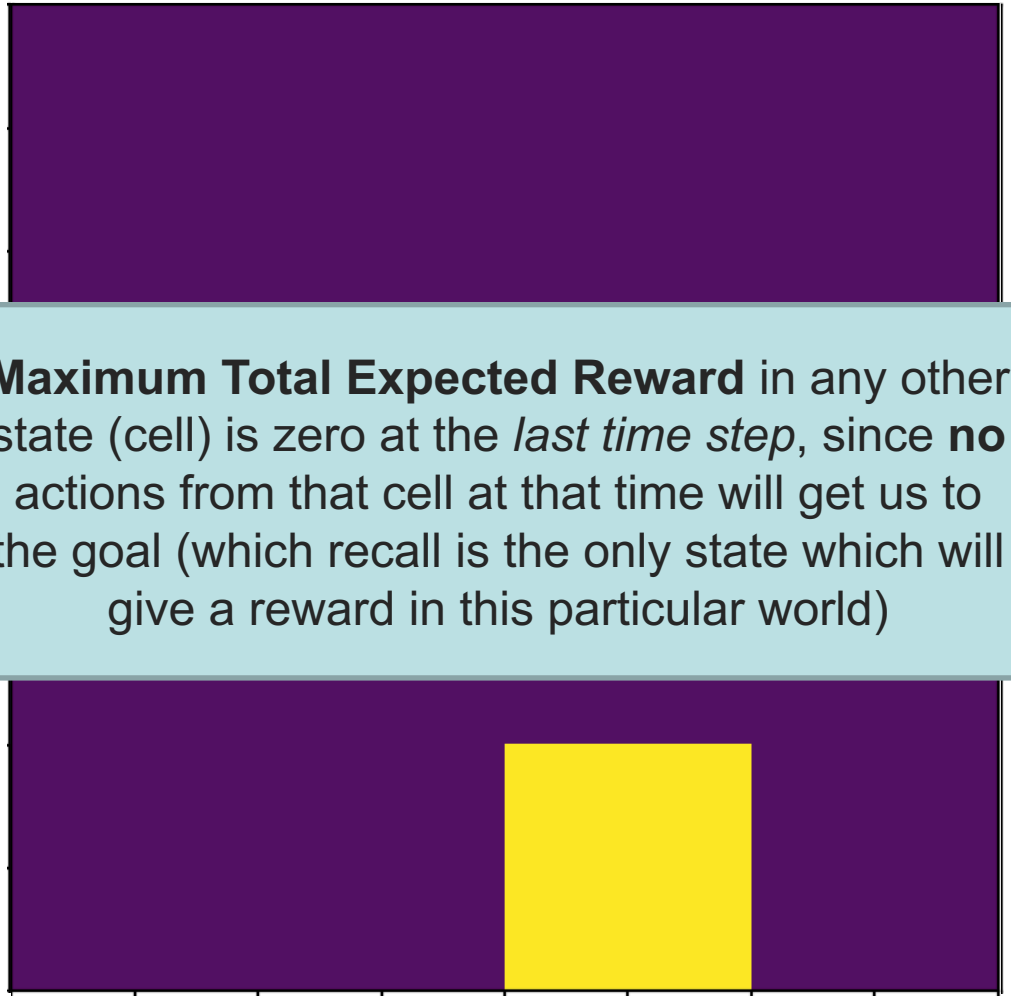
Reinforcement Learning Task for Autonomous Agent

Let's reshape the states back into the grid, and visualize the **Maximum Total Expected Reward** for each time step



Reinforcement Learning Task for Autonomous Agent

Let's reshape the states back into the grid, and visualize the **Maximum Total Expected Reward** for each time step



Maximum Total Expected Reward in any other state (cell) is zero at the *last time step*, since **no** actions from that cell at that time will get us to the goal (which recall is the only state which will give a reward in this particular world)

Last time-step

Reinforcement Learning Task for Autonomous Agent

But what if it doesn't make sense to have a finite time-span *T*? The robot may roam around **indefinitely** in the world.

Idea: let's let the agent roam around forever, and collect rewards. How can we compute the actions it should take from now to infinity?

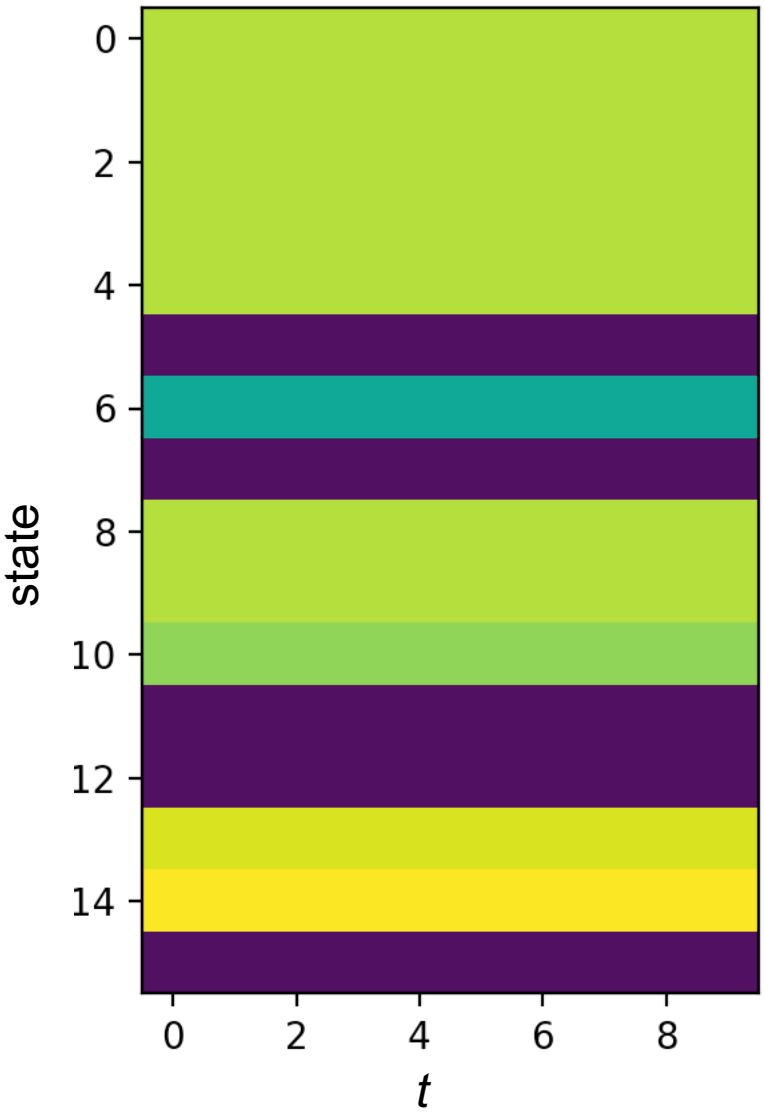
Reinforcement Learning Task for Autonomous Agent

But what if it doesn't make sense to have a finite time-span *T*? The robot may roam around indefinitely in the world.

Let's simulate it!

Let's make *T* from 10 (as it was in the previous example) to 1000 or 10000, and look at the **maximum total expected reward** for the first 10 time steps.

Reinforcement Learning Task for Autonomous Agent

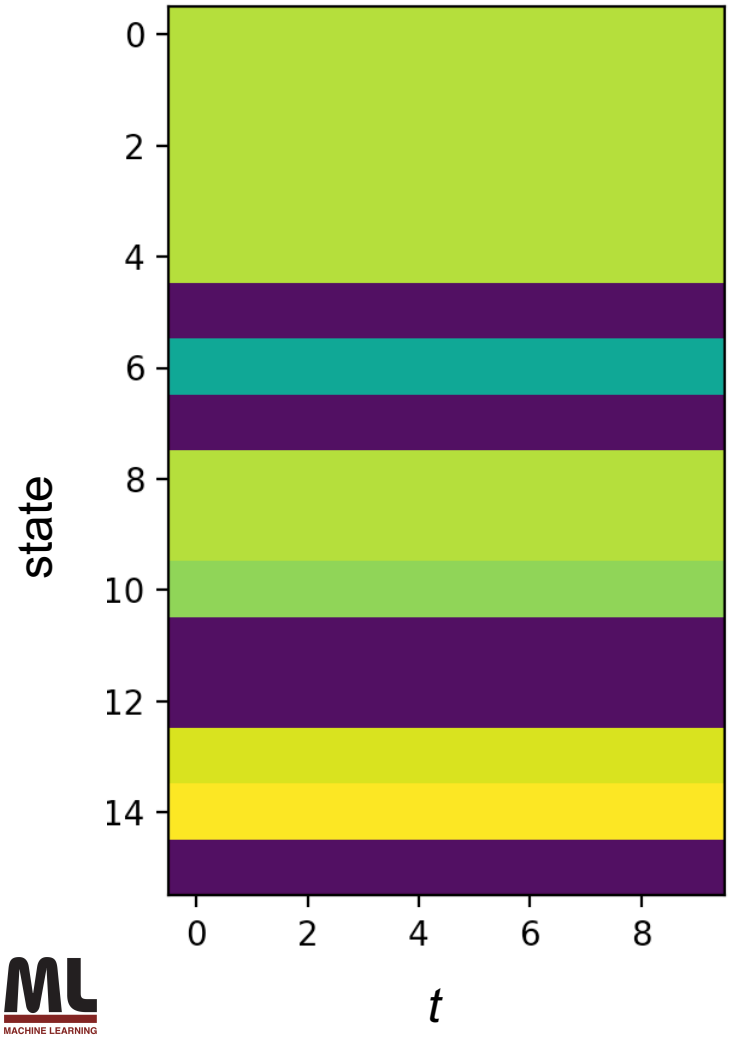


Maximum Total Expected Reward
From *t* to the end (*T-1)*
Where *T = 1000*, but only showing first 10 time-steps

