Welcome!
#pod-031
Week #2, Day 5

(Reviewed by: Deepak)

neuromatch academy

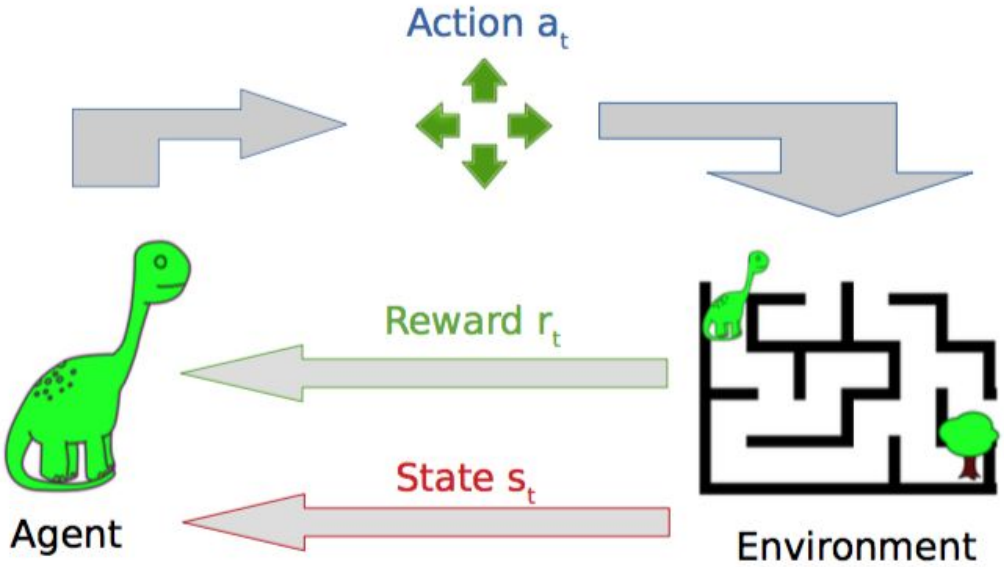
facebook Reality Labs

Agenda

- Day-#10
  - Tutorial part 1 (Temporal difference learning)
    - 1 Exercise + 1 bonus
  - Tutorial part 2 (Multi armed bandits)
    - 2 Exercises
  - Tutorial part 3 (Q learning)
    - 2 Exercises
  - Tutorial part 4 (Dyna Q)
    - 2 Exercises

Reinforcement Learning - Basic idea

Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward.



Tutorial #1 Explanations

Goals

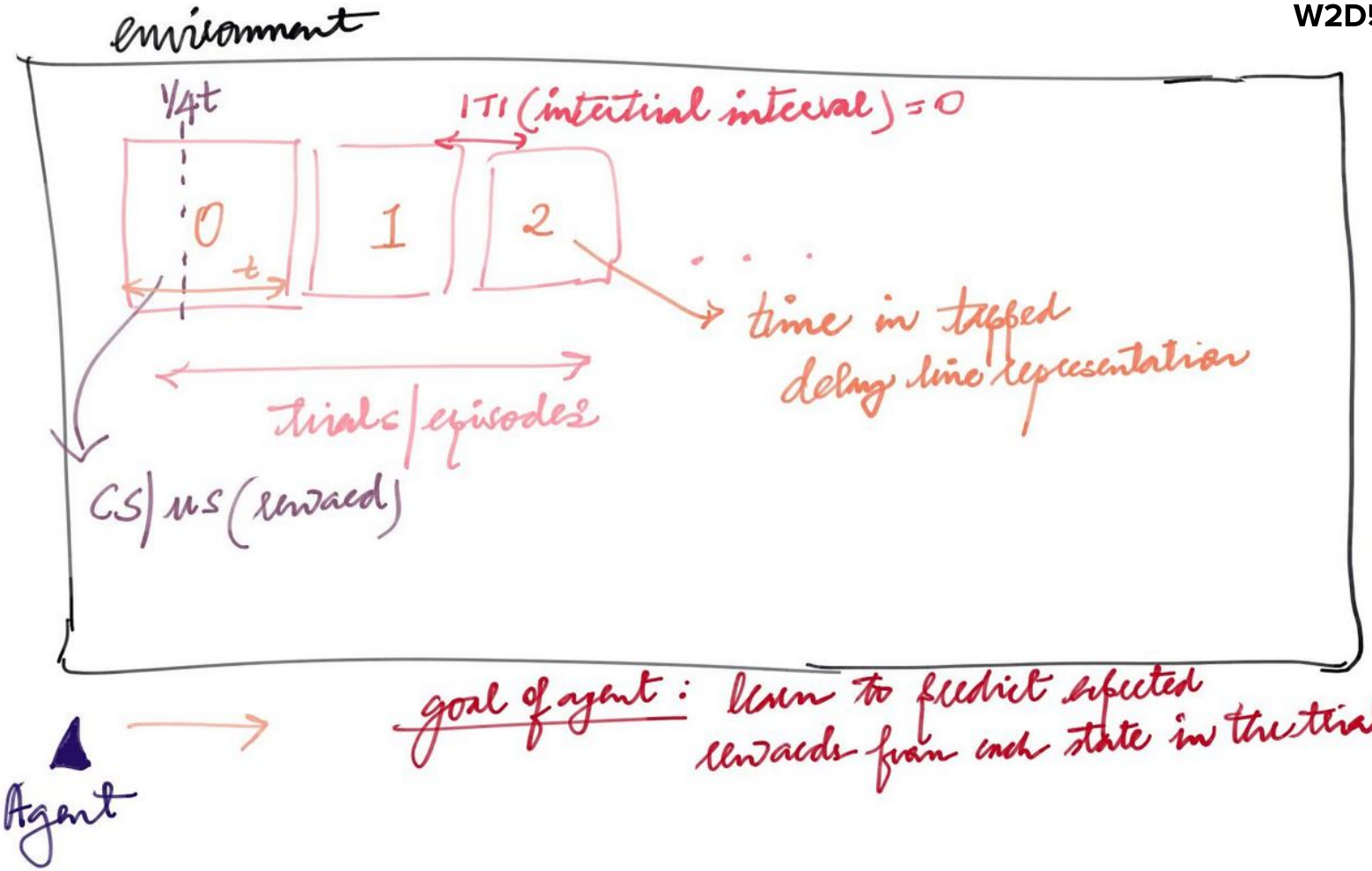
- Understand the use of standard tapped delay line conditioning model
- Understand how reward prediction errors move to conditioned stimulus/classical conditioning
- Understand how variability in reward size effects reward prediction errors
- Understand how differences in conditioned/unconditioned stimulus timing effect reward prediction errors
- Estimate state-value functions in a classical conditioning paradigm using Temporal Difference (TD) learning
- Examine Temporal Difference-errors at the presentation of the conditioned and unconditioned stimulus under different contingencies.
- Expectation to see if Dopamine represents a "canonical" reward prediction error

Temporal Difference learning

TD learning is an unsupervised technique in which the learning agent learns to predict the expected value of a variable occurring at the end of a sequence of states.

Experimental setup

- The agent experiences the environment in episodes or trials.
- Episodes terminate by transitioning to the inter-trial-interval (ITI) state and they are initiated from the ITI state as well. We clamp the value of the terminal/ITI states to zero.
- The classical conditioning environment is composed of a sequence of states that the agent deterministically transitions through. Starting at State 0, the agent moves to State 1 in the first step, from State 1 to State 2 in the second, and so on. These states represent time in the tapped delay line representation
- Within each episode, the agent is presented a CS and US (reward).
- The CS is always presented at 1/4 of the total duration of the trial. The US (reward) is then delivered after the CS. The interval between the CS and US is specified by reward_time.
- The agent's goal is to learn to predict expected rewards from each state in the trial.


environment
1/4t
ITI (intertrial interval) = 0
0
1
2
t
. . .
time in tapped delay line representation
trials / episodes
CS / US (reward)
goal of agent: learn to predict expected rewards from each state in the trial.
Agent

G_t (future cumulative reward)

$$= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} = r_{t+1} + \gamma G_{t+1}$$

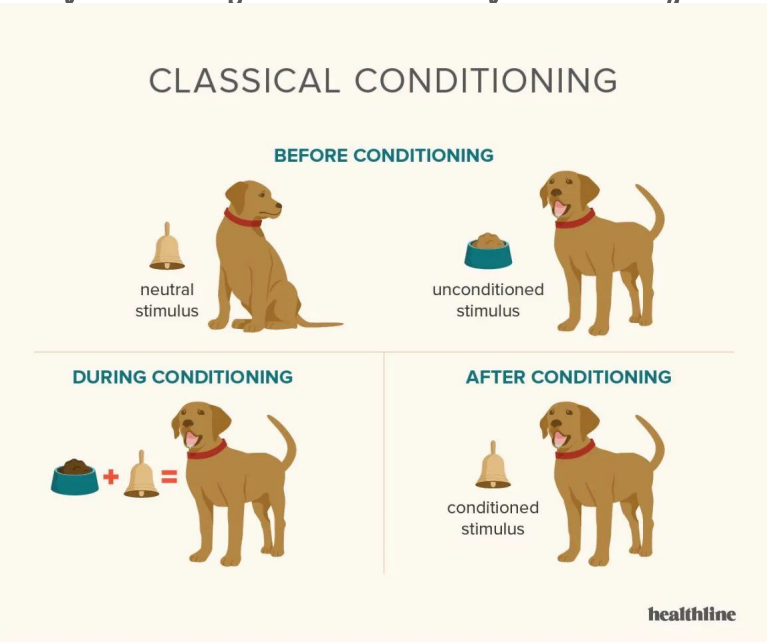
discount factor $\in [0, 1]$

(controls importance of future rewards)

$\equiv$ probability of continuing the trajectory.

TD learning in classical conditioning setting

TD-learning to estimate the state-value function in the classical-conditioning world with guaranteed rewards, with a fixed magnitude, at a fixed delay



Reward prediction errors

Reward prediction errors play an important role in this process by helping adjust the value of states.

Discount factor

Tells how important future rewards are to the current state.
Discount factor is a value between 0 and 1.

A reward R that occurs N steps in the future from the current state, is multiplied by γ^N to describe its importance to the current state

Changing Environment!

Replace the environment with one that dispenses multiple rewards.

Single reward delivered intermittently

This simple model closely resembles the behavior of subjects undergoing classical conditioning tasks and the dopamine neurons that may underlie that behavior.

Value function $V_\pi(s_t = s)$

expectation of return : $E\left[G_t \middle| s_t = s,\ a_{t:\infty} \sim \pi\right]$

$$= E\left[r_{t+1} + \gamma G_{t+1} \middle| s_t = s,\ a_{t:\infty} \sim \pi\right]$$

assuming markov process -

$$V_\pi(s_t = s) = E\left[r_{t+1} + \gamma V_\pi(s_{t+1}) \middle| s_t = s,\ a_{t:\infty} \sim \pi\right]$$

$$= \sum_a \pi(a|s) \sum_{r,s'} p(s', r) \left(r + V_\pi\left(s_{t+1} = s'\right)\right).$$

temporal difference (TD) learning

- markov assumption: $V(s_{t+1})$ as imperfect proxy for the true value G_{t+1} (monte carlo bootstrapping)

generalised equation for TD error:

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

value updated using learning rate constant α

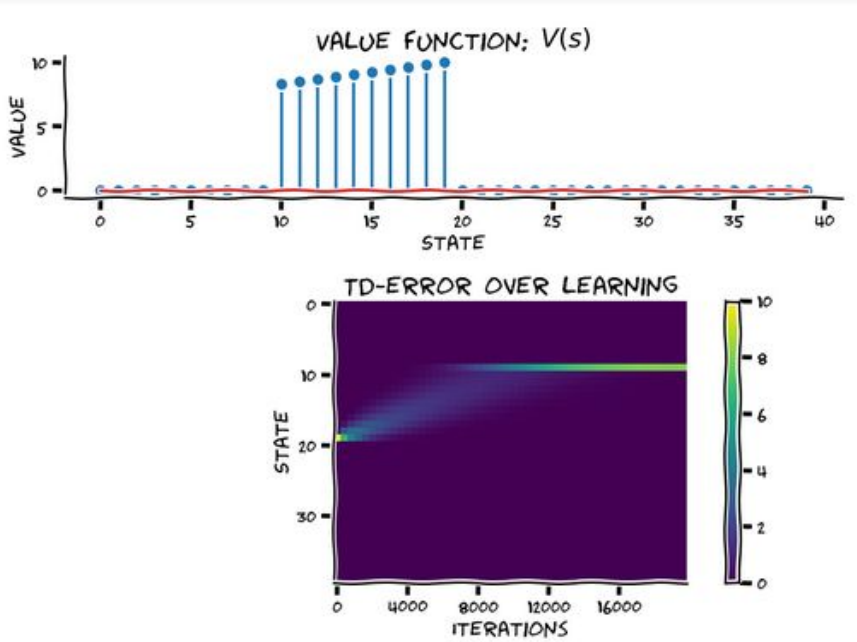
$$V(s_t) \leftarrow V(s_t) + \alpha \delta_t$$

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

value updates: $V(s_t) \leftarrow V(s_t) + \alpha \delta_t$

TD learning with guaranteed rewards

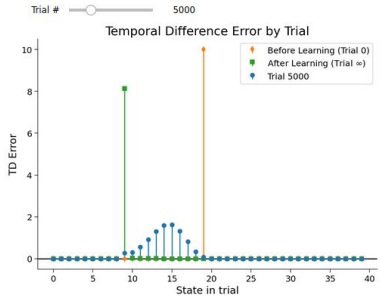
TD-learning is used to estimate the state-value function in classical-conditioning world with guaranteed rewards, with a fixed magnitude, at a fixed delay after the conditioned stimulus (effect simulated by updating $V(s)$ during the delay period after stimulus).

