



TDSB Co-op Project

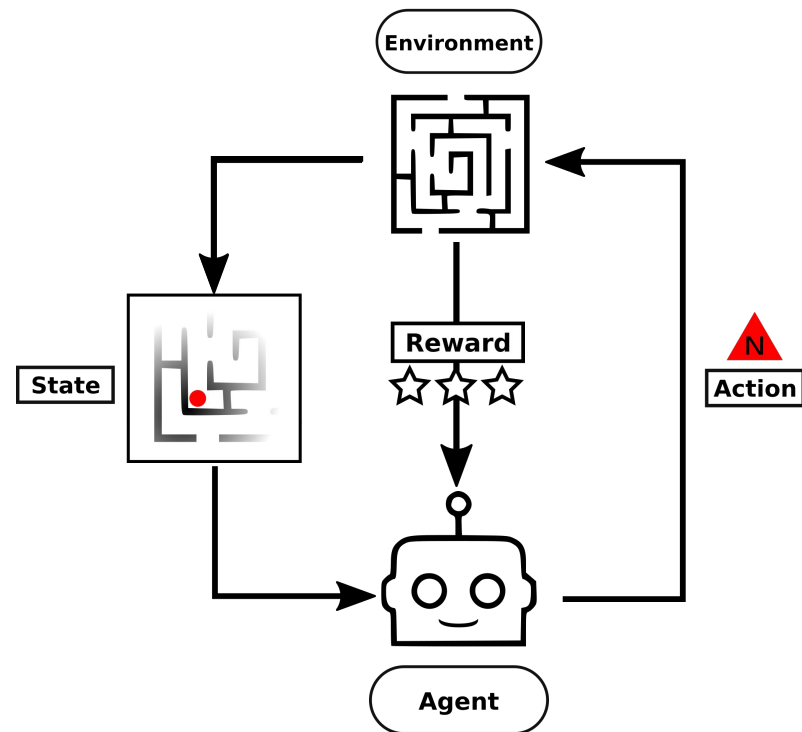
Meet Patel & Annik Carson

Overview / To Do List

- **Reinforcement Learning Problems**
- **Environments**
 - **Transition and Reward functions**
 - **States and observations**
- **Agents**
 - **Value functions ($V(s)$ / $Q(s,a)$)**
 - **Policies**
- **Tabular Q Learning**
- **Deep Learning (Neural Networks)**
 - **Deep Q networks (still using ϵ -greedy policy)**
 - **Policy Approximation**
 - **Actor-critic networks**

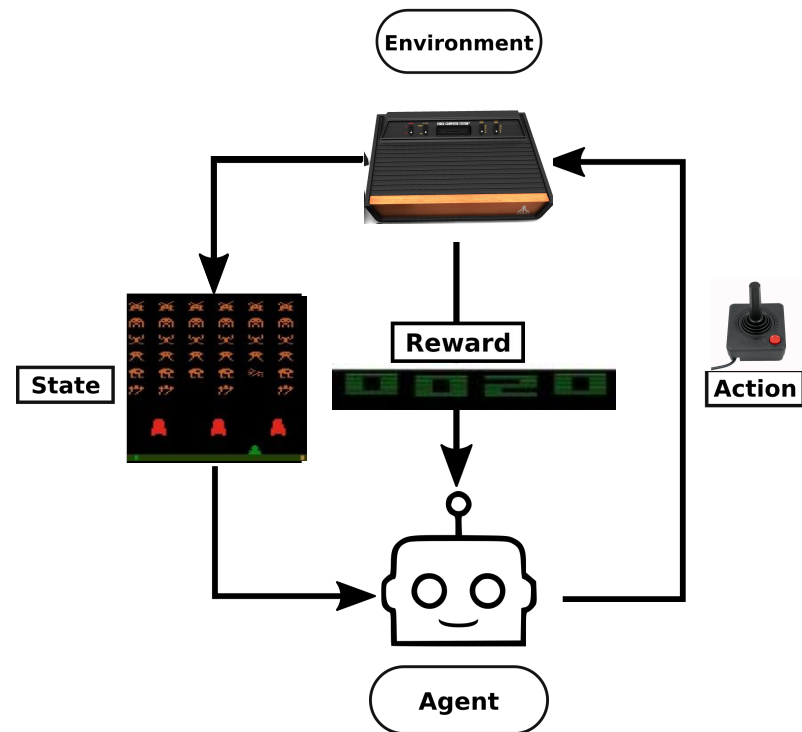
Learning Smart Behaviour

- How should agents act **optimally**?
 - Optimality ~ maximizing cumulative reward
- What information is available to learn with?
 - Trial and error interaction with the world