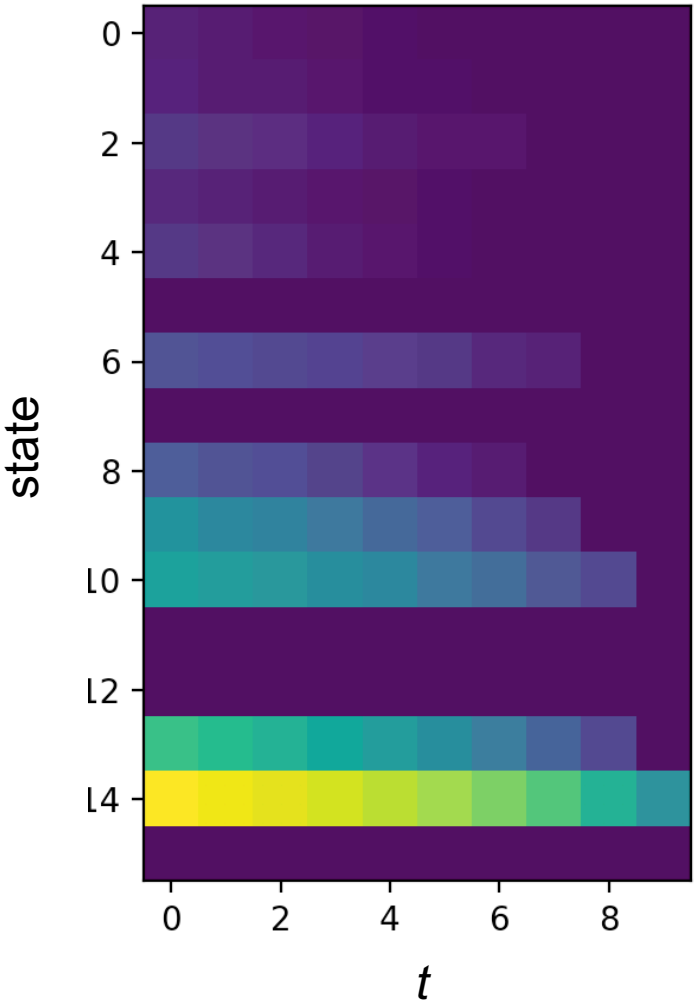
Reinforcement Learning Task for Autonomous Agent

Compare **Maximum Total Expected Reward** between *T=1000 and T=10*

T = 1000

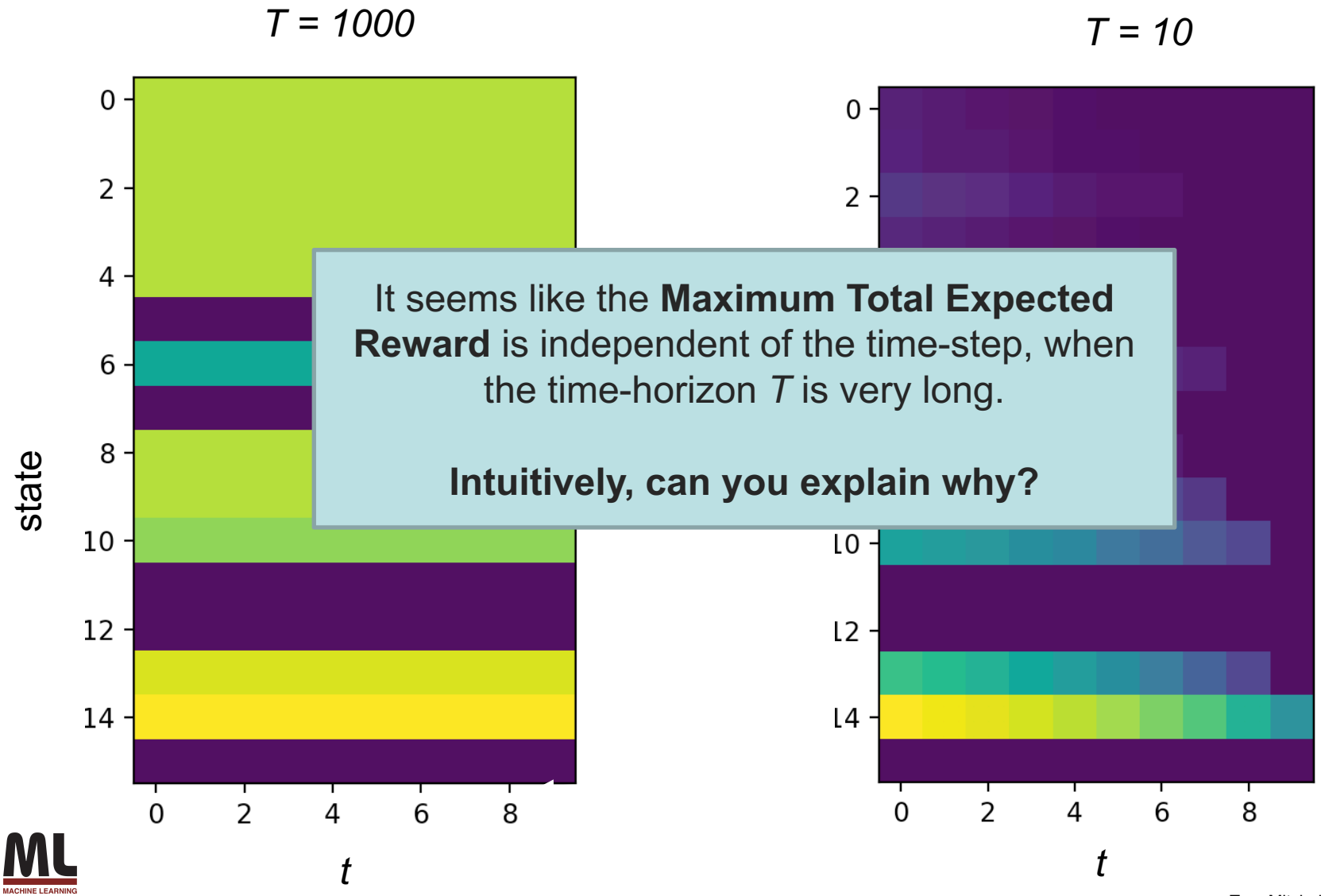


T = 10



Reinforcement Learning Task for Autonomous Agent

Compare **Maximum Total Expected Reward** between *T=1000 and T=10*



It seems like the **Maximum Total Expected Reward** is independent of the time-step, when the time-horizon *T* is very long.

Intuitively, can you explain why?

Reinforcement Learning Task for Autonomous Agent

Maximum Total Expected Reward at some state during some time-step is not particularly interesting. We care about knowing what *action* the agent should take at that state during that time-step, to maximize that reward.

Can we easily modify the code for compute **Maximum Total Expected Reward** to recover the optimal set of actions to get that reward?

Reinforcement Learning Task for Autonomous Agent

Maximum Total Expected Reward at some state during some time-step is not particularly interesting. We care about knowing what *action* the agent should take at that state during that time-step, to maximize that reward.

Can we easily modify the code for compute **Maximum Total Expected Reward** to recover the optimal set of actions to get that reward?

First, let's create a table of **Maximum Total Expected Reward** for every state in every time step

```
V = np.zeros((env.observation_space.n, T))
for s in range(env.observation_space.n):
    vals   = [ max_reward(s, t, T) for t in range(T) ]
    V[s,:] = vals
```

Reinforcement Learning Task for Autonomous Agent

Can we easily modify the code for compute **Maximum Total Expected Reward** to recover the optimal set of actions to get that reward?

First, let's create a table of **Maximum Total Expected Reward** for every state in every time step

```
V = np.zeros((env.observation_space.n, T))
for s in range(env.observation_space.n):
    vals   = [ max_reward(s, t, T) for t in range(T) ]
    V[s,:] = vals
```

Let's now recover the actions based on the table for **maximum total expected reward**:

```
def get_policy(s,t,V):
    Q_sat = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sat[action] += prob * (reward + V[s_next, t])

    return np.argmax(Q_sat)
```

Reinforcement Learning Task for Autonomous Agent

Compare this with the code we wrote to compute the **maximum total expected reward earlier**

```
def max_reward(s, t, T):
    Q_sat = [ 0. ] * env.action_space.n

    if t == t_max:
        return 0.0

    for a in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][a]:
            Q_sat[a] += prob * ( reward + max_reward(s_next, t+1, T) )

    max_Q = max(Q_sat)
    return max_Q
```

```
V = np.zeros((env.observation_space.n, T))
for s in range(env.observation_space.n):
    vals   = [ max_reward(s, t, T) for t in range(T) ]
    V[s,:] = vals
```

And our code for extracting the optimal set of actions to achieve the maximum total expected reward. **It's almost identical**.

```
def get_policy(s,t,V):
    Q_sat = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sat[action] += prob * (reward + V[s_next, t])

    return np.argmax(Q_sat)
```

Reinforcement Learning Task for Autonomous Agent

What are these optimal actions for every state in every time-step?

Let's compute the optimal actions for each time-step in every state based on the code we just wrote. Let's do that for both the time-horizon *T=10* and *T=1000*

```
[1, 1,                                                    0]
[3, 3,                                                    3]
[2, 2, 2, 2, 2, 0, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 3, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 3, 1, 1, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 2, 2, 2, 2, 2, 2, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Reinforcement Learning Task for Autonomous Agent

What are these optimal actions for every state in every time-step?

T = 10

```
[1, 1, 1, 2, 2, 1, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 2, 1, 0, 0, 0]
[2, 2, 2, 2, 2, 0, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 3, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 3, 1, 1, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 2, 2, 2, 2, 2, 2, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