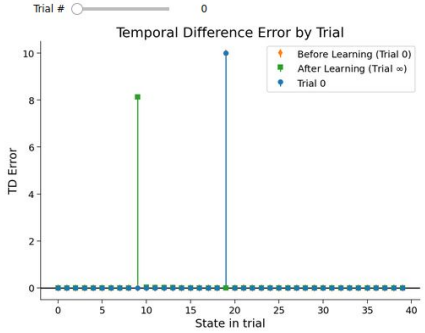


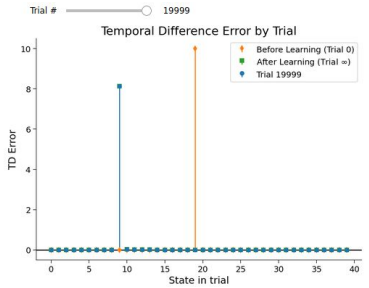
How reward prediction errors change over time?

During classical conditioning, the subject's behavioral response (e.g., salivating) transfers from the unconditioned stimulus (like the smell of tasty food) to the conditioned stimulus (like Pavlov ringing his bell) that predicts it. Reward prediction errors play an important role in this process by adjusting the value of states according to their expected, discounted return.

Observations: Before training (orange line), only the reward state has high reward prediction error. As training progresses (blue line), the reward prediction errors shift to the conditioned stimulus, where they end up when the trial is complete (green line). Dopamine neurons, which are thought to carry reward prediction errors *in vivo*, show exactly the same behavior!

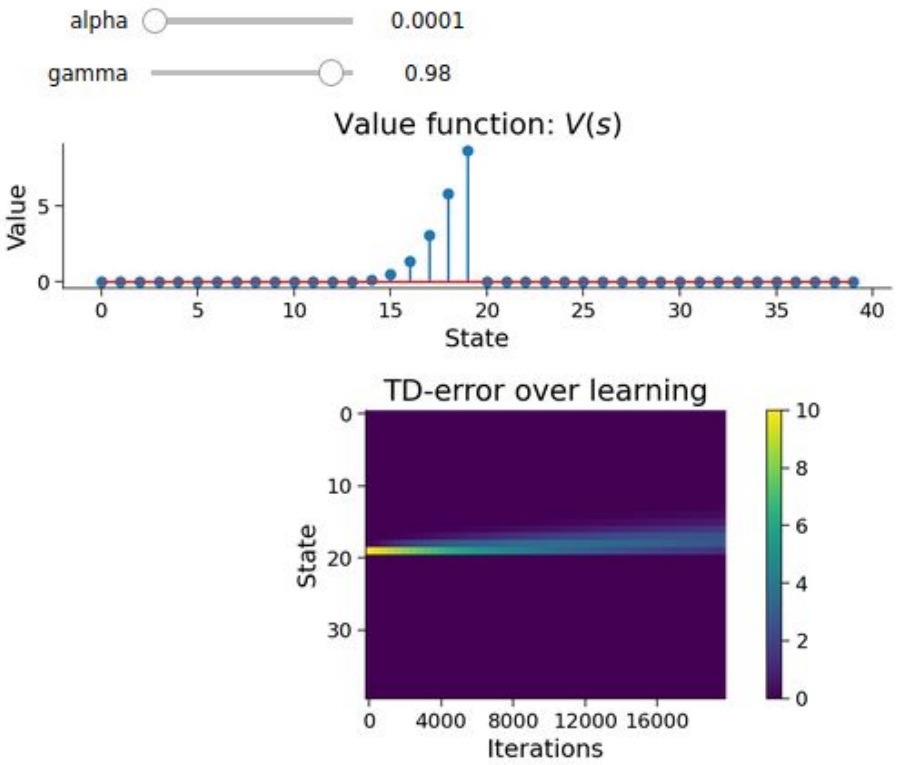






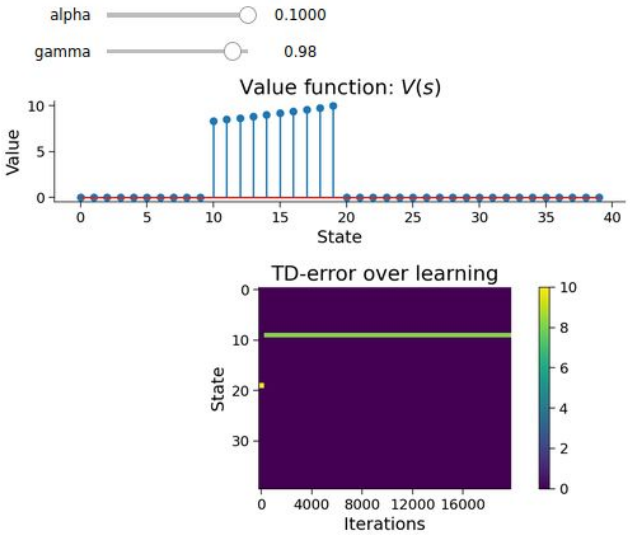
Learning Rates and Discount Factors

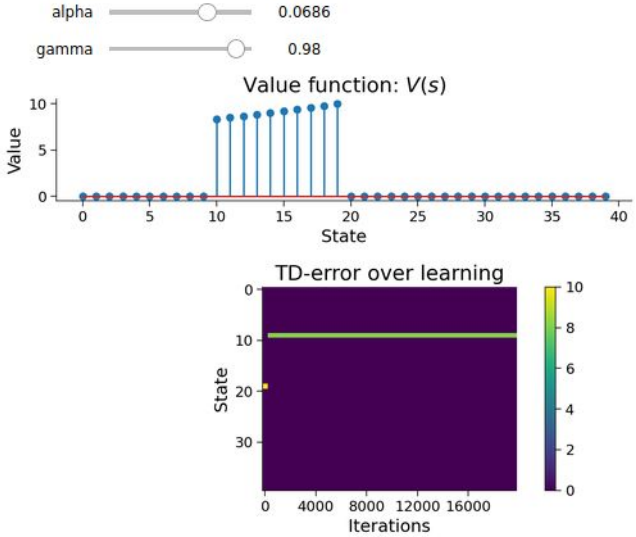
TD-learning agent has two parameters that control how it learns: α, the learning rate, and γ, the discount factor which alter the model that TD-learning learns.

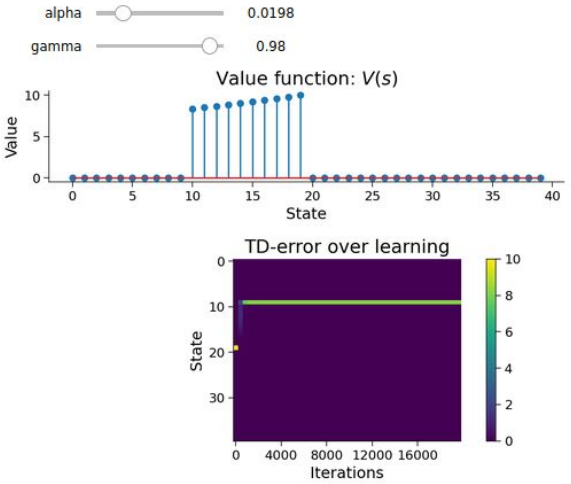


Learning rates

α controls the size of the Value function updates produced by each TD-error. In our simple, deterministic world, will this affect the final model we learn? Is a larger α necessarily better in more complex, realistic environments?

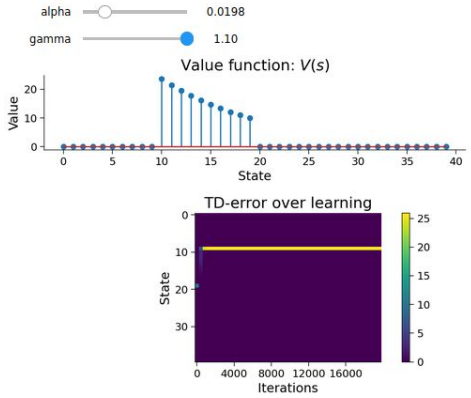


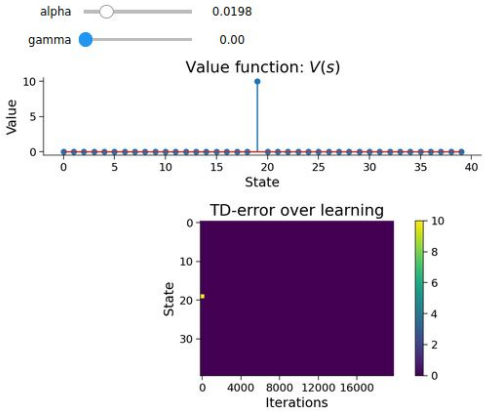


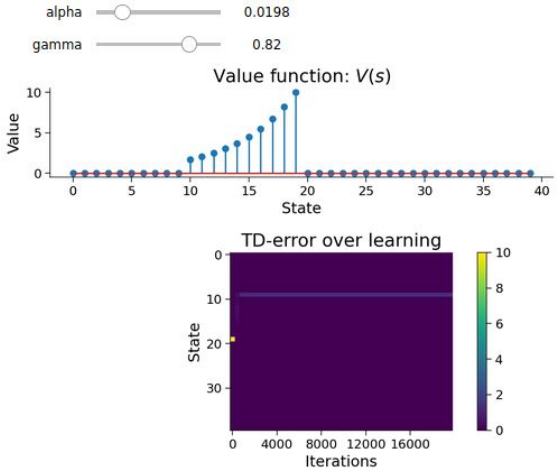


Discount factors

The discount rate γ applies an exponentially-decaying weight to returns occuring in the future, rather than the present timestep. How does this affect the model we learn? What happens when $\gamma = 0$ or $\gamma \geq 1$?







Observations

Alpha determines how fast the model learns. In the simple, deterministic world, this allows the model to quickly converge onto the "true" model that heavily values the conditioned stimulus. In more complex environments, however, excessively large values of alpha can slow, or even prevent, learning.

Gamma effectively controls how much the model cares about the future: larger values of gamma cause the model to weight future rewards nearly as much as present ones. At gamma=1, the model weights all rewards, regardless of when they occur, equally and when greater than one, it starts to *prefer* rewards in the future, rather than the present (this is rarely good). When gamma=0, however, the model becomes greedy and only considers rewards that can be obtained immediately.

Matching Value functions

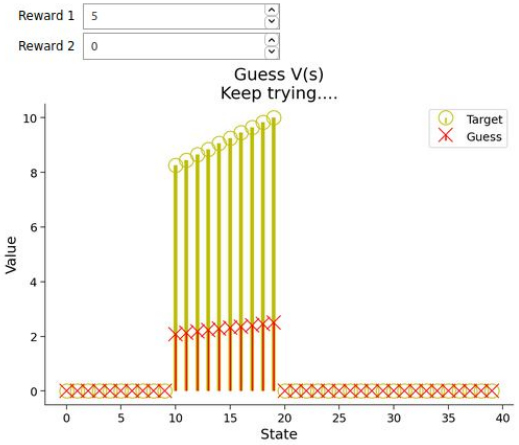
Replace the environment with one that dispenses one of several rewards, chosen at random.