Learning Smart Behaviour

- How should agents act **optimally**?
 - Optimality ~ maximizing cumulative reward
- What information is available to learn with?
 - Trial and error interaction with the world



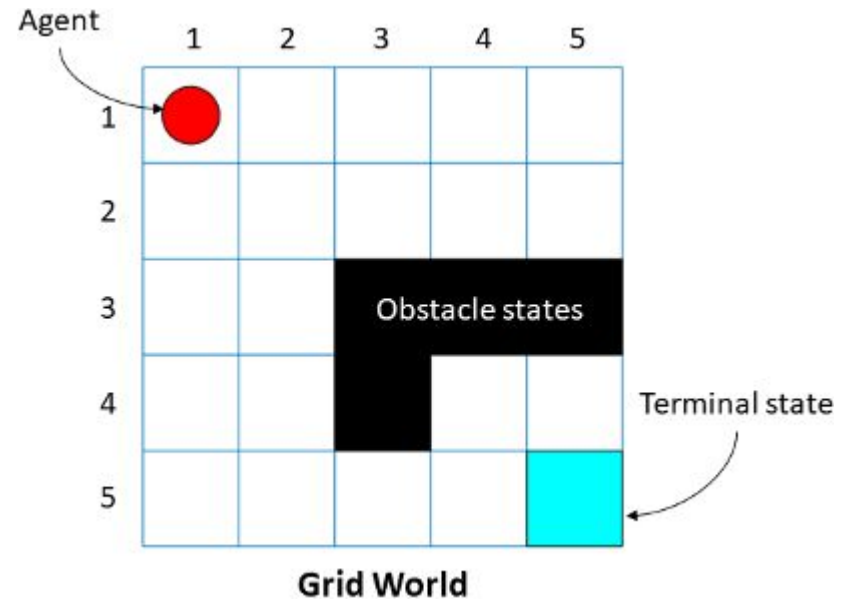
David Silver, 2015

Environment

State Ex. (1,1), (5,5)

Action Ex. "Down"

Reward Ex. -1 or +10



Environment

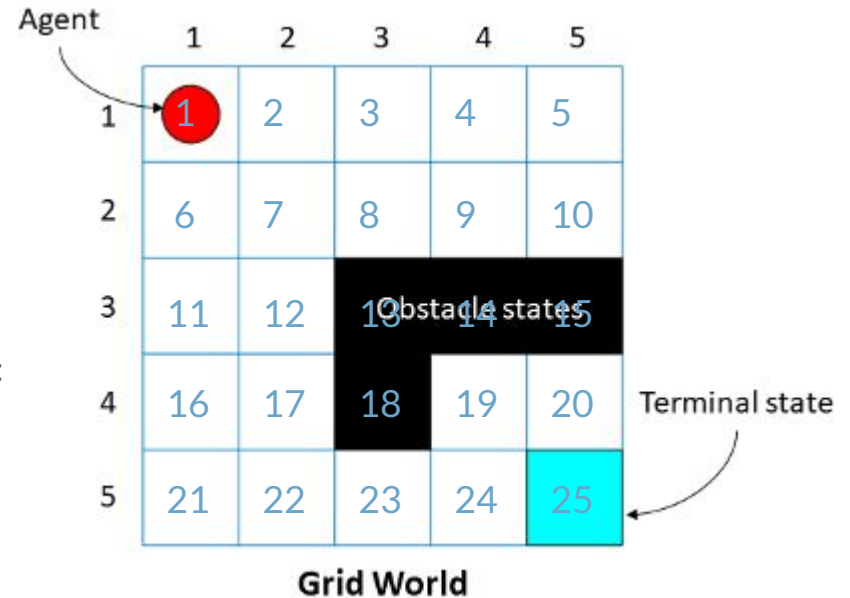
Reward function $R(\text{state})$

- $R(1,1) = -1$
- $R(5,5) = +10$

$R(x)$ for this environment can be represented by a vector:

```
[-1., -1., -1., ..., -1., -1., 10.]
```

How long is the vector $R(x)$?