T = 1000

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

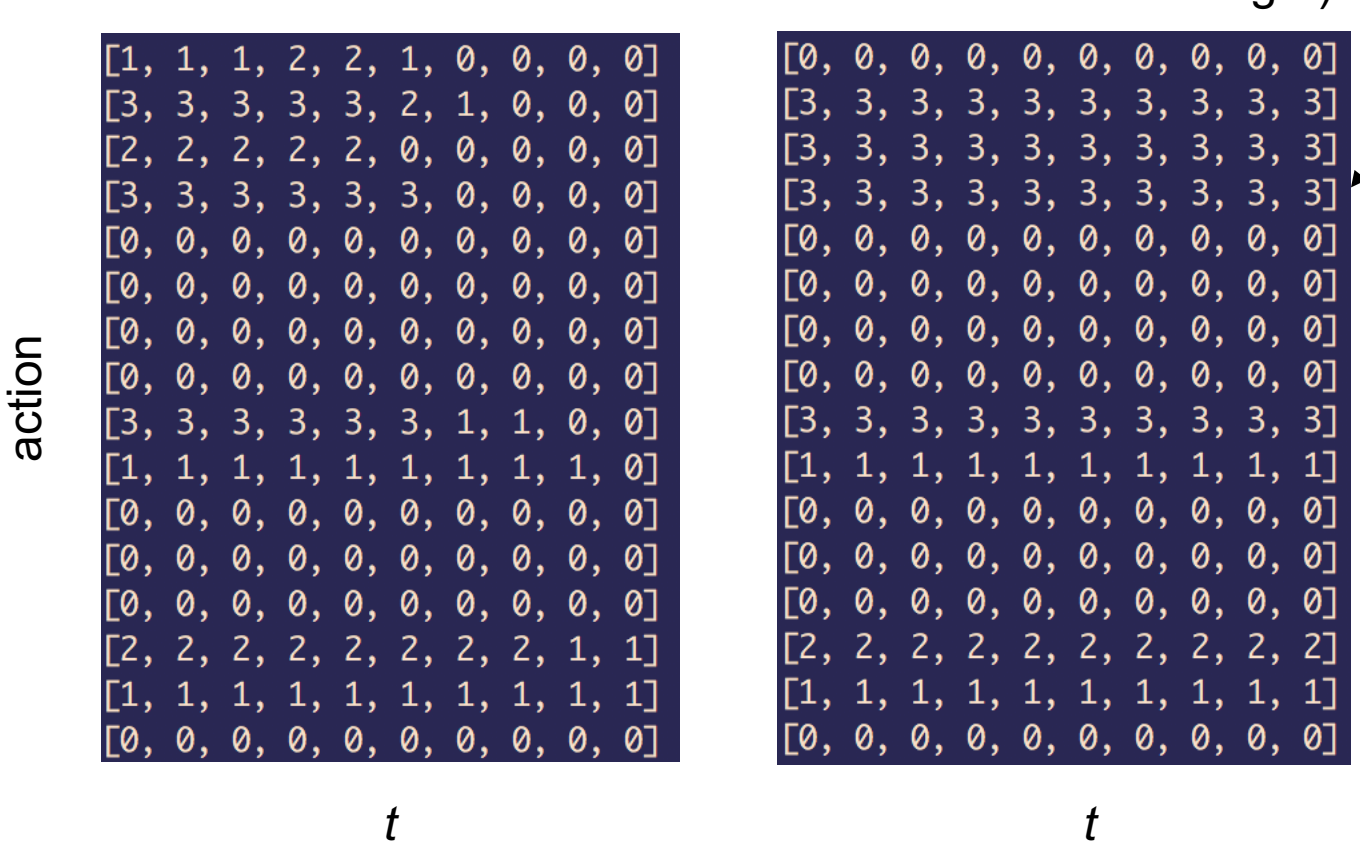
Reinforcement Learning Task for Autonomous Agent

Compare **Actions** that lead to **maximum** total reward between *T=10 and T=1000*

Each number is a discrete action (e.g., up, down, left, right)

T = 10

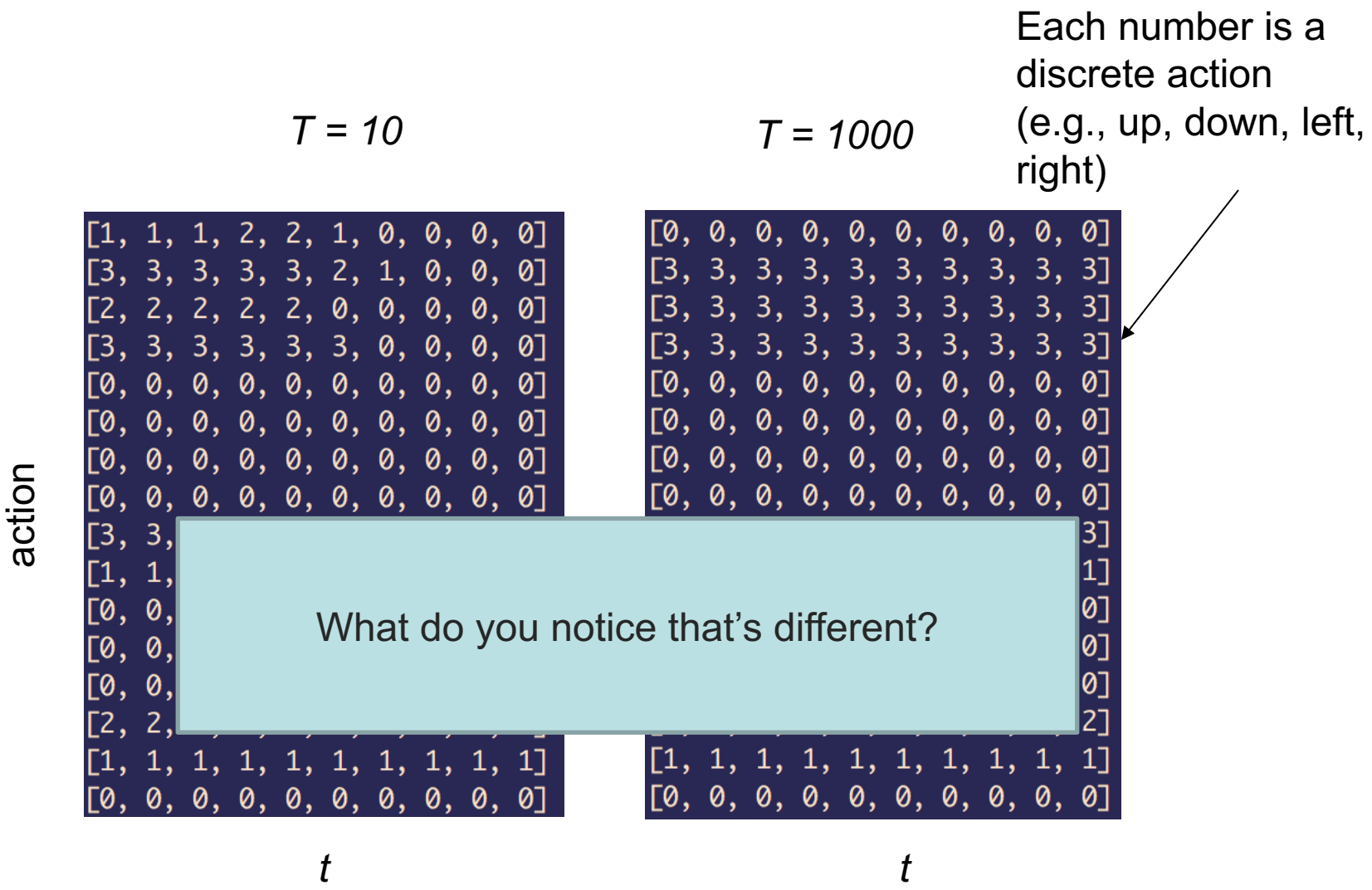
T = 1000



Reinforcement Learning Task for Autonomous Agent

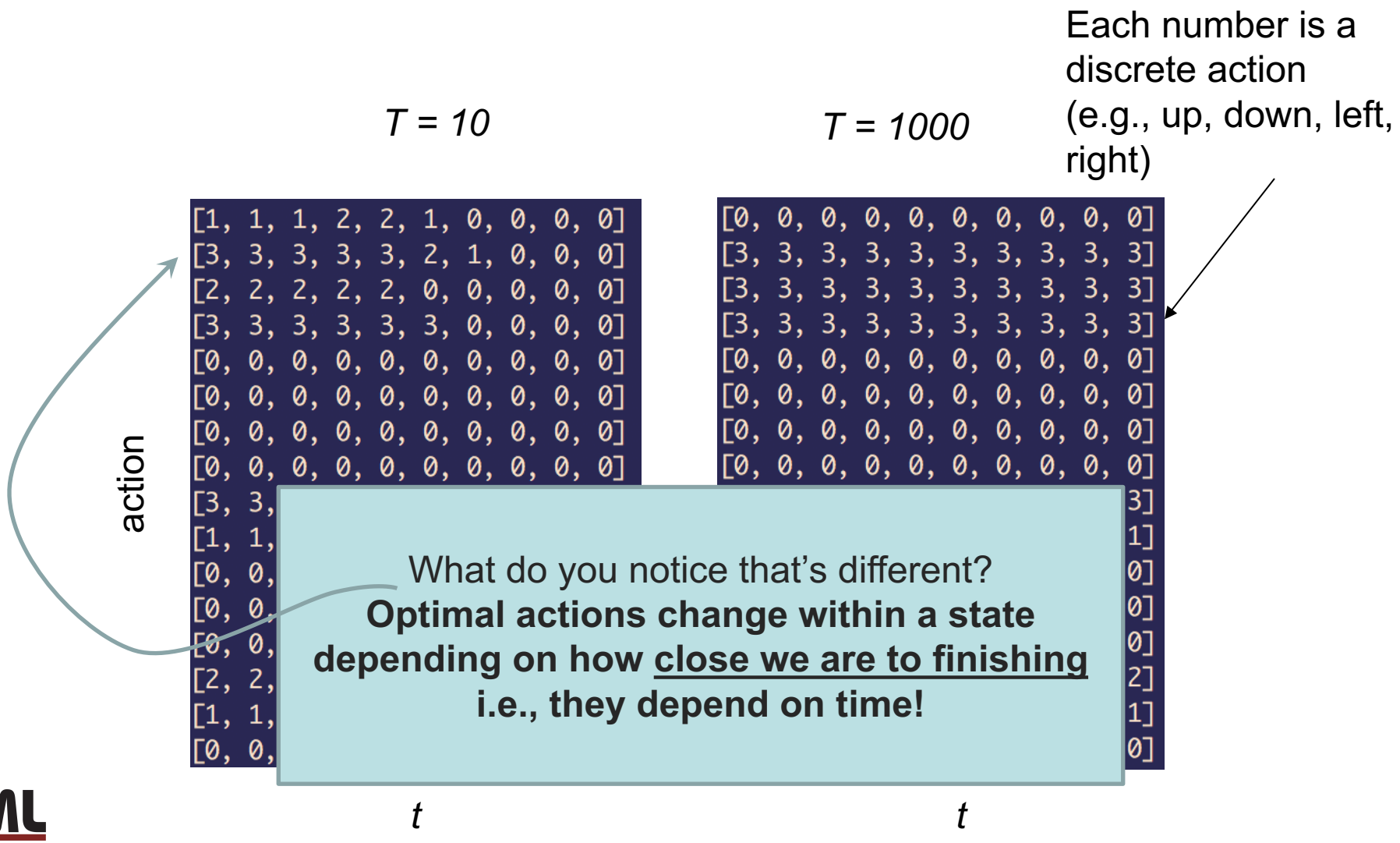
Compare **Actions** that lead to **maximum** total reward between *T=1000 and T=10*

Each number is a discrete action (e.g., up, down, left, right)