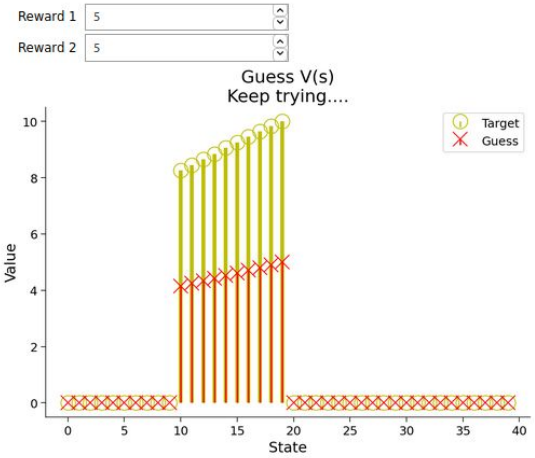
Task: find another pair of rewards that cause the agent to learn the same value function. Final value function V for a TD learner that was trained in an environment where the CS predicted a reward of 6 or 14 units; both rewards were equally likely.

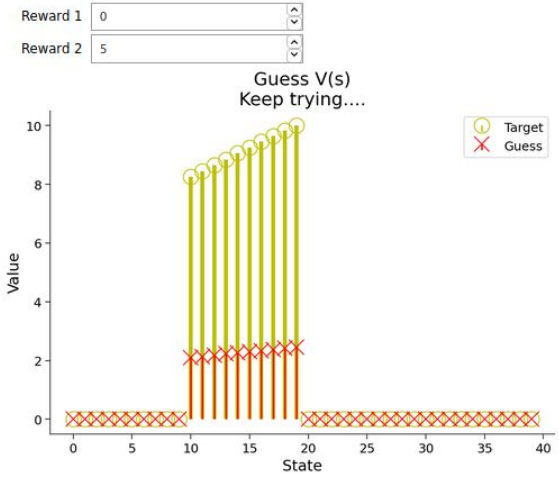
Assume each reward will be dispensed 50% of the time.

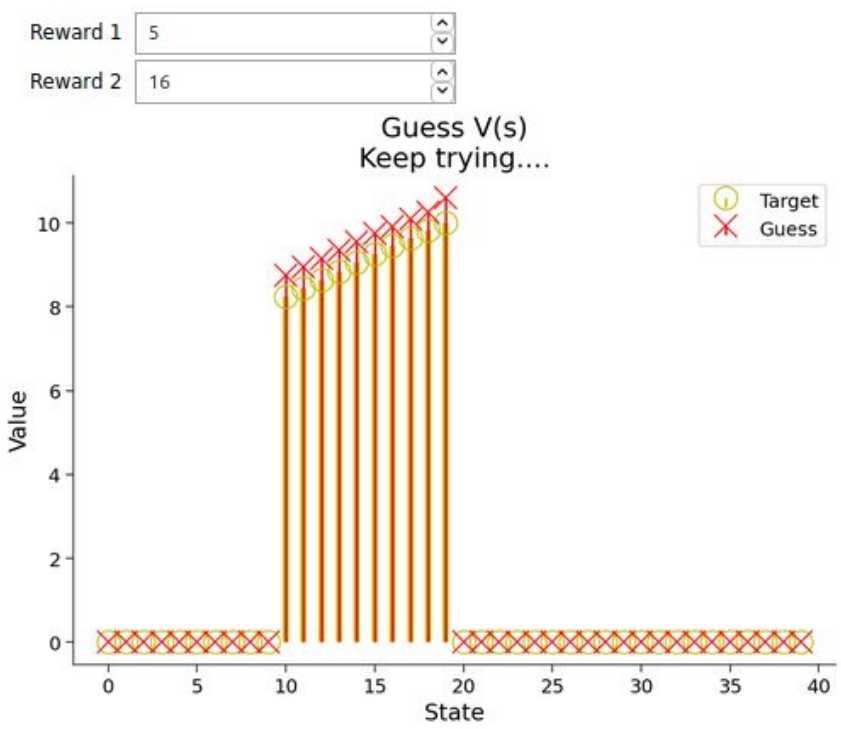
Hints:

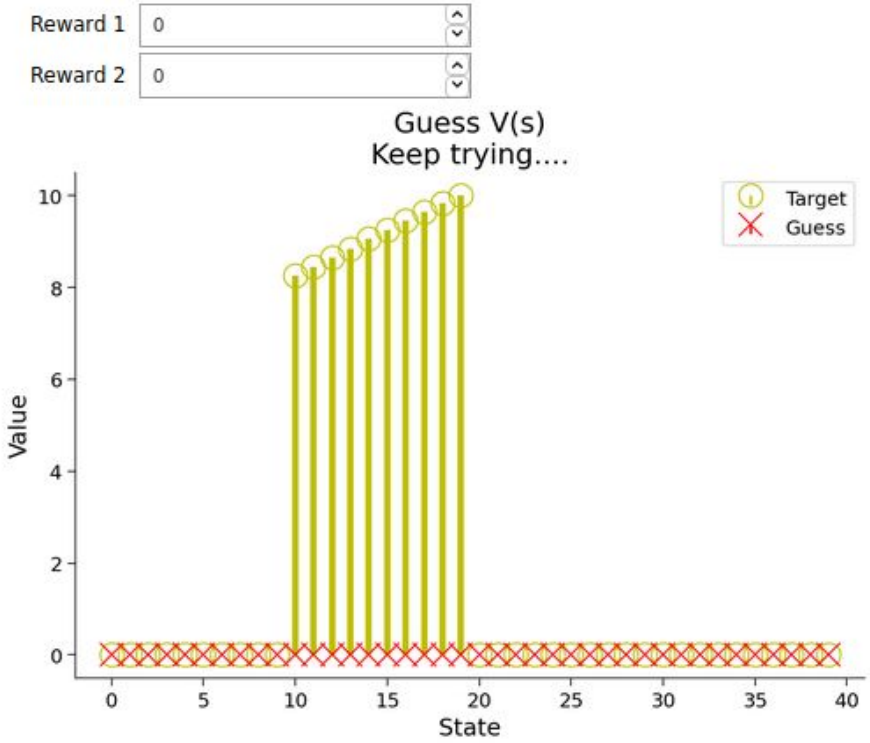
- Carefully consider the definition of the value function V. This can be solved analytically.
- There is no need to change α or γ.
- Due to the randomness, there may be a small amount of variation.


Reward 1 5
Reward 2 0
Guess V(s)
Keep trying....
Target
Guess
Value
State


Reward 1 5
Reward 2 5
Guess V(s)
Keep trying....
Target
Guess
Value
State


Reward 1 0
Reward 2 5
Guess V(s)
Keep trying....
Target
Guess
Value
State

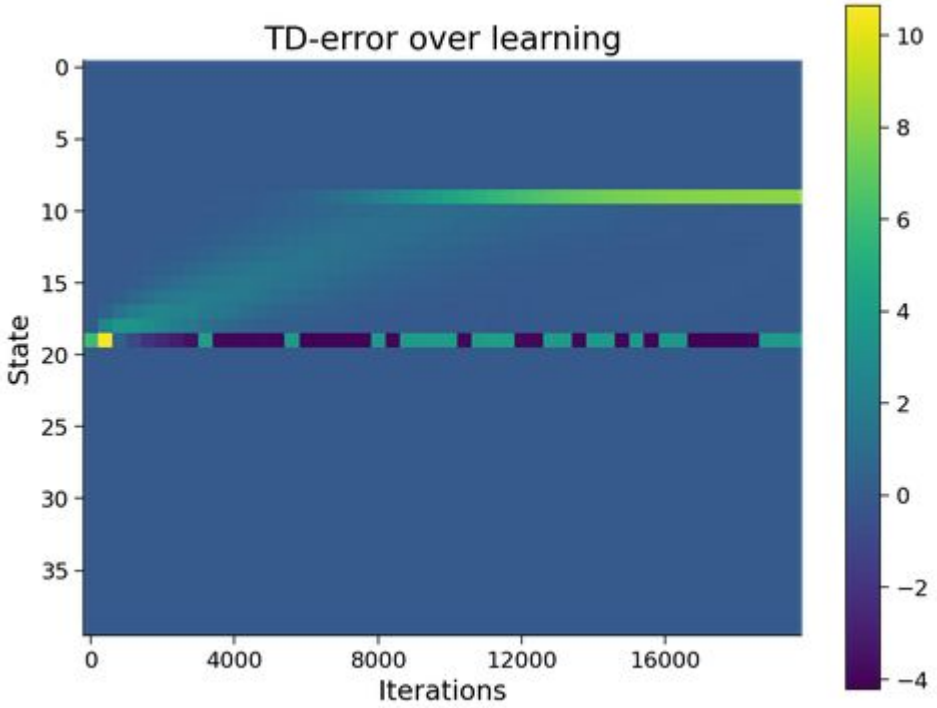

Reward 1
5
Reward 2
16
Guess V(s)
Keep trying....
Target
Guess
Value
State
0
2
4
6
8
10
0
5
10
15
20
25
30
35
40


Reward 1
0
Reward 2
0
Guess V(s)
Keep trying....
Target
Guess
Value
State
0
2
4
6
8
10
0
5
10
15
20
25
30
35
40

Examine TD error

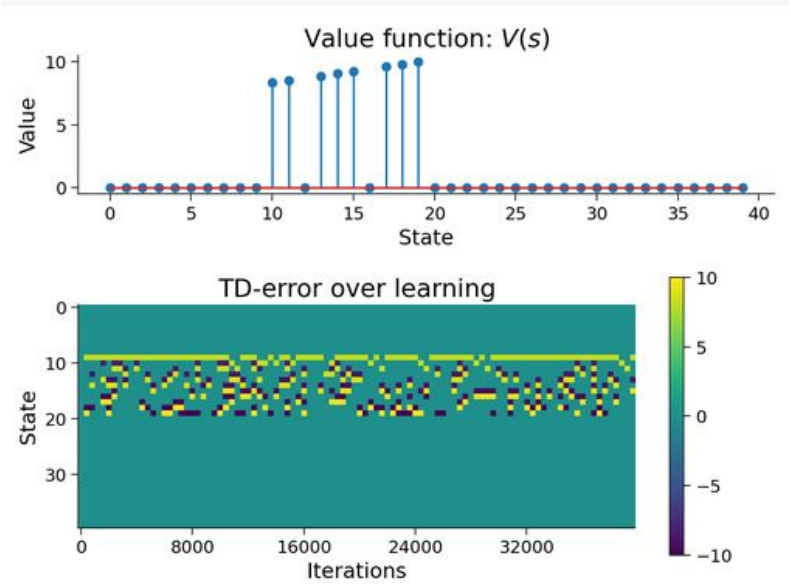
Plot of TD errors from our multi-reward environment.

The TD trace now takes on negative values because the reward delivered is sometimes larger than the expected reward and sometimes smaller.



TD-learning with probabilistic rewards

In this environment, we'll deliver a single reward of ten units. However, it will be delivered intermittently: on 20 percent of trials, the conditioned stimulus will be shown but the agent will not receive the usual reward; the remaining 80% will proceed as usual.



Observations

The multi-reward and probabilistic reward environments are the same. You could simulate a probabilistic reward of 10 units, delivered 50% of the time, by having a mixture of 10 and 0 unit rewards, or vice versa. The *average* or expected reward is what matters during TD Learning.

Large values of alpha prevent the TD Learner from converging. As a result, the value function seems implausible: one state may have an extremely low value while the neighboring ones remain high. This pattern persists even if training continues for hundreds of thousands of trials.

Summary

By manipulating the environment of a simple TD Learner and examining the evolution of its state representations and reward prediction errors during training along with its parameters (α, γ), one can develop an intuition for how it behaves.

This simple model closely resembles the behavior of subjects undergoing classical conditioning tasks and the dopamine neurons that may underlie that behavior. The update rule used here has been extensively studied for more than 70 years as a possible explanation for artificial and biological learning.

Further, we use the carefully calculated value of each state to plan Actions!

Tutorial #1 Bonus Explanations

Effect of removing V[state]

You should only be updating the V[state] once the conditioned stimulus appears.

If you remove this term the Value Function becomes periodic, dropping towards zero right after the reward and gradually rising towards the end of the trial. This behavior is actually correct, because the model is learning the time until the *next* reward, and latter States are closer to a reward than earlier ones. In an actual experiment, the animal often just wants rewards; it doesn't care about experiment or trial structure!

Tutorial #2
Explanations

Objective

Use 'bandits' to understand the fundamentals of how a policy interacts with the learning algorithm in reinforcement learning.