Environment

Transition Function $T(\text{state1}, \text{state2}, \text{action})$:

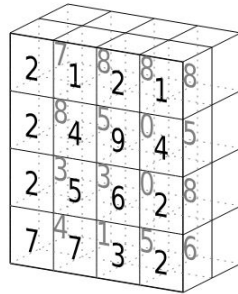
A rule for what moves are allowed

't'
'e'
'n'
's'
'o'
'r'

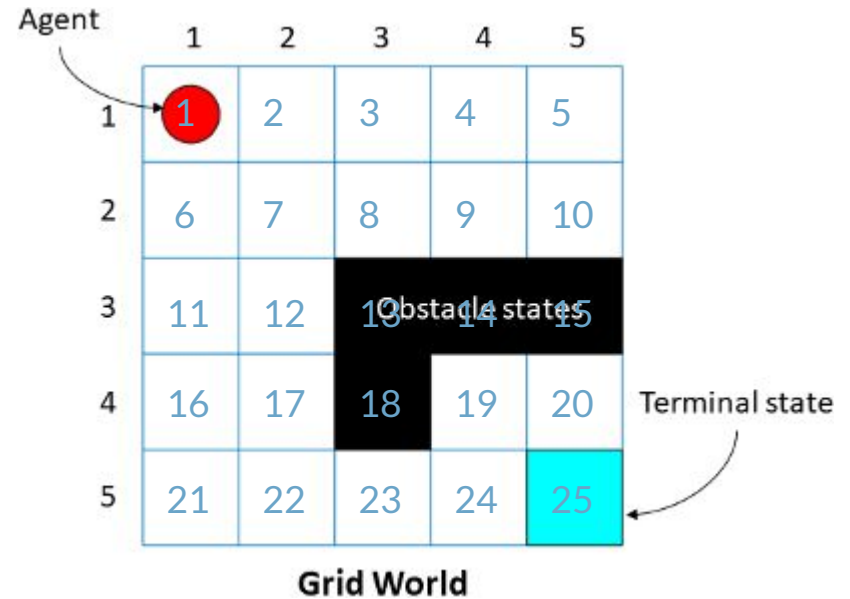
tensor of dimensions [6]
(vector of dimension 6)

3	1	4	1
5	9	2	6
5	3	5	8
9	7	9	3
2	3	8	4
6	2	6	4

tensor of dimensions [6,4]
(matrix 6 by 4)



tensor of dimensions [4,4,2]
(matrix 6 by 4)



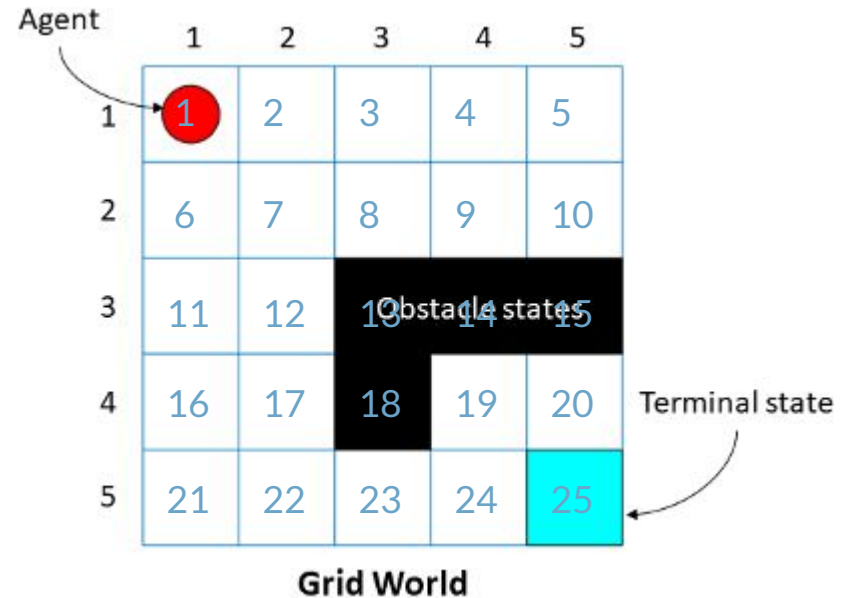
Environment

Transition Function $T(\text{state1}, \text{state2}, \text{action})$:

A rule for what moves are allowed

For state 1 (i.e. (1,1)):

	S1	S2	S3	S4	S5	S6	...	S25
Down	[0., 0., 0., 0., 0., 1., ..., 0.]							
Up	[1., 0., 0., 0., 0., 0., ..., 0.]							
Left	[1., 0., 0., 0., 0., 0., ..., 0.]							
Right	[0., 1., 0., 0., 0., 0., ..., 0.]							