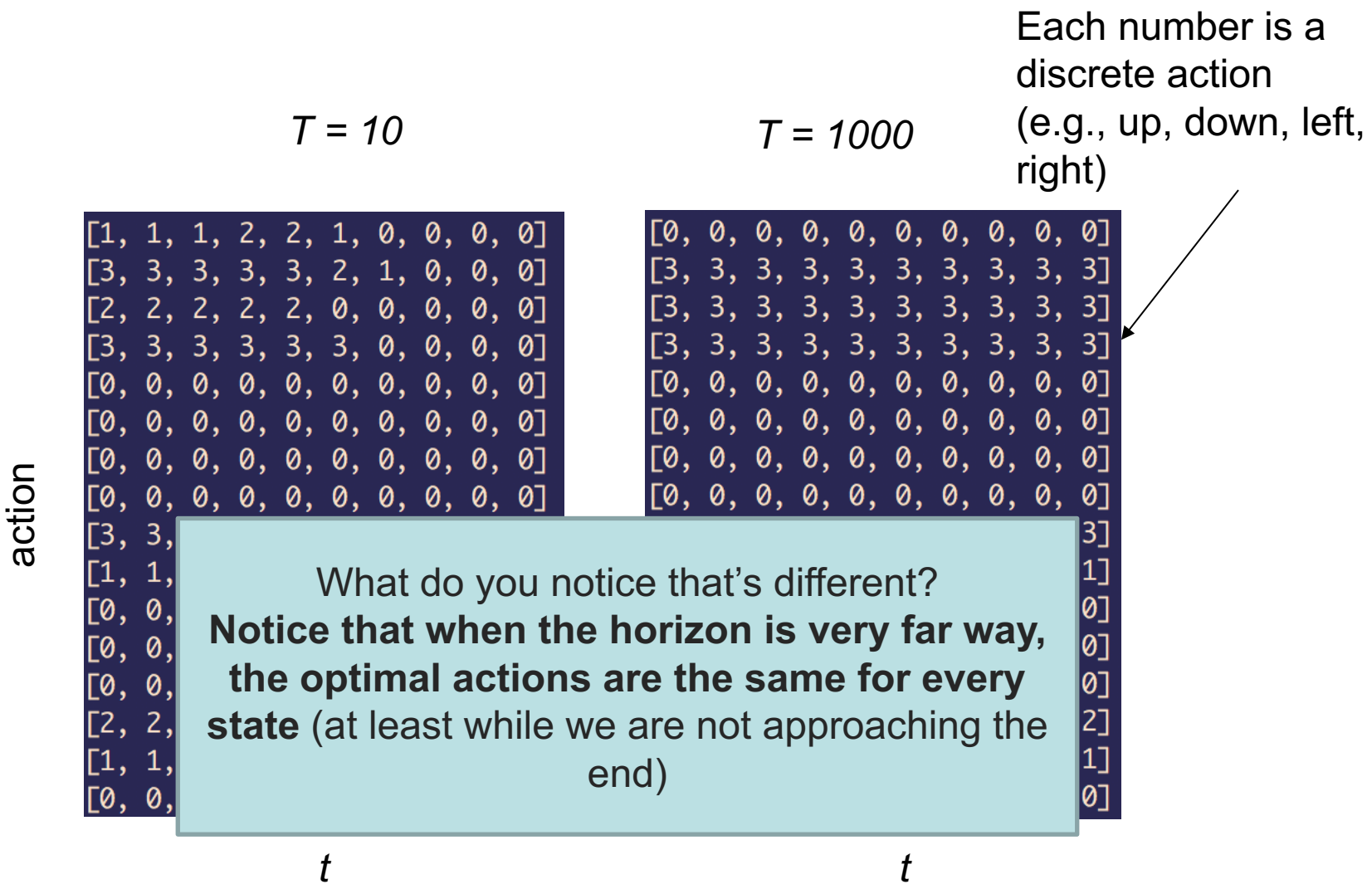
Reinforcement Learning Task for Autonomous Agent

Compare **Actions** that lead to **maximum** total reward between *T=1000 and T=10*



Reinforcement Learning Task for Autonomous Agent

Compare **Actions** that lead to **maximum** total reward between *T=1000 and T=10*



Reinforcement Learning Task for Autonomous Agent

If we make the time horizon approach infinity, it appears that we don't need to know what time-step we are in at all!

It may be enough to simply know how to act in a certain state, and time doesn't matter!

In general, it is true that in a Markov Decision Process with an infinite time-horizon, there exist an optimal **stationary policy**, i.e., function from states to actions, **independent of time**.

Reinforcement Learning Task for Autonomous Agent

If we make the time horizon approach infinity, it appears that we don't need to know what time-step we are in at all!

It may be enough to simply know how to act in a certain state, and time doesn't matter!

In general, it is true that in a Markov Decision Process with an infinite time-horizon, there exist an optimal **stationary policy**, i.e., function from states to actions, **independent of time**.

How do we compute it practically?
Should we simulate a very long chain like we just did?

Or is there a better way?

Reinforcement Learning Task for Autonomous Agent

How do we compute it practically?
Should we simulate a very long chain like we just did?

Since we can assume that the maximum total expected reward from any time to the end (let's say infinity) is the same, then we don't really need to index maximum total expected reward by time at all.

We can rewrite our original equation, dropping the time index:

Finite Time Horizon

$$TR_{t\ldots T-1}(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + TR_{t\ldots T-1}(s_{t+1}) \right\}$$

$$TR(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + TR(s_{t+1}) \right\}$$

Infinite Time Horizon

Reinforcement Learning Task for Autonomous Agent

Finite Time Horizon

$$TR_{t\ldots T-1}(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + TR_{t\ldots T-1}(s_{t+1}) \right\}$$

$$TR(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + TR(s_{t+1}) \right\}$$

Infinite Time Horizon

In literature, this maximum expected total reward from some state for all time steps to infinity called the **value function**, i.e., a function from **state** to the **expected total reward** from following the optimal policy for infinite time.

Reinforcement Learning Task for Autonomous Agent

$$TR(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + TR(s_{t+1}) \right\}$$

Infinite Time Horizon

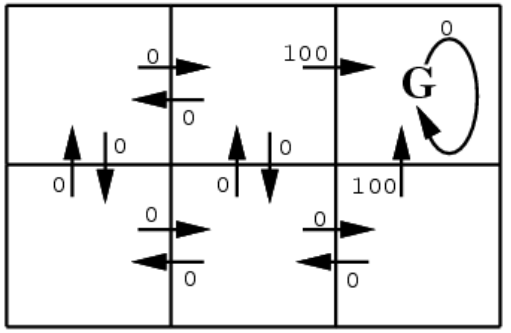
To compute it, we can initialize the value of each state to a random number, and then update the value using the above formula until the values of all states converge. Using the notation of *V(s)* to represent the value of a state, we can write:

$$V(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + V_{prev}(s_{t+1})] \right\}$$

where $V_{prev}(s)$ is the previous value computed for state *s* before updating.

The algorithm for computing the **value function** in this iterative fashion is known as **Value Iteration**

Value Function for each Policy



- For the case with an infinite horizon:
- Given a policy π : S → A, define

$$V^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t r_t]$$

assuming action sequence chosen according to π, starting at state *s*

- Then we want the *optimal* policy π^* where

$$\pi^* = \arg\max_{\pi} V^{\pi}(s), \quad (\forall s)$$

- For any MDP, such a policy exists!

Value Iteration

Interestingly, value iteration works even if we randomly traverse the environment instead of looping through each state and action methodically

- but we must still visit each state infinitely often on an infinite run
- For details: [Bertsekas 1989]
- Implications: online learning as agent randomly roams

Value Iteration for learning V* : assumes $P(S_{t+1}|S_t, A)$ known

Here is how we can modify our original DP algorithm for computing the **maximum total expected reward** in the finite horizon case to make into a **value iteration algorithm** for the infinite horizon case

```
def value_iteration():
    # initialize value function to 0 for every state
    v = [ 0. ] * env.observation_space.n

    # pick a fixed number of iterations (hopefully it will converge
    # within this many iterations (might need to increase if it doesnt)
    for iter in range(1000):
        # go through every state
        for s in range(env.observation_space.n):
            Q_sa = [ 0. ] * env.action_space.n

            # consider all actions possible
            for a in range(env.action_space.n):
                # see what next state this action can take us
                # and with what probability
                # and reward
                for prob, s_next, reward, _ in env.env.P[s][a]:
                    Q_sa[a] += prob * (reward + v[s_next])

            # update the value according to the best action
            v[s] = max(Q_sa)
    return v
```

Value Itera…, A) known

Here is ho…puting
the **maxi**…se to make
into a **valu**…

What's the biggest significant change from the finite horizon case?

```
def v…
    # initialize value function to 0 for every state
    v = [ 0. ] * env.observation_space.n

    # pick a fixed number of iterations (hopefully it will converge
    # within this many iterations (might need to increase if it doesnt)
    for iter in range(1000):
        # go through every state
        for s in range(env.observation_space.n):
            Q_sa = [ 0. ] * env.action_space.n

            # consider all actions possible
            for a in range(env.action_space.n):
                # see what next state this action can take us
                # and with what probability
                # and reward
                for prob, s_next, reward, _ in env.env.P[s][a]:
                    Q_sa[a] += prob * (reward + v[s_next])

            # update the value according to the best action
            v[s] = max(Q_sa)
    return v
```

Value Itera…, A) known

Here is ho…puting
the **maxi**…se to make
into a **valu**…

What's the biggest significant change from the finite horizon case?

Will this even converge?

```
def v…
    # initialize value function to 0 for every state
    v… = [ 0 ] * env.observation_space.n

    #
    #                                                    nt)
    f…

            for a in range(env.action_space.n):
                # see what next state this action can take us
                # and with what probability
                # and reward
                for prob, s_next, reward, _ in env.env.P[s][a]:
                    Q_sa[a] += prob * (reward + v[s_next])

            # update the value according to the best action
            v[s] = max(Q_sa)
    return v
```

Reinforcement Learning Task for Autonomous Agent

$$V(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + V_{prev}(s_{t+1})] \right\}$$

where $V_{prev}(s)$ is the previous value computed for state *s* before updating.

Is this guaranteed to converge?

Reinforcement Learning Task for Autonomous Agent

$$V(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + V_{prev}(s_{t+1})] \right\}$$

where $V_{prev}(s)$ is the previous value computed for state *s* before updating.

Is this guaranteed to converge?

Since this algorithm is computing the **maximum expected total reward** for an **infinite horizon**, we have to be careful so it doesn't blow-up!

In some cases, depending on the reward and state transition function, it may converge, but in other cases, it may not. We can ensure it always converges by adding a discount coefficient $\gamma < 1$ to ensure the rewards infinitely far in the future approach 0

$$V(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + \gamma V_{prev}(s_{t+1})] \right\}$$

Reinforcement Learning Task for Autonomous Agent

$$V(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + V_{prev}(s_{t+1})] \right\}$$

where V_{pr ...efore updating.

Is this guara...

Since this alg... **reward** for an **infinite h...**

Sanity check: Run **Value Iteration** and compare the results to the dynamic program with a **finite horizon** that's long enough and see if they yield very similar results!