Understand the fundamental trade-off between exploration and exploitation in a policy.

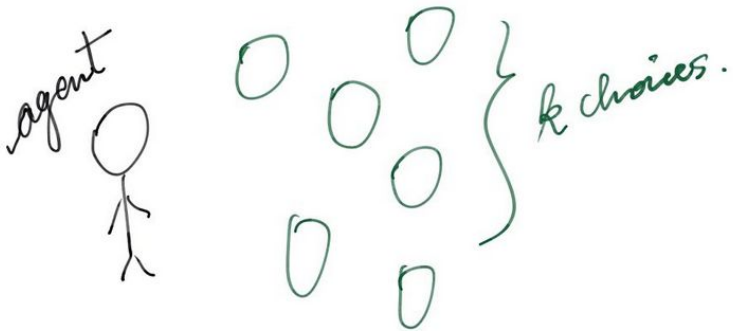
Understand how the learning rate interacts with exploration to find the best available action.

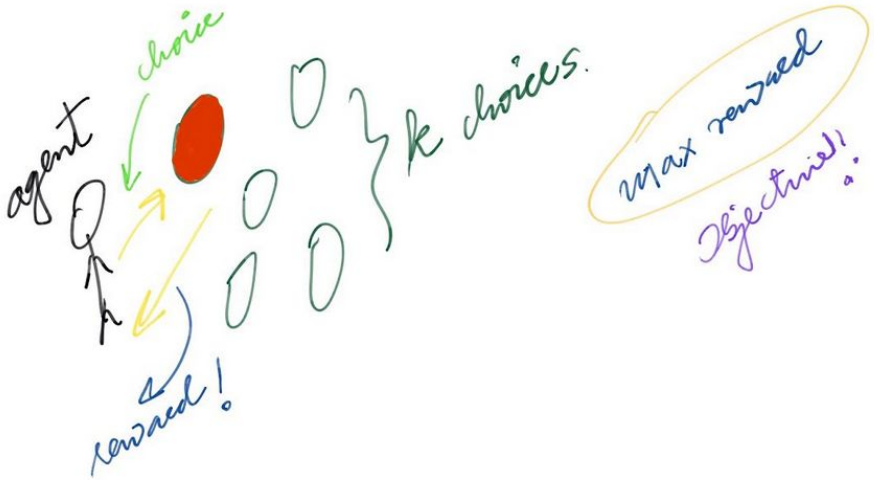
Multi arm bandit problem

In the multi-armed bandit problem, at each stage, an agent (or decision maker) chooses one action (or arm) from a set of k different actions, and receives a reward from it. The agent aims at maximizing his expected total reward for some time steps/action selections.

While there are many different levels of sophistication and assumptions in how the rewards are determined, for simplicity's sake we will assume that each action results in a reward drawn from a fixed Gaussian distribution with unknown mean and unit variance.
This problem setting is referred to as the environment, and goal is to find the arm with the highest mean value.

K-armed bandit problem





Policy

The strategy for choosing actions based on our expectations is called a policy.

Types for choices!

1. **Random!**

   We could have a random policy -- just pick an arm at random each time
   Issue? Unlikely to be an optimal way of optimising reward

2. **Greedy**

   action with the highest expected value.
   Issue? It easily gets trapped in local maxima. It never explores to see what it hasn't seen before if one option is already better than the others.

Exploration vs exploitation

Exploitation consists of probing a limited (but promising) region of the search space with the hope of improving a promising solution S that we already have at hand. This operation amounts then to intensifying (refining) the search in the vicinity of S. By doing so, we would be doing, de facto, a local search.

Exploration, on the other hand, consists of probing a much larger portion of the search space with the hope of finding other promising solutions that are yet to be refined. This operation amounts then to diversifying the search in order to avoid getting trapped in a local optimum. By doing so, we would be doing, de facto, a global search.

Exploration Exploitation Dilemma

Do you go with you best choice now, or risk the less certain option with the hope of finding something better? Too much exploration, however, means you may end up with a sub-optimal reward once it's time to stop.

Avoiding local minima problem

A simple extension to our greedy policy is to add some randomness. For instance, a coin flip -- heads we take the best choice now, tails we pick one at random. This is referred to as the ϵ-greedy policy:

action value function $q(a) = \mathbb{E}[r_t \mid a_t = a]$

reward — action

{ reward given action } — expected value

decision beliefs about values each action should return

greedy policy: $a_t = \text{argmax}_a \, q_t(a)$

(choice action maximize current value function)

ε-greedy policy.

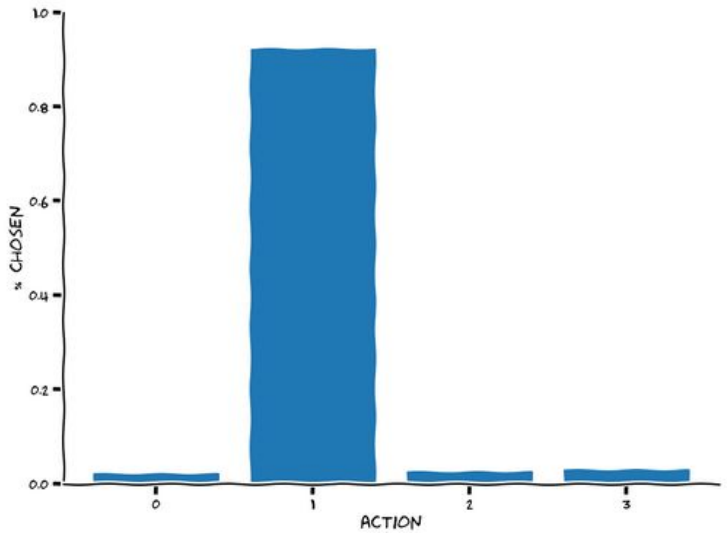
$$P(a_t = a) = \begin{cases} 1 - \varepsilon + \varepsilon/N & \text{if } a_t = \operatorname{argmax}_a Q_t(a) \\ \varepsilon/N & \text{else} \end{cases}$$

Probability $1-\varepsilon$ for $\varepsilon \in [0,1]$ $\Rightarrow$ select greedy choice

else: select an action at random

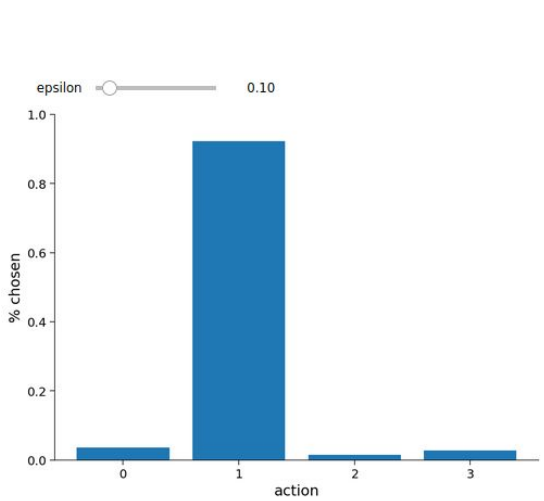
Observations

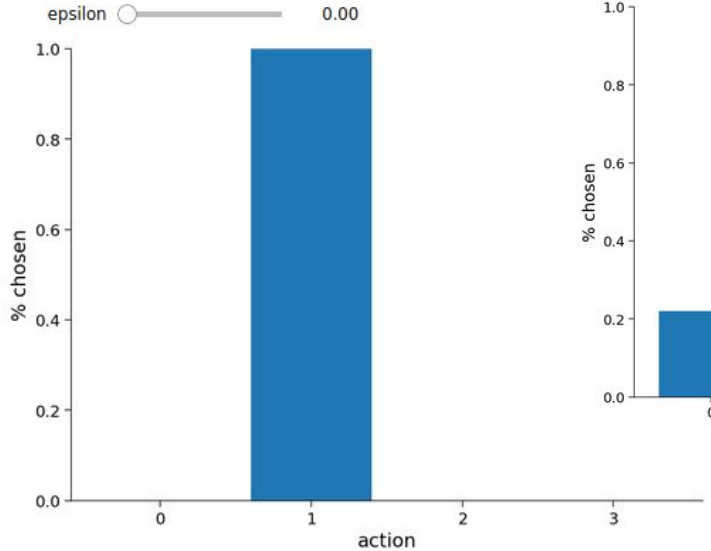
When epsilon is zero, the agent always chooses the currently best option; it becomes greedy. When epsilon is 1, the agent chooses randomly.

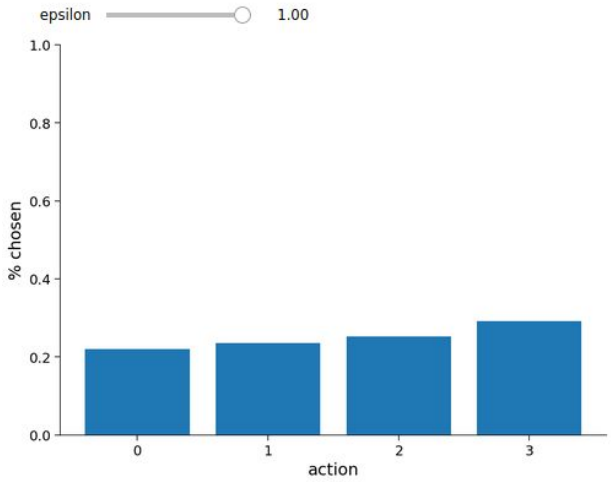


Changing ϵ affects the distribution of selected actions.

Epsilon is our one parameter for balancing exploitation and exploration.







learning from rewards

→ keep record of every result & use averages for each action.

Streaming mean calculation

$$q_{t+1}(a) \leftarrow q_t(a) + \frac{1}{n_t}\left(r_t - q_t(a)\right)$$

- r_t: reward for taking action a
- $q_{t+1}(a)$: action value function (mean of rewards seen so far)
- n_t: no. of actions taken at time t

requires remembering #actions

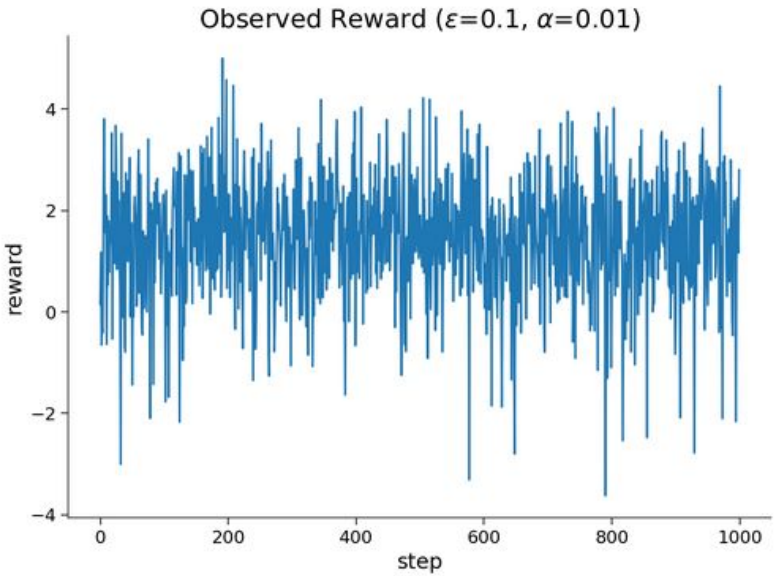
⇒ generalising:

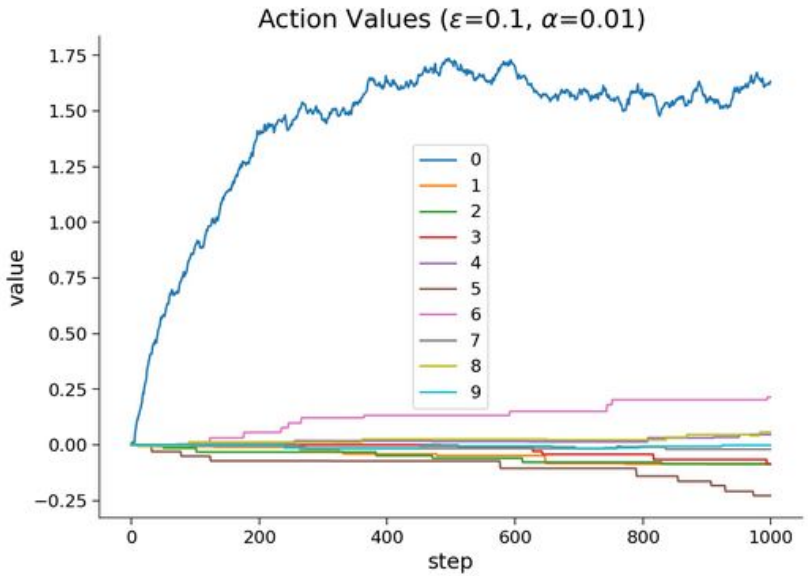
replace action Total with general parameter α (learning rate)

$$q_{t+1}(a) \leftarrow q_t(a) + \alpha\left(r_t - q_t(a)\right)$$

Action value update rule.