Environment

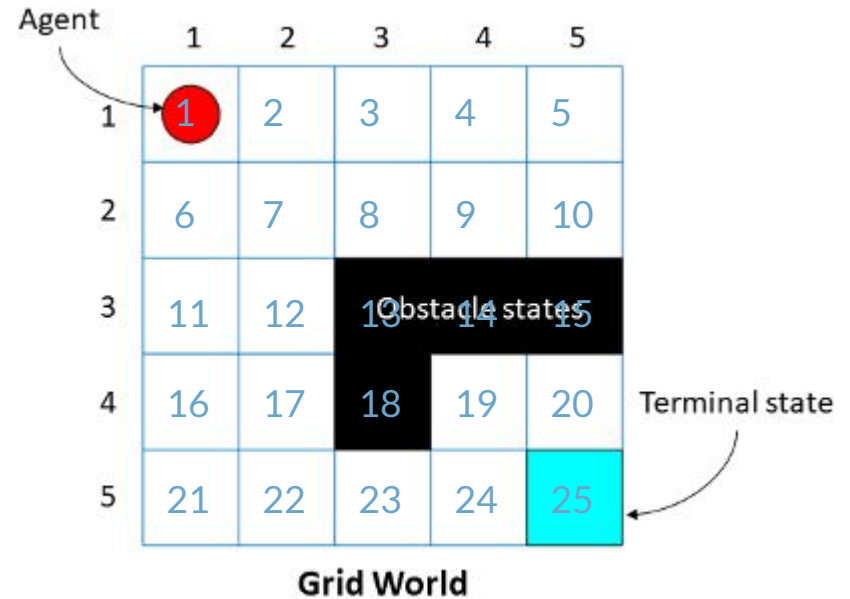
A different transition function:

Environment has wrapping edges ex. pacman

For state 1 (i.e. (1,1)):

	S1	S2	S3	S4	S5	S6	...	S25
Down	[0., 0., 0., 0., 0., 0., 1., ..., 0.]							
Up	[0., 0., 0., 0., 0., 0., 0., ..., 0.]							
Left	[0., 0., 0., 0., 0., 0., 0., ..., 0.]							
Right	[0., 1., 0., 0., 0., 0., 0., ..., 0.]							

There should be a 1 in the matrix at position [1, 20]
i.e. for action up the probability of transitioning to
state 21 is 1



Environment

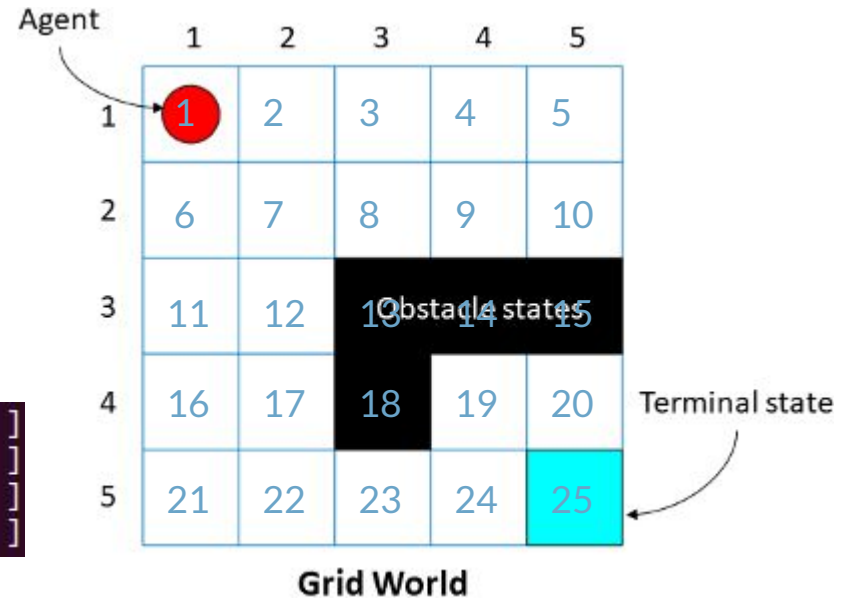
A different transition function:

Environment has a strong wind blowing to the right

For state 1 (i.e. (1,1)):

	S1	S2	S3	S4	S5	S6	S7	S8	...	S25
Down	[0. , 0. , 0. , 0. , 0. , 0.5, 0.5, 0. , ... , 0.]									
Up	[0.5, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , ... , 0.]									
Left	[1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , ... , 0.]									
Right	[0. , 0.3, 0.6, 0.1, 0. , 0. , 0. , 0. , 0. , ... , 0.]									

The probability of leaving a state = 1
 → Sum of values along the row = 1



Environment