In some cases, depending on the reward and state transition function, it may converge, but in other cases, it may not. We can ensure it always converges by adding a discount coefficient $\gamma < 1$ to ensure the rewards infinitely far in the future approach 0

$$V(s_t) = \max_{a_t} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t)[R(s_{t+1}, s_t, a_t) + \gamma V_{prev}(s_{t+1})] \right\}$$

So far: learning optimal policy when we know $P(s_t \mid s_{t-1}, a_{t-1})$

What if we don't?

Q learning

Recall our original code to get the optimal sequence of actions in a finite horizon MDP:

```
def get_policy(s,t,V):
    Q_sat = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sat[action] += prob * (reward + V[s_next, t])

    return np.argmax(Q_sat)
```

Q learning

Recall our original code to get the optimal sequence of actions in a finite horizon MDP:

```
def get_policy(s,t,V):
    Q_sat = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sat[action] += prob * (reward + V[s_next, t])

    return np.argmax(Q_sat)
```

Let's transform this to deal with the case of an infinite horizon.

Q learning

Recall our original code to get the optimal sequence of actions in a finite horizon MDP:

```
def get_policy(s,t,V):
    Q_sat = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sat[action] += prob * (reward + V[s_next, t])

    return np.argmax(Q_sat)
```

Let's transform this to deal with the case of an infinite h

What changes would have to be made to the above code?

Q learning

Recall our original code to get the optimal sequence of actions in a finite horizon MDP:

```
def get_policy(s,t,V):
    Q_sat = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sat[action] += prob * (reward + V[s_next, t])

    return np.argmax(Q_sat)
```

Let's transform this to deal with the case of an infinite horizon.

Since we said that the value of a state doesn't depend on time when the horizon is infinite, we can simply get rid of the time step index

Q learning

Recall our original code to get the optimal sequence of actions in a finite horizon MDP:

Finite Horizon

```
def get_policy(s,t,V):
    Q_sat = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sat[action] += prob * (reward + V[s_next, t])

    return np.argmax(Q_sat)
```

```
def get_policy_infinite(s,V):
    Q_sa = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sa[action] += prob * (reward + V[s_next])

    return np.argmax(Q_sa)
```

Infinite Horizon

Q learning

```
def get_policy_infinite(s,V):
    Q_sa = [ 0. ] * env.action_space.n
    for action in range(env.action_space.n):
        for prob, s_next, reward, _ in env.env.P[s][action]:
            Q_sa[action] += prob * (reward + V[s_next])

    return np.argmax(Q_sa)
```

Notice that although we needed to know the transition probability and reward function to compute *Q*, to make the actual decision for the best action, we only need to have these *Q* values and nothing else.

This means that IF we have some other means of getting (estimating) Q, we don't need to know the true transition probabilities or the reward function to act optimally (learn a policy)!

Q learning

Main idea behind **Q learning:**

Estimate *Q(s,a)* as we (the agent) explore the world and collects rewards.

Q learning

Main idea behind **Q learning:**

Estimate *Q(s,a)* as we (the agent) explore the world and collect rewards.

Recall, that when the agent performs an action *a,* the world "samples" the next state from a transition distribution P(s_next | s, a), and then gives the agent the reward based on this **next state**, the **current state** and the **action**.

We can thus interpret each reward we receive in state *s* after performing action *a* as a sample from a distribution over rewards!

Q learning

Main idea behind **Q learning:**

That means we can try and estimate the Q value of a state action pair (s,a) empirically by averaging.

Q learning

Main idea behind **Q learning:**

That means we can try and estimate the Q value of a state action pair (s,a) empirically by averaging.

Suppose, we visited state *s* and performed action *a* in that state $n(s,a)$ times.

Every time we visit state *s* and execute action *a,* we increment that count.

We can then write down an estimate of *Q(s,a)* after that many visits to state *s* and having performed action *a.*

Q learning

Main idea behind **Q learning:**

We can then write down an estimate of *Q(s,a)* after that many visits to state *s* and having performed action *a.*

Reward that was sampled on the nth application of action *a* in state *s*

Our best estimate of future expected maximum reward

$$\hat{Q}^{(n)}(s,a) = \frac{1}{n(s,a)} \sum_{n}^{n(s,a)} P(s^{(n)}_{next} \mid s,a)[R_n(s^{(n)}_{next}, s, a) + \max_{a'}\{Q^{(n-1)}(s^{(n)}_{next}, a'\}$$

Estimate of *Q(s,a)* after nth application of action *a* in state *s.*

Number of times action *a* was applied in state *s.*

Next state that was ”sampled” by the world on nth

Q learning

Main idea behind **Q learning:**

$$\hat{Q}^{(n)}(s,a) = \frac{1}{n(s,a)} \sum_{n}^{n(s,a)} P(s_{next}^{(n)} \mid s,a)[R_n(s_{next}^{(n)}, s, a) + \max_{a'}\{Q^{(n-1)}(s_{next}^{(n)}, a'\}$$

Can we compute the estimate of Q without having to remember all of our previous state visits and rewards?

Q learning

Main idea behind **Q learning:**

$$\hat{Q}^{(n)}(s,a) = \frac{1}{n(s,a)} \sum_{n}^{n(s,a)} P(s_{next}^{(n)} \mid s,a)[R_n(s_{next}^{(n)}, s, a) + \max_{a'}\{Q^{(n-1)}(s_{next}^{(n)}, a'\}$$

Can we compute the estimate of Q without having to remember all of our previous state visits and rewards?

Rewrite the empirical estimate of Q in a way that allows us to compute it from the previous estimate of Q and the new observed state and reward

Q learning

Main idea behind **Q learning:**

$$\hat{Q}^{(n)}(s,a) = \frac{1}{n(s,a)} \sum_{n}^{n(s,a)} P(s_{next}^{(n)} \mid s,a)[R_n(s_{next}^{(n)}, s, a) + \max_{a'}\{Q^{(n-1)}(s_{next}^{(n)}, a'\}$$

Can we compute the estimate of Q without having to remember all of our previous state visits and rewards?

$$\hat{Q}^{(n+1)}(s,a) = \frac{n(s,a)}{n(s,a)+1}\hat{Q}^{n}(s,a) + \frac{1}{n(s,a)+1}\left([R_{n+1}(s_{next}^{(n+1)}, s, a) + \max_{a'}\{Q^{(n)}(s_{next}^{(n+1)}, a')]\}\right)$$

Q learning

Main idea behind **Q learning:**

$$\hat{Q}^{(n)}(s,a) = \frac{1}{n(s,a)} \sum_{n}^{n(s,a)} P(s_{next}^{(n)} \mid s,a)[R_n(s_{next}^{(n)}, s, a) + \max_{a'}\{Q^{(n-1)}(s_{next}^{(n)}, a'\}$$

Can we compute the estimate of Q without having to remember all of our previous state visits and rewards?

$$\hat{Q}^{(n+1)}(s,a) = \frac{n(s,a)}{n(s,a)+1}\hat{Q}^{n}(s,a) + \frac{1}{n(s,a)+1}\left([R_{n+1}(s_{next}^{(n+1)}, s, a) + \max_{a'}\{Q^{(n)}(s_{next}^{(n+1)}, a')]\}\right)$$

Previous estimate of this Q value

New reward and state "sampled" from the world

Q learning

Main ide

As new information comes in, we compute a weighted average between the previous estimate and the new "data-point"

$$\hat{Q}^{(n)}(s,a) = \frac{1}{n}$$

Can we compute the estimate of Q without having to remember all of our previous state visits and rewards?

$$\hat{Q}^{(n+1)}(s,a) = \frac{n(s,a)}{n(s,a)+1}\hat{Q}^{n}(s,a) + \frac{1}{n(s,a)+1}\left([R_{n+1}(s_{next}^{(n+1)},s,a) + \max_{a'}\{Q^{(n)}(s_{next}^{(n+1)},a')]\}\right)$$

Previous estimate of this Q value

New reward and state "sampled" from the world

Q learning

Main ide

Can be shown to converge. Intuitively, why?

$$\hat{Q}^{(n)}(s,a) = \frac{1}{n(s,a)} \sum_{i=1}^{n} [r_i + \max_{a'} \{\hat{Q}^{(n)}(s_{next}, a')\}]$$

Can we compute the estimate of Q without having to remember all of our previous state visits and rewards?

$$\hat{Q}^{(n+1)}(s,a) = \frac{n(s,a)}{n(s,a)+1} \hat{Q}^{n}(s,a) + \frac{1}{n(s,a)+1} \left([R_{n+1}(s_{next}^{(n+1)}, s, a) + \max_{a'} \{Q^{(n)}(s_{next}^{(n+1)}, a')]\} \right)$$

Previous estimate of this Q value

New reward and state "sampled" from the world

Q learning

Let's cook-up a simple implementation in the same FrozenLake world:

```
Q = np.zeros((env.observation_space.n, env.action_space.n))
rewards    = []
iterations = []

# Parameters
alpha = 0.75
discount = 0.95
episodes = 5000

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

Modified from this example:
https://www.oreilly.com/learning/introduction-to-reinforcement-learning-and-openai-gym

Q learning

Let's cook-up a simple implementation in the same FrozenLake world:

```
Q = np.zeros((env.observation_space.n, env.action_space.n))
rewards    = []
iterations = []

# Parameters
alpha = 0.75
discount = 0.95
episodes = 5000

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

Initialize Q values to all zeros
(as many of them as number of States X Actions)

Q learning

Let's cook-up a simple implementation in the same FrozenLake world:

```
Q = np.zeros((env.observation_space.n, env.action_space.n))
rewards    = []
iterations = []

# Parameters
alpha = 0.75
discount = 0.95
episodes = 5000

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

Initialize Q values to all zeros
(as many of them as number of States X Actions)

Pick the "best" action according to current Q estimates

Q learning

Let's cook-up a simple implementation in the same FrozenLake world:

```
Q = np.zeros((env.observation_space.n, env.action_space.n))
rewards    = []
iterations = []

# Parameters
alpha = 0.75
discount = 0.95
episodes = 5000

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

Initialize Q values to all zeros
(as many of them as number of States X Actions)

Pick the "best" action according to current Q estimates

Update Qs based on new reward and state

Let's cook-u e world:

Notice we repeat the simulation multiple times (number of episodes), resetting it in every episode.

```
Q = np.zeros((en
rewards    = []
iterations = []

# Parameters
alpha = 0.75
discount = 0.95
episodes = 5000

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

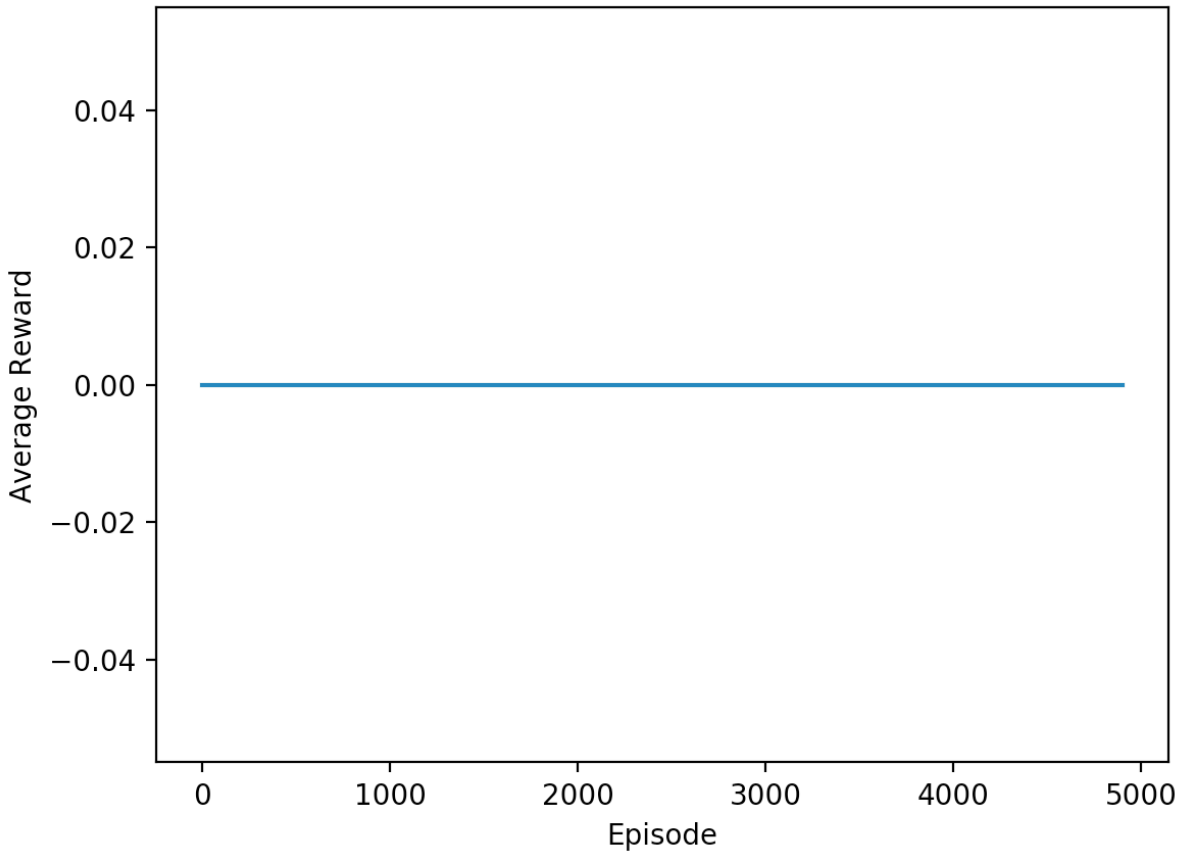
Initialize Q values to all zeros
(as many of them as number of States X Actions)

Pick the "best" action according to current Q estimates

Update Qs based on new reward and state

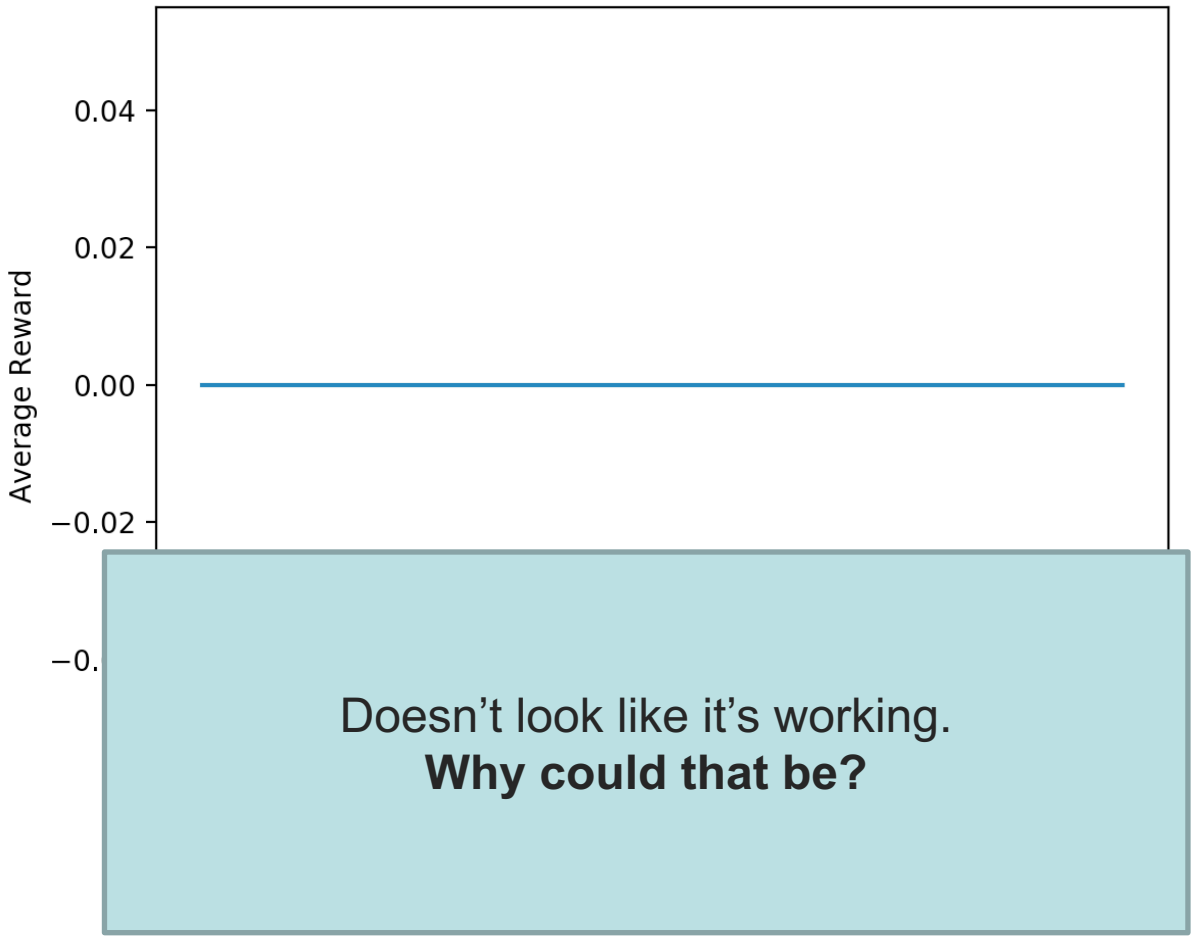
Q learning

Let's look at the average total reward as a function of episode



Q learning

Let's look at the average total reward as a function of episode



Doesn't look like it's working.
Why could that be?

Q learning

Let's cook-up a simple implementation in the same FrozenLake world:

```
Q = np.zeros((env.observation_space.n, env.action_space.n))
rewards    = []
iterations = []

# Parameters
alpha = 0.75
discount = 0.95
episodes = 5000

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

Initialize Q values to all zeros
(as many of them as number of States X Actions)

Pick the "best" action according to current Q estimates

Notice that because we initialized our Q values to all-zeros, we will always pick the next best state to be the same!

We will never explore any other states or actions, so we won't learn about them!

Let's cook … ke world:

```
Q = np.zeros(
rewards    = []
iterations = []

# Parameters
alpha = 0.75
discount = 0.95
episodes = 5000

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

Initialize Q values to all zeros
(as many of them as number of States X Actions)

Pick the "best" action according to current Q estimates

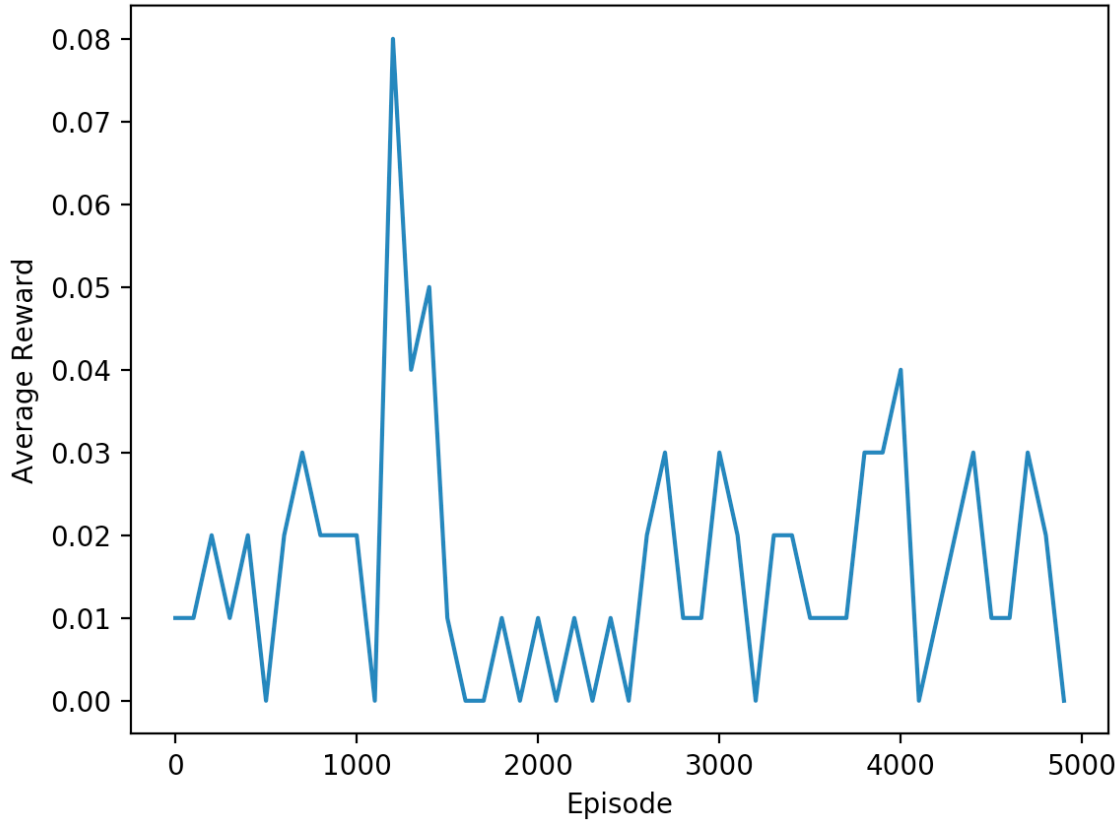
We can fix it by randomizing the initial Q

Let's cook ke world:

```
Q = 0.01*np.random.rand(env.observation_space.n, env.action_space.n)
```

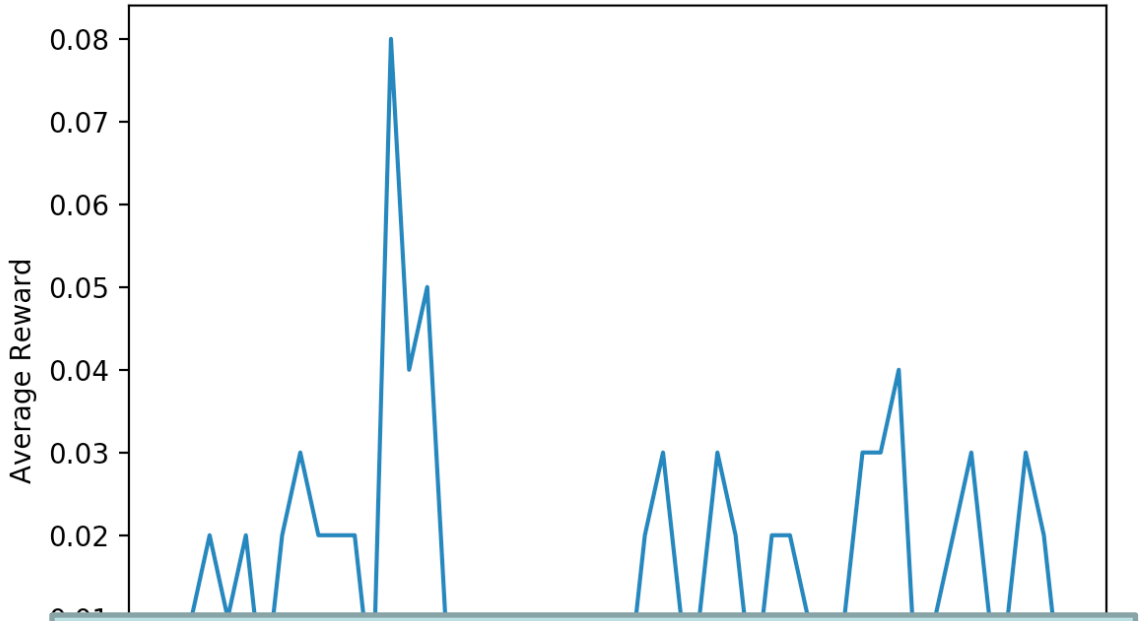
Q learning

Let's see now what the average reward as a function of the episode looks like:



Q learning

Let's see now what the average reward as a function of the episode looks like:



There is something now, but it doesn't seem to be doing too well.

Why do you think that's the case?

The problem is still that our algorithm will bias its policy to how the initial *Q* values were initialized.

The initial random *Q* values could lead to poor **exploration** of the state/action space, causing us to never learn the benefit of certain actions or states, because we rarely get there.

Let's cook ke world:

```
Q = np.zeros(
rewards    =
iterations =

# Parameters
alpha = 0.75
discount = 0.95
episodes = 5000

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

Initialize Q values to all zeros
(as many of them as number of States X Actions)

Pick the "best" action according to current Q estimates

The problem is still that our algorithm will bias its policy to how the initial Q values were initialized.

The initial random Q values could lead to poor **exploration** of the state/action space, causing us to never learn the benefit of certain actions or states, because we rarely get there.

How to fix that?

Let's cook ke world:

```
Q = np.zeros(
rewards    =
iterations =

# Parameters
alpha = 0.75
discount = 0.
episodes = 50

# Episodes
for episode in xrange(episodes):
    # Refresh state
    state         = env.reset()
    done          = False
    total_reward  = 0
    max_steps     = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :])
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

(as many of them as number of States X Actions)

Pick the "best" action according to current Q estimates

Add **randomization** (noise) during earlier actions, encouraging the agent to explore the state/action space early without committing to single actions for each state, but reduce this randomization (noise) as the agent learns more about the world.

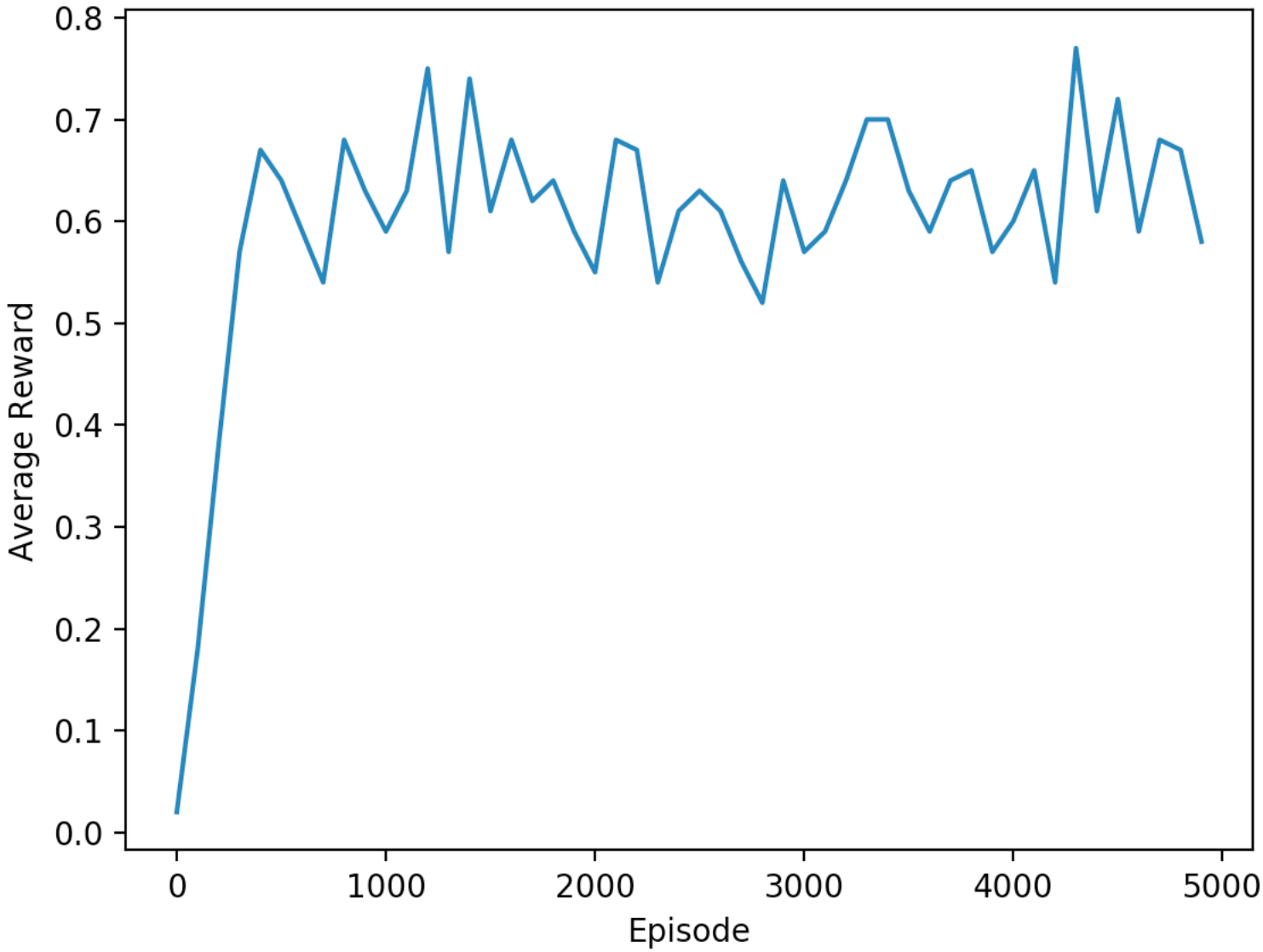
This ensures that the agent starts to behave near-optimally when it has explored the state/action space sufficiently well

Let's cook ke world:

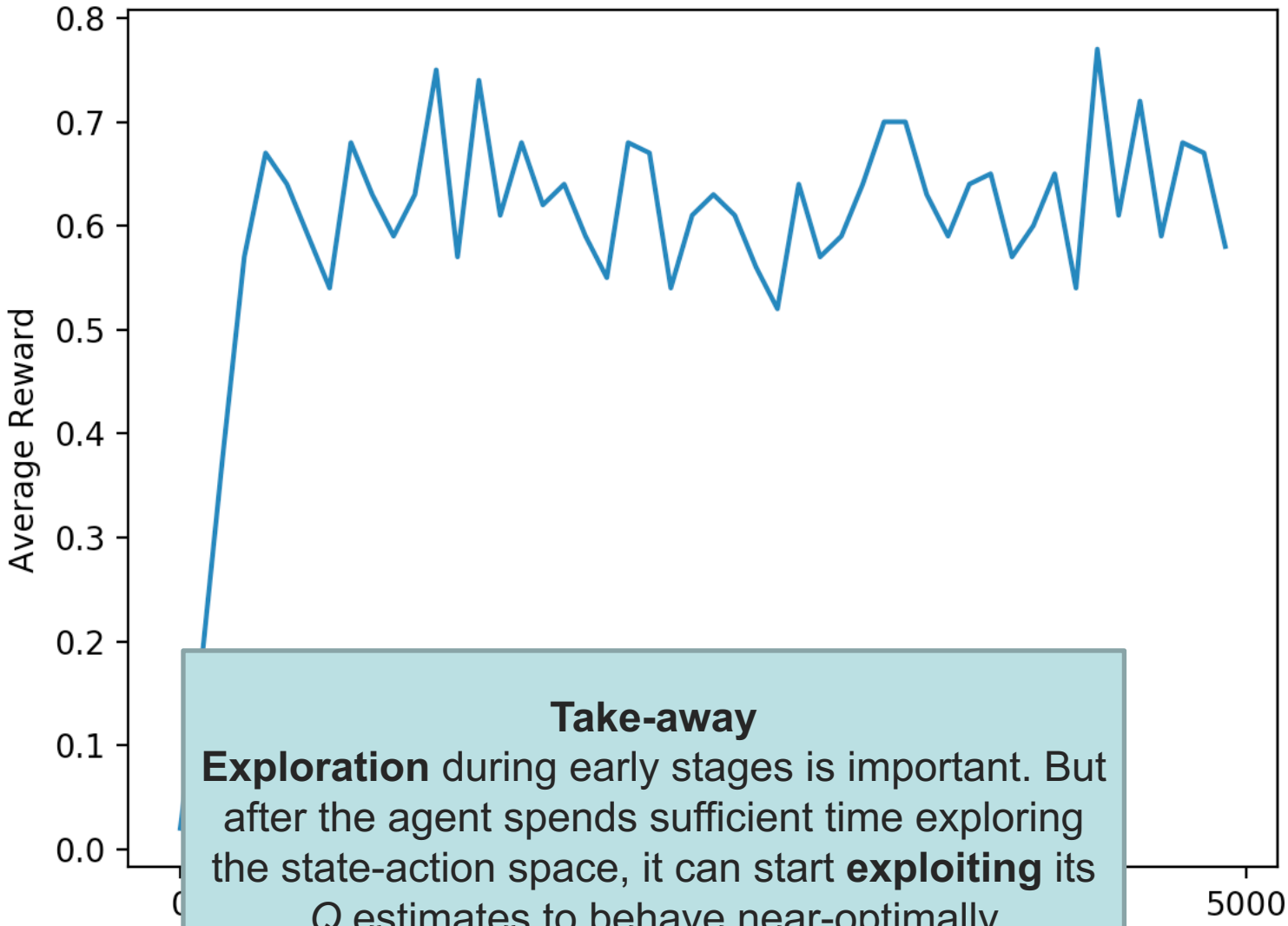
```
# Episodes
for episode in xrange(epi
    # Refresh state
    state          = env.reset()
    done           = False
    total_reward   = 0
    max_steps      = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')