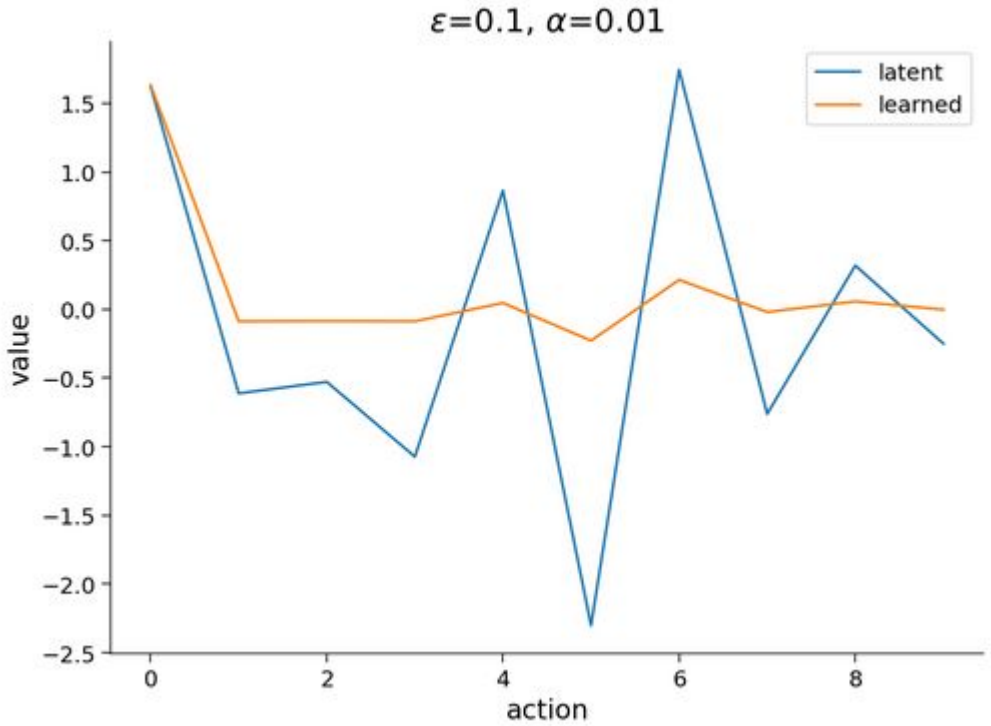
Use multi-armed bandit method to evaluate how epsilon-greedy policy and learning rule perform at solving the task.





Observations

we got some rewards that are kind of all over the place, but the agent seemed to settle in on the first arm as the preferred choice of action relatively quickly. Evaluate recovering true means of the Gaussian random variables behind the arms.



Observations

Very good estimate for action 0, but most of the others are not great.

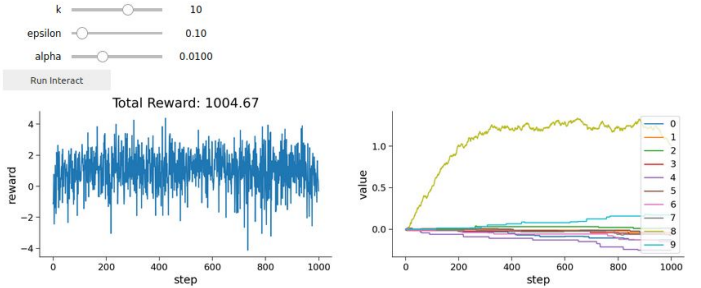
See the effect of the local maxima trap at work -- the greedy part of our algorithm locked onto action 0, which is actually the 2nd best choice to action 6. Since these are the means of Gaussian random variables, we see the overlap between the two would be quite high, so even if we did explore action 6, we may draw a sample that is still lower than our estimate for action 0.

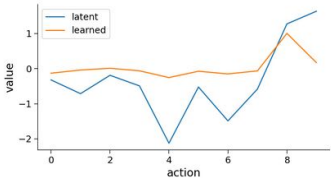
However, this was just one choice of parameters.

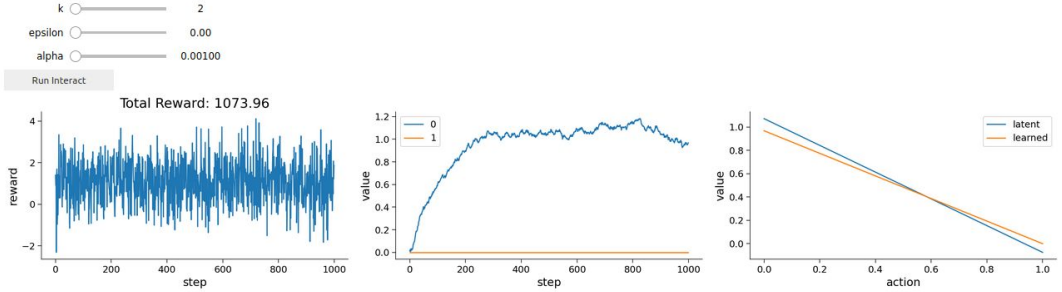
#TASK : Find better combinations

Changing Epsilon and Alpha

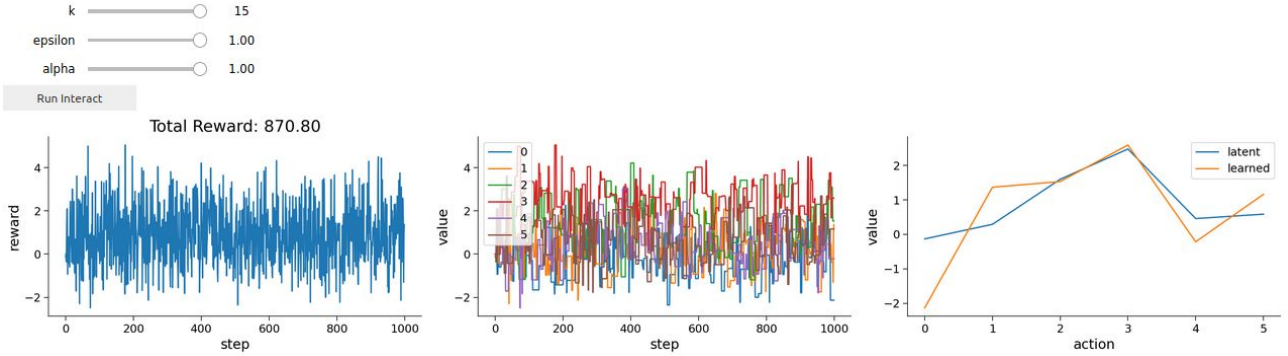
Explore how varying the values of ϵ (exploitation-exploration trade-off), α (learning rate), and even the number of actions k, changes the behavior of our agent.

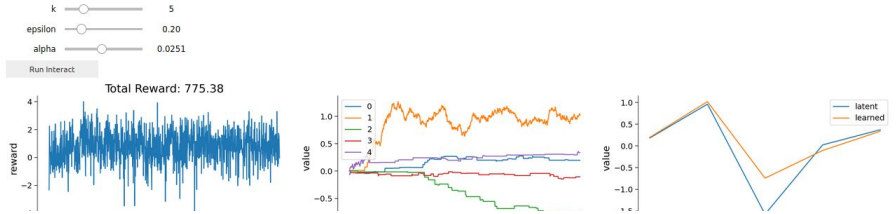


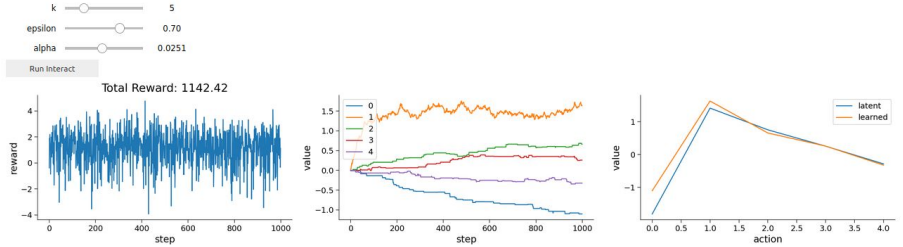


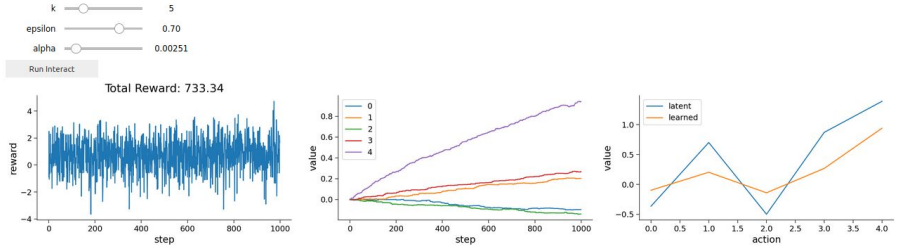


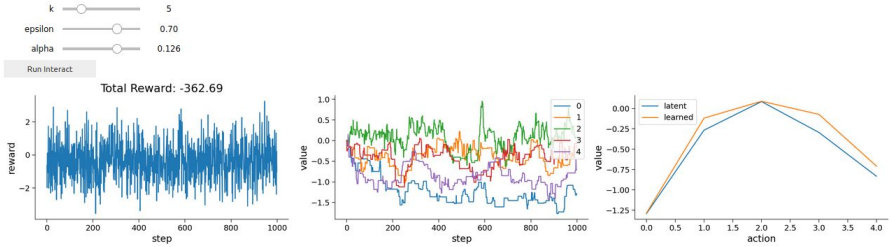
W2D5_pod 031


k 15
epsilon 1.00
alpha 1.00
Run Interact
Total Reward: 870.80
reward
step
value
step
latent
learned
value
action


k 5
epsilon 0.20
alpha 0.0251
Run Interact
Total Reward: 775.38
reward
value
latent
learned
value


k 5
epsilon 0.70
alpha 0.0251
Run Interact
Total Reward: 1142.42
reward
step
value
step
latent
learned
value
action


k 5
epsilon 0.70
alpha 0.00251
Run Interact
Total Reward: 733.34
reward
step
value
step
latent
learned
value
action


k 5
epsilon 0.70
alpha 0.126
Run Interact
Total Reward: -362.69
reward
step
value
step
latent
learned
value
action

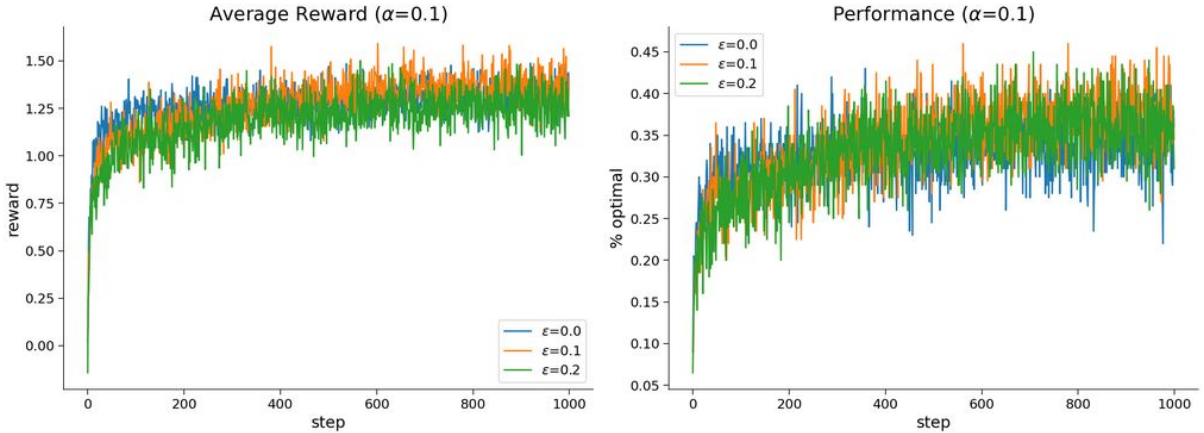
Observations

While we can see how changing the epsilon and alpha values impact the agent's behavior, this doesn't give as a great sense of which combination is optimal. Due to the stochastic nature of both our rewards and our policy, a single trial run isn't sufficient to give us this information. Run multiple trials and compare the average performance.

Observations

On the left we have plotted the average reward over time, and we see that while ϵ=0

(the greedy policy) does well initially, ϵ=0.1 starts to do slightly better in the long run, while ϵ=0.2 does the worst. Looking on the right, we see the percentage of times the optimal action (the best possible choice at time t) was taken, and here again we see a similar pattern of ϵ=0.1 starting out a bit slower but eventually having a slight edge in the longer run.

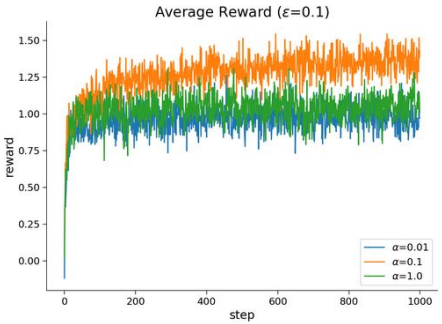


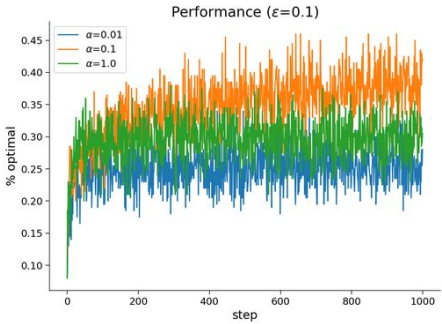
Observations

Again we see a balance between an effective learning rate.

α=0.01 is too weak to quickly incorporate good values, while

α=1 is too strong likely resulting in high variance in values due to the Gaussian nature of the rewards.





Take aways

By implementing both the epsilon-greedy decision algorithm and a learning rule for solving a multi-armed bandit scenario, we see how balancing exploitation and exploration in action selection is critical in finding optimal solutions. Choosing an appropriate learning rate determines how well an agent can generalize the information they receive from rewards.

Tutorial #3
Explanations

Objective

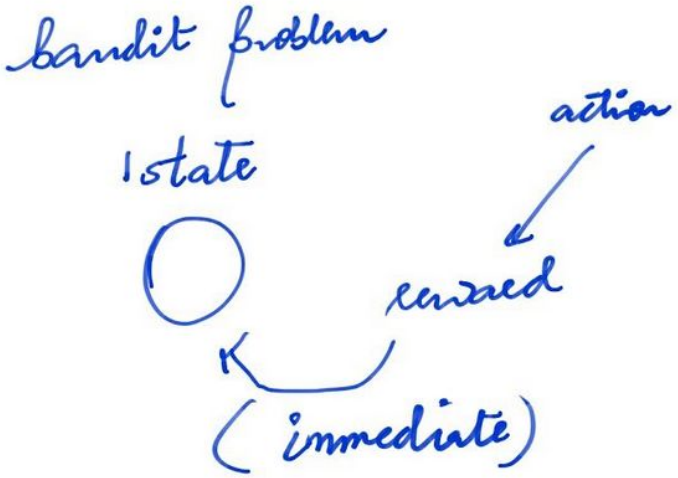
Learn how to act in the more realistic setting of sequential decisions, formalized by Markov Decision Processes (MDPs). In a sequential decision problem, the actions executed in one state not only may lead to immediate rewards (as in a bandit problem), but may also affect the states experienced next (unlike a bandit problem). Each individual action may therefore affect affect all future rewards. Thus, making decisions in this setting requires considering each action in terms of their expected **cumulative** future reward.

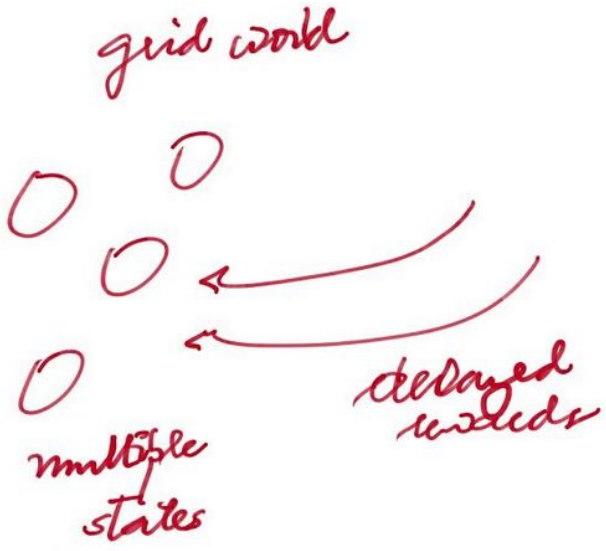
For instance like in spatial navigation, where actions (movements) in one state (location) affect the states experienced next, and an agent might need to execute a whole sequence of actions before a reward is obtained.

Learn grid worlds and how they help in evaluating simple reinforcement learning agents

Basics of the Q-learning algorithm for estimating action values

Concept of exploration and exploitation, reviewed in the bandit case, also applies to the sequential decision setting

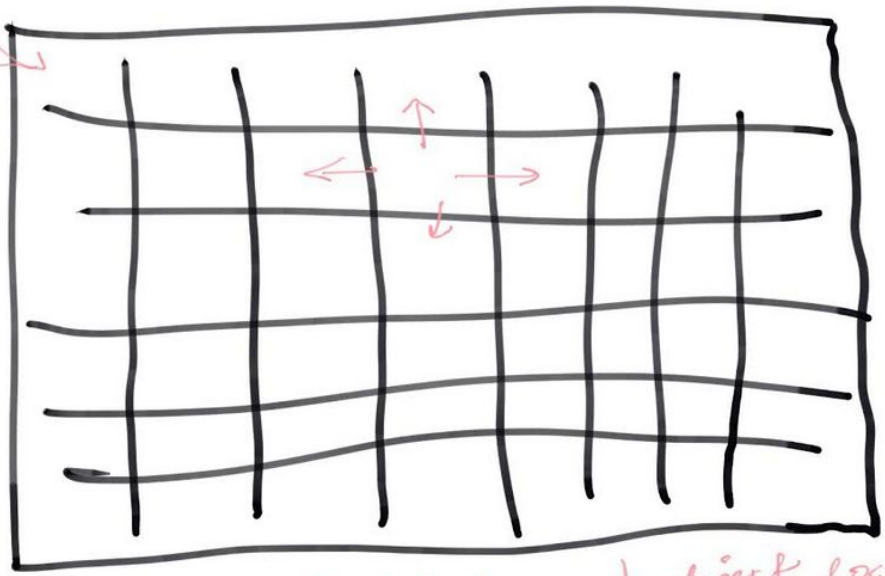




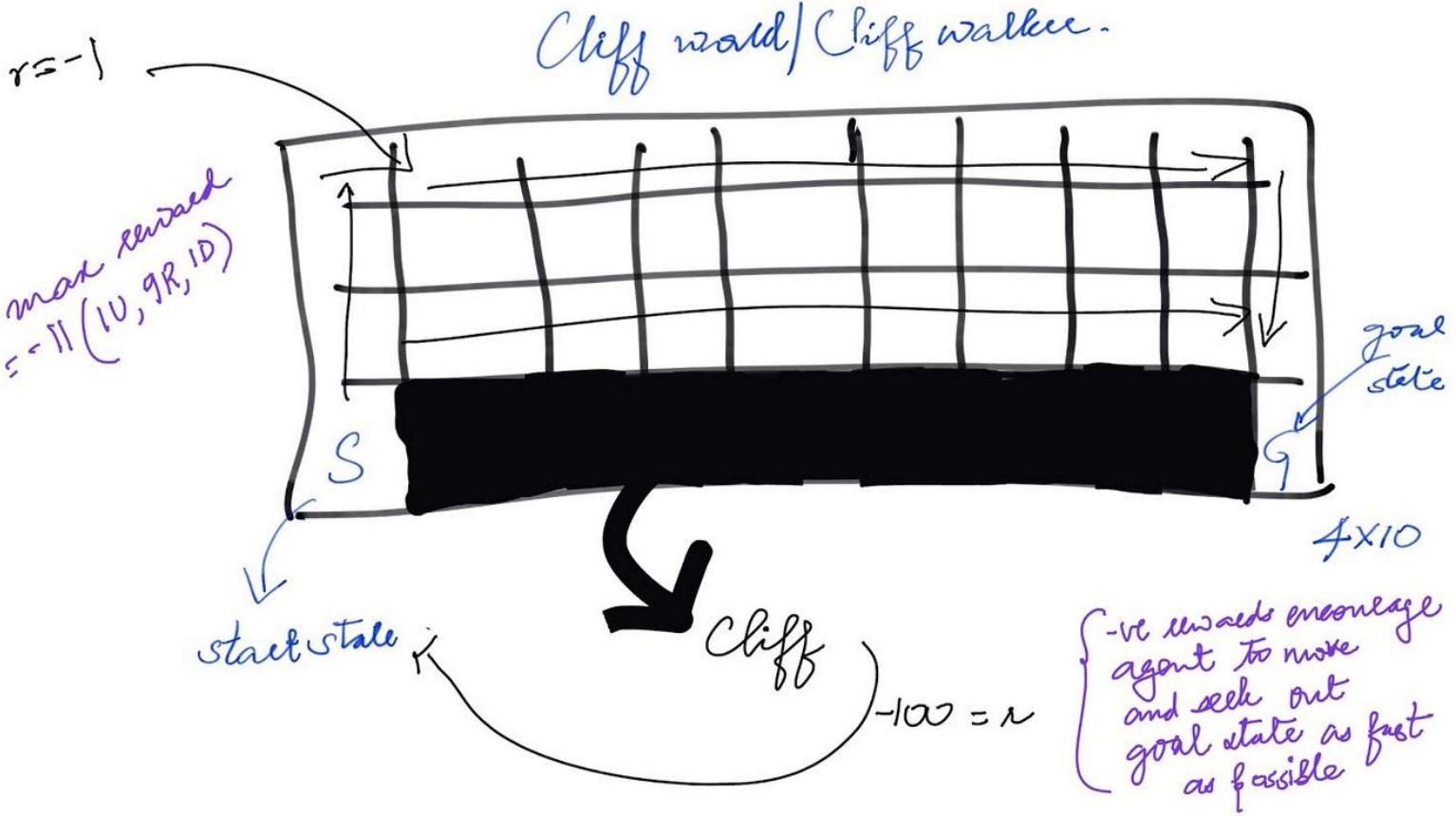
don't know if choices made will pay off over time or if actions contributed to rewards observed

Grid World

each tile ~ one state

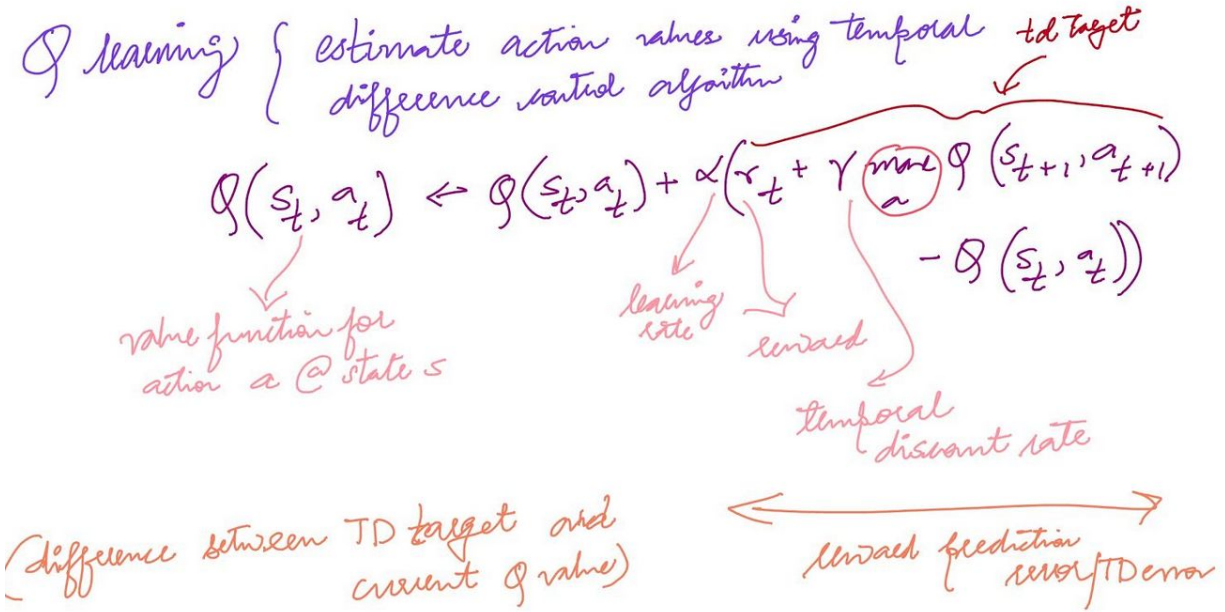


goal: find way to goal tile in most direct possible way while overcoming maze/other obstacles (static/dynamic)