    # Run episode
    for i in xrange(max_steps):
        if done:
            break

        current = state
        action  = np.argmax(Q[current, :] + np.random.randn(1, env.action_space.n) / float(episode + 1))
        state, reward, done, info = env.step(action)
        total_reward += reward
        Q[current, action] = (1.0-alpha) *  Q[current,action] + alpha * (reward + discount * np.max(Q[state, :]))

    rewards.append(total_reward)
    iterations.append(i)
```

Add some noise to every action, with more noise in earlier actions, and less noise in later actions

Q learning


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Average Reward
0
1000
2000
3000
4000
5000
Episode

Q learning



Take-away

Exploration during early stages is important. But after the agent spends sufficient time exploring the state-action space, it can start **exploiting** its *Q* estimates to behave near-optimally

Q learning

Possible Issues

Q learning

Q learning

Q learning

Q learning

Q learning

Possible Issues

State space can be huge!

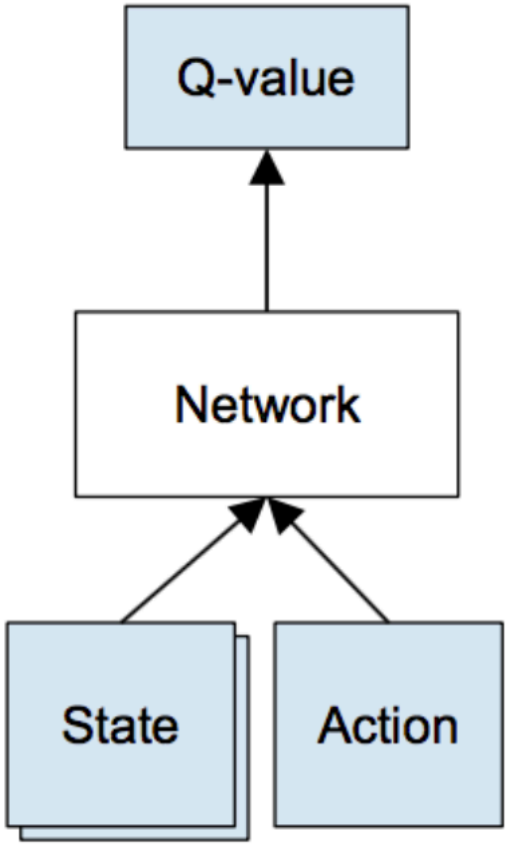
We should be able to generalize to states we've never seen if we have visited similar states in the past (e.g., in slightly different views of the same road or slightly different roads, a self-driving car should still know how to act)

Q learning

Possible Issues

State space can be huge!

We should be able to generalize to states we've never seen if we have visited similar states in the past (e.g., in slightly different views of the same road or slightly different roads, a self-driving car should still know how to act)



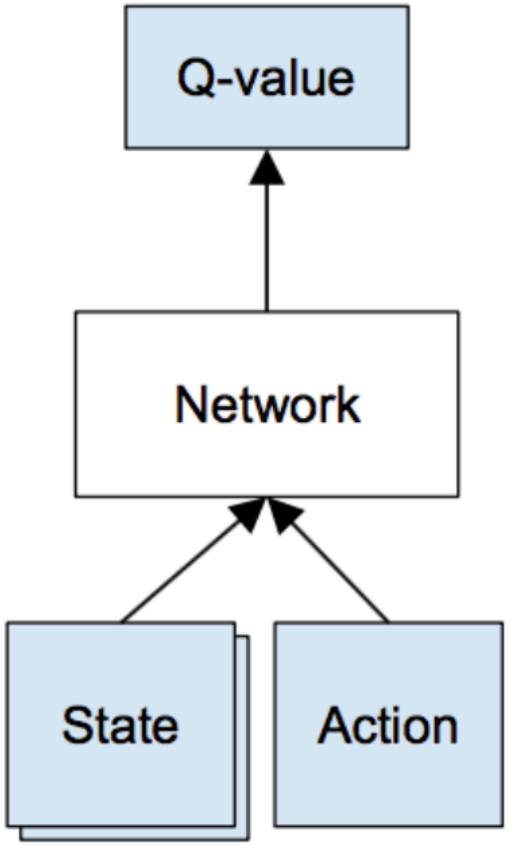
Q learning

Possible Issues

State space can be huge!

We should be able to generalize to states we've never seen if we have visited similar states in the past (e.g., in slightly different views of the same road or slightly different roads, a self-driving car should still know how to act)

Learn a function from states to Q-values directly!



Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

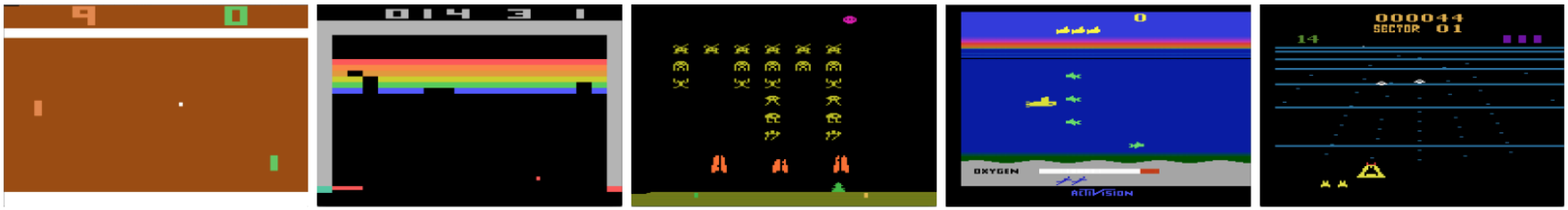
DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

Playing Atari with Deep Reinforcement Learning



Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

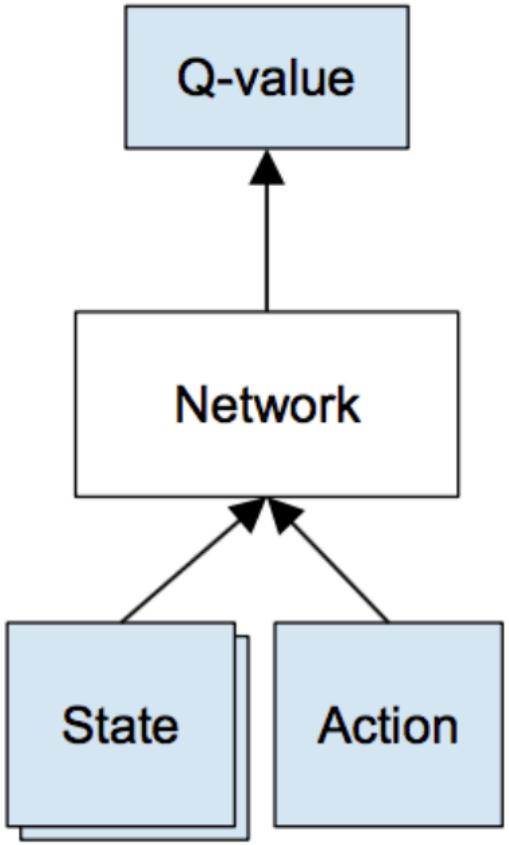
Q learning

Possible Problems

State space can be huge!

States are often not independent

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18



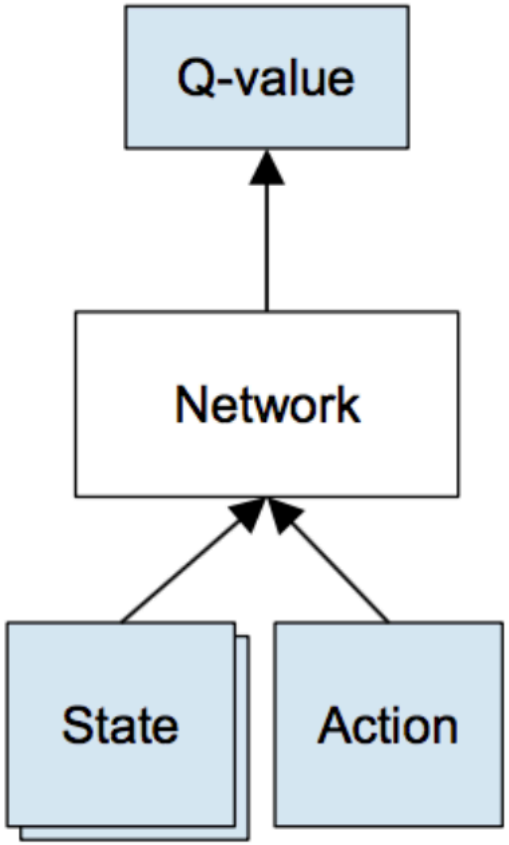
Q learning

Possible Problems

State space can be huge!

States are often not independent

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18



$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

Loss Function

Deep Q learning

1. Do a feedforward pass for the current state s to get predicted Q-values for all actions.
2. Do a feedforward pass for the next state s' and calculate maximum overall network outputs $max_{a'} Q(s', a')$.
3. Set Q-value target for action to $r + \gamma max_{a'} Q(s', a')$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation.

Sources:

Thanks to Aarti Singh for several slides

Guest Post (Part I): Demystifying Deep Reinforcement Learning
(https://www.intelnervana.com/demystifying-deep-reinforcement-learning/)

https://www.oreilly.com/learning/introduction-to-reinforcement-learning-and-openai-gym