Cliff world / Cliff walker.
r = -1
max reward
= -11 (1U, 9R, 1D)
S
goal state
G
4X10
start state
Cliff
-100 = r
-ve rewards encourage agent to move and seek out goal state as fast as possible

Q learning

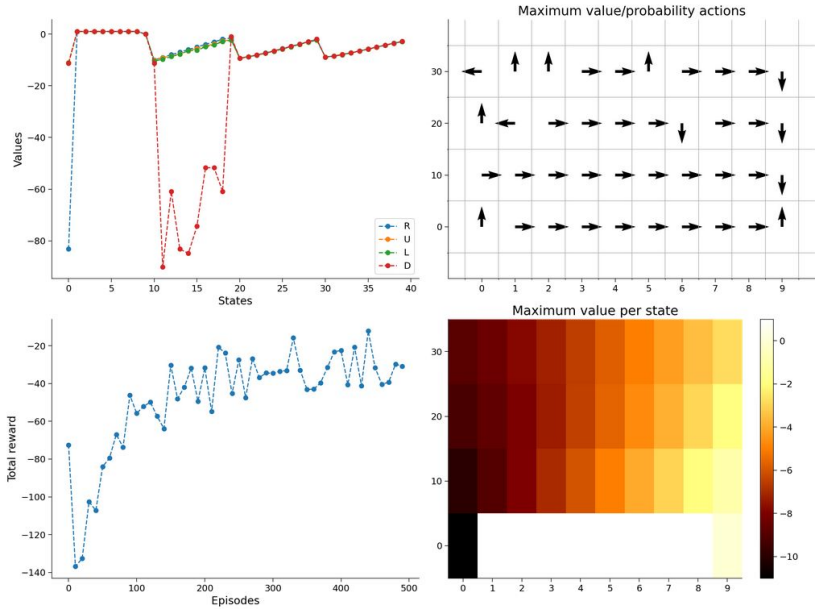
Because of the max operator used to select the optimal Q-value in the TD target, Q-learning directly estimates the optimal action value, i.e. the cumulative future reward that would be obtained if the agent behaved optimally, regardless of the policy currently followed by the agent. For this reason, Q-learning is referred to as an **off-policy** method.



Q-learning agent: epsilon-greedy strategy.

At each step, the agent will decide with probability $1-\epsilon$ to use the best action for the state it is currently in by looking at the value function, otherwise just make a random choice.

The process by which our the agent will interact with and learn about the environment implements the entire learning episode lifecycle of stepping through the state observation, action selection (epsilon-greedy) and execution, reward, and state transition.



Observations

We should see four plots that show different aspects on our agent's learning and progress.

- The top left is a representation of the Q-table itself, showing the values for different actions in different states. Notably, going right from the starting state or down when above the cliff is clearly very bad.
- The top right figure shows the greedy policy based on the Q-table, i.e. what action would the agent take if it only took its best guess in that state.
- The bottom right is the same as the top, only instead of showing the action, it's showing a representation of the maximum Q-value at a particular state.
- The bottom left is the actual proof of learning, as we see the total reward steadily increasing after each episode until asymptoting at the maximum possible reward of ~11.

Summary

Reinforcement learning agent based on Q-learning to solve the Cliff World environment.

Q-learning combined the epsilon-greedy approach to exploration-exploitation with a table-based value function to learn the expected future rewards for each state.

Tutorial #3 Bonus Explanations

SARSA (State-action-reward-state-action)

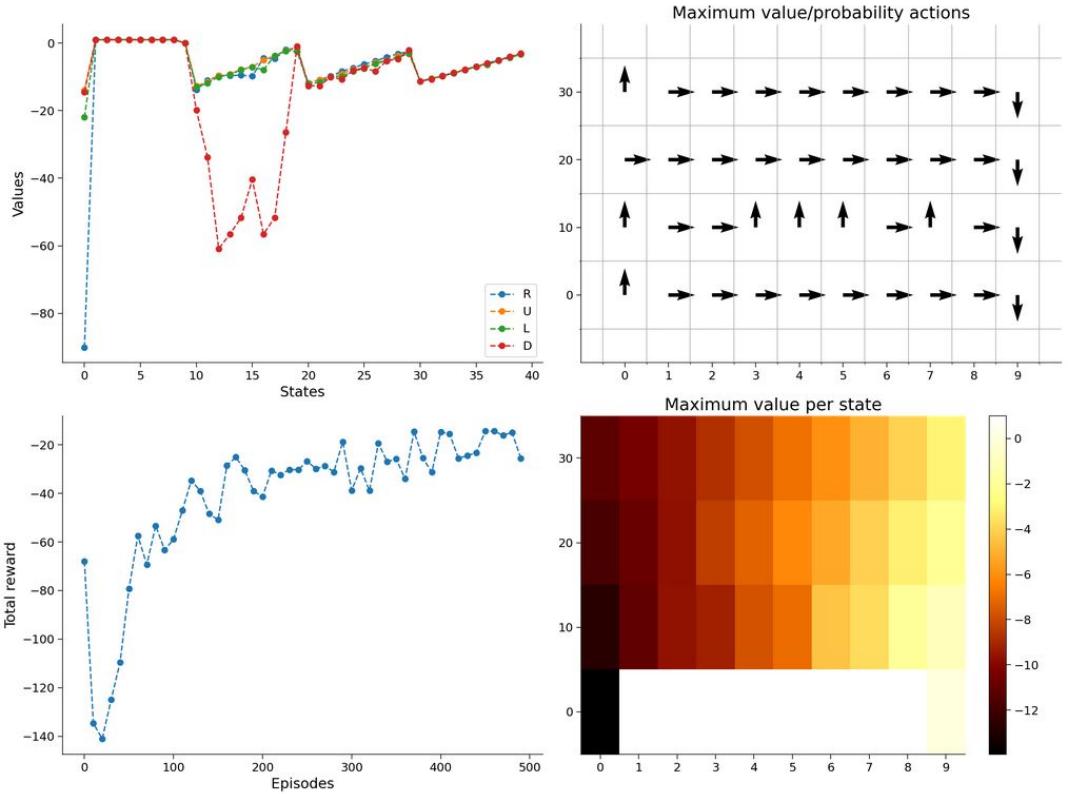
An alternative to Q-learning, the SARSA algorithm also estimates action values. However, rather than estimating the optimal (off-policy) values, SARSA estimates the **on-policy** action value, i.e. the cumulative future reward that would be obtained if the agent behaved according to its current beliefs.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right)$$

TD target calculation uses policy to select next action (ϵ greedy) rather than using action that maximises Q value

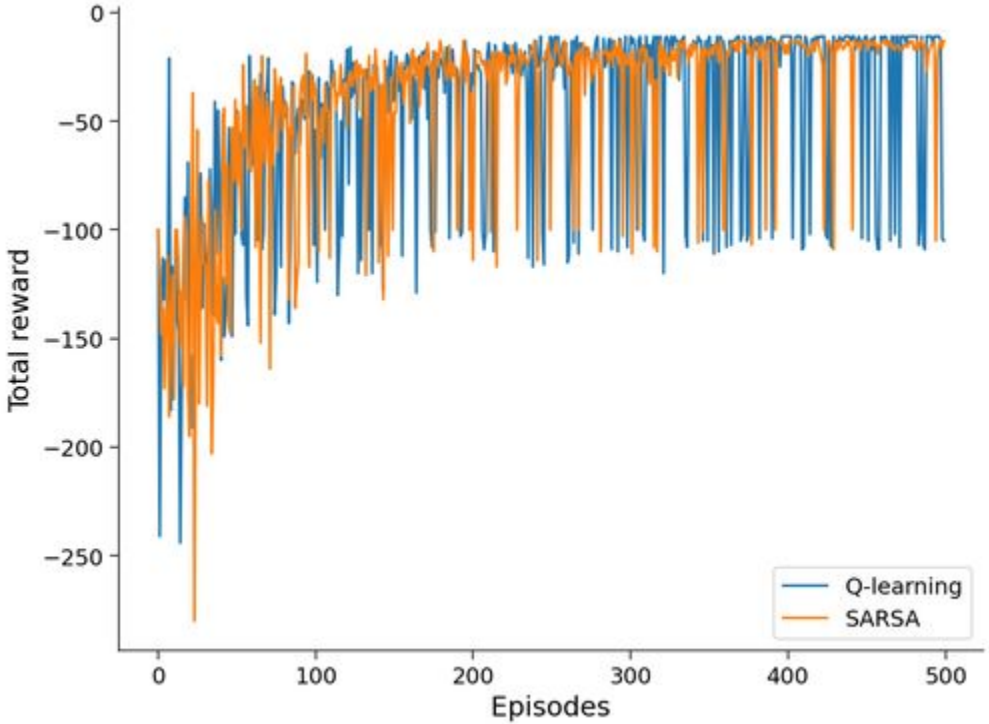
Observations

We should see that SARSA also solves the task with similar looking outcomes to Q-learning. One notable difference is that SARSA seems to be skittish around the cliff edge and often goes further away before coming back down to the goal.

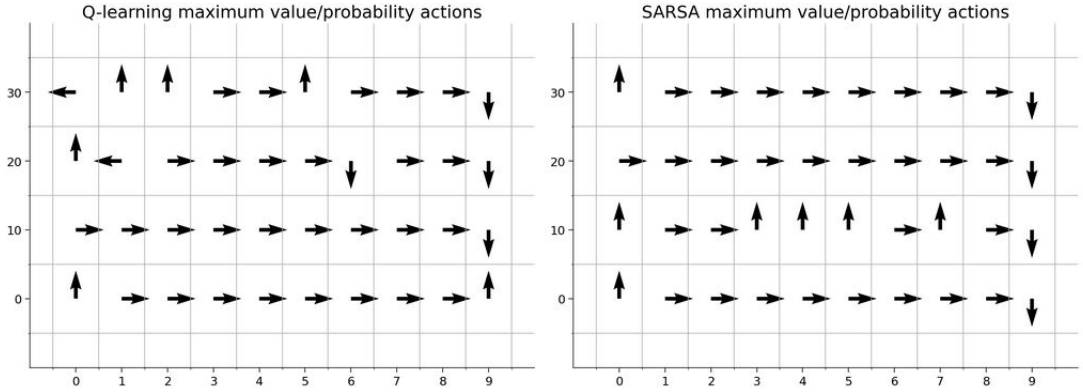


On policy vs off policy

On this simple Cliff world task, Q-learning and SARSA are almost indistinguishable from a performance standpoint, but we can see that Q-learning has a slight-edge within the 500 episode time horizon.



Observations



Q-learning learned to go up, then immediately go to the right, skirting the cliff edge, until it hits the wall and goes down to the goal. The policy further away from the cliff is less certain.

SARSA, on the other hand, appears to avoid the cliff edge, going up one more tile before starting over to the goal side. This also clearly solves the challenge of getting to the goal, but does so at an additional -2 cost over the truly optimal route.

Tutorial #4
Explanations

Objectives

Implement one of the simplest model-based Reinforcement Learning algorithms, Dyna-Q, by understanding world models, how it can improve the agent's policy, and the situations in which model-based algorithms are more advantageous than their model-free counterparts.

- Implement a model-based RL agent, Dyna-Q, that can solve a simple task;
- Investigate the effect of planning on the agent's behavior;
- Compare the behaviors of a model-based and model-free agent in light of an environmental change.

Model free vs model based

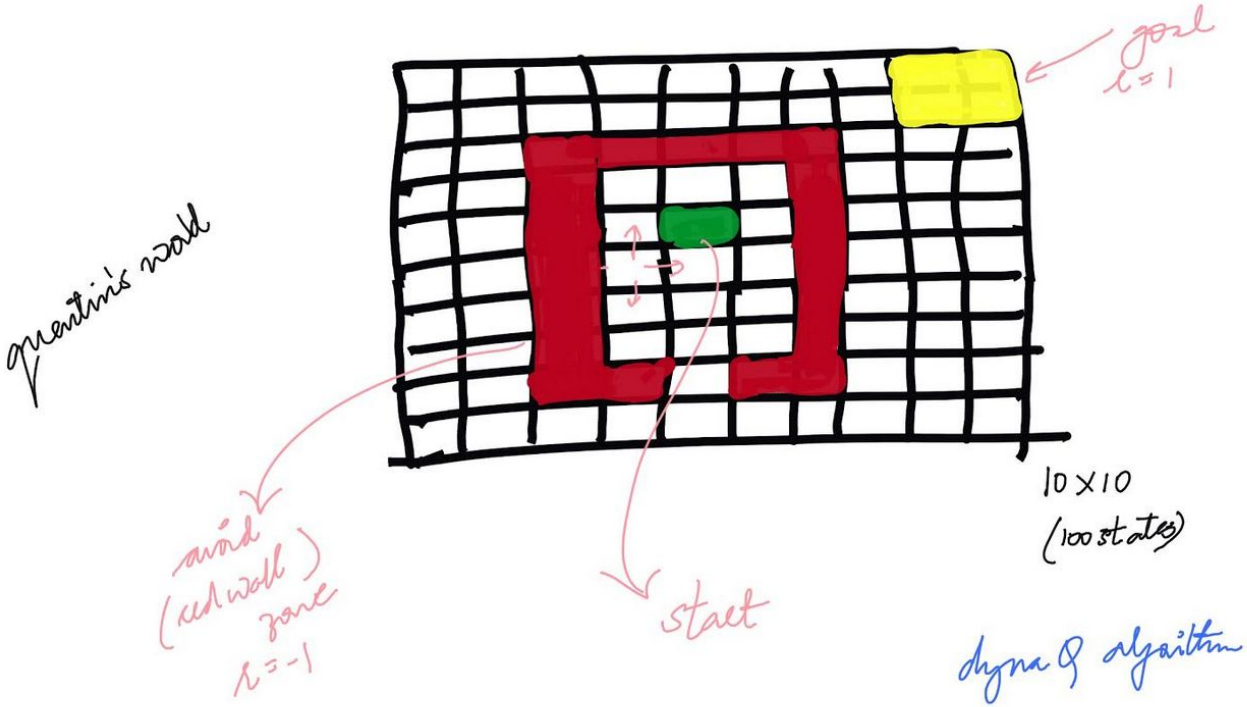
Model-free: they do not require a model to use or control behavior.

study a different class of algorithms called model-based.

In contrast to model-free RL, model-based methods use a model to build a policy

A model (sometimes called a world model or internal model) is a representation of how the world will respond to the agent's actions (representation of how the world works). With such a representation, the agent can simulate new experiences and learn from these simulations. This is advantageous for two reasons.

a. *First, acting in the real world can be costly and sometimes even dangerous. Learning from simulated experience can avoid some of these costs or risks.*
b. *Second, simulations make fuller use of one's limited experience. To see why, imagine an agent interacting with the real world. The information acquired with each individual action can only be assimilated at the moment of the interaction. In contrast, the experiences simulated from a model can be simulated multiple times -- and whenever desired -- allowing for the information to be more fully assimilated.*


goal
r = 1
puenting's world
wind
(red wall)
zone
r = -1
start
10 x 10
(100 states)
dyna Q algorithm

dyna Q agent { acting + learning + planning.

Algorithm: + planning routine to Q learning agent

{ agent acts in real world & learns from observed experience.

- agent allowed series of k planning steps.

@ each step }

↳ model generates simulated experience by using same Q learning rule

⇒ Dyna Q agent learns from 1 step of real experience during acting and from k steps of simulated experience during planning.

Dyna Q model

In Dyna-Q, as the agent interacts with the environment, the agent also learns the model. For simplicity, Dyna-Q implements model-learning in an almost trivial way, as simply caching the results of each transition. Thus, after each one-step transition in the environment, the agent saves the results of this transition in a big matrix, and consults that matrix during each of the planning steps. Obviously, this model-learning strategy only makes sense if the world is deterministic (so that each state-action pair always leads to the same state and reward), and this is the setting of the exercise below. However, even this simple setting can already highlight one of Dyna-Q major strengths: the fact that the planning is done at the same time as the agent interacts with the environment, which means that new information gained from the interaction may change the model and thereby interact with planning in potentially interesting ways.

Dyna Q algorithm

TABULAR DYNA-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in S$ and $a \in A$.

Loop forever:

> (a) $S \leftarrow$ current (nonterminal) state
> (b) $A \leftarrow \epsilon$-greedy(S,Q)
> (c) Take action A; observe resultant reward, R, and state, S'
> (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
> (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
> (f) Loop repeat k times:
>
> > $S \leftarrow$ random previously observed state
> > $A \leftarrow$ random action previously taken in S
> > $R, S' \leftarrow Model(S,A)$
> > $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

Dyna Q model updates

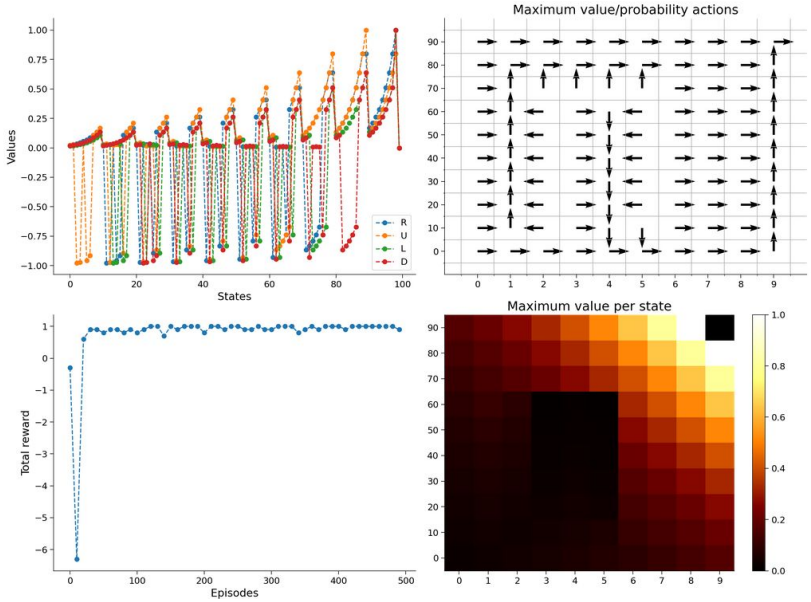
More specifically, after each action that the agent executes in the world, we need to update our model to remember what reward and next state we last experienced for the given state-action pair. Use it in the planning phase of Dyna-Q to simulate past experiences.

Dyna Q planning

We will sample a random state-action pair from those we've experienced, use our model to simulate the experience of taking that action in that state, and update our value function using Q-learning with these simulated state, action, reward, and next state outcomes

Observations

Dyna-Q agent is able to solve the task quite quickly, achieving a consistent positive reward after only a limited number of episodes

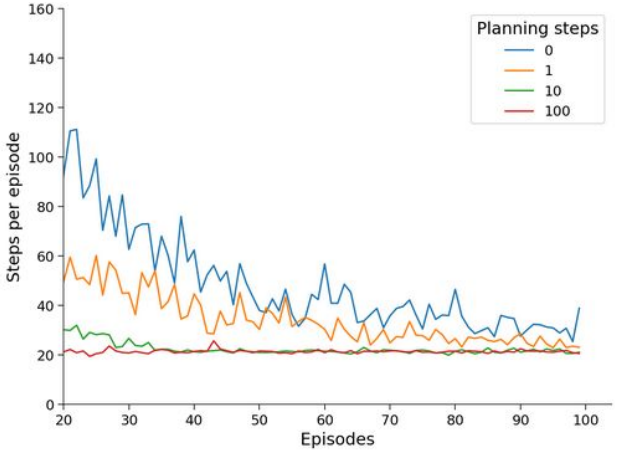


How much to plan?

How does changing the value of k impact our agent's ability to learn?

Run several experiments over several different values of k to see how their average performance compares. In particular, we will choose $k \in \{0,1,10,100\}$. $k=0$ corresponds to no planning. This is, in effect, just regular Q-learning.

After an initial warm-up phase of the first 20 episodes, number of planning steps has a noticeable impact on our agent's ability to rapidly solve the environment. After a certain value of k, our relative utility goes down, so it's important to balance a large enough value of k that helps us learn quickly without wasting too much time in planning.



Accommodating changes

planning can also help the agent to quickly incorporate new information about the environment into its policy. Thus, if the environment changes (e.g. the rules governing the transitions between states, or the rewards associated with each state/action), the agent doesn't need to experience that change *repeatedly* (as would be required in a Q-learning agent) in real experience. Instead, planning allows that change to be incorporated quickly into the agent's policy, without the need to experience the change more than once.

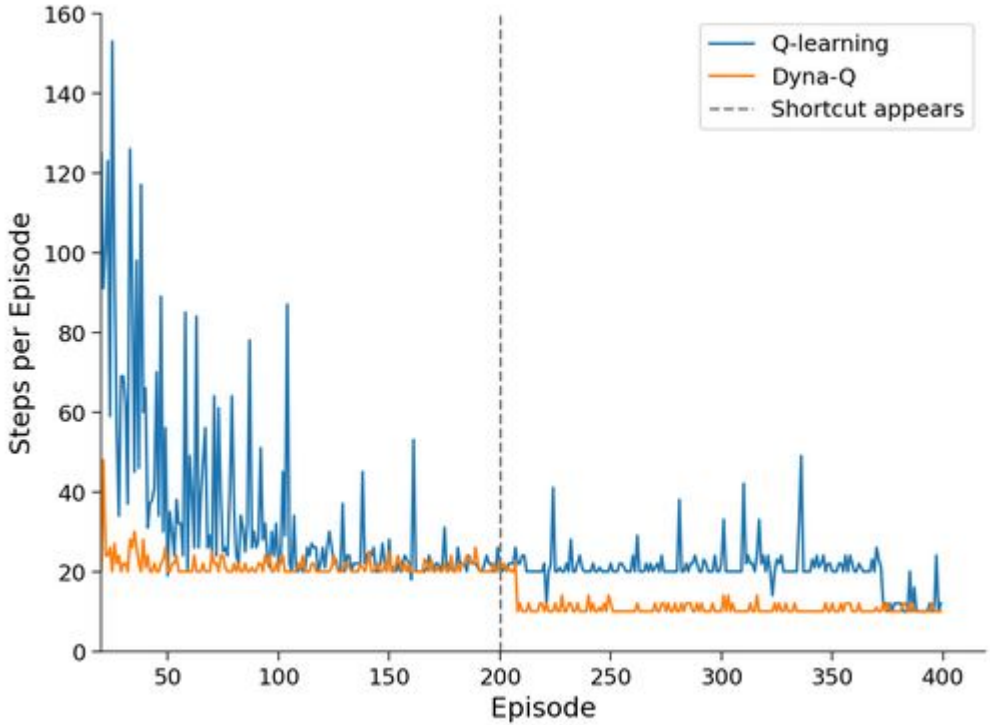
ADDING SHORTCUTS IN QUENTIN'S WORLD

k=10 for our Dyna-Q agent with 10 planning steps.
The main difference is we now add in an indicator as to when the shortcut appears. In particular, we will run the agents for 400 episodes, with the shortcut appearing in the middle after episode #200.

When this shortcut appears we will also let each agent experience this change once i.e. we will evaluate the act of moving upwards when in the state that is below the now-open shortcut. After this single demonstration, the agents will continue on interacting in the environment.

Observations

If all went well, we should see the Dyna-Q agent having already achieved near optimal performance before the appearance of the shortcut and then immediately incorporating this new information to further improve. In this case, the Q-learning agent takes much longer to fully incorporate the new shortcut.



Summary

Dyna-Q is very much like Q-learning, but instead of learning only from real experience, you also learn from ***simulated*** *experience. This small difference, however, can have huge benefits! Planning frees the agent from the limitation of its own environment, and this in turn allows the agent to speed-up learning -- for instance, effectively incorporating environmental changes into one's policy.*

Not surprisingly, model-based RL is an active area of research in machine learning. Some of the exciting topics in the frontier of the field involve (i) learning and representing a complex world model (i.e., beyond the tabular and deterministic case), and (ii) what to simulate -- also known as search control -- (i.e., beyond the random selection of experiences).

The framework above has also been used in neuroscience to explain various phenomena such as planning, memory sampling, memory consolidation, and even dreaming!

Resources

https://analyticsindiamag.com/reinforcement-learning-that-dreams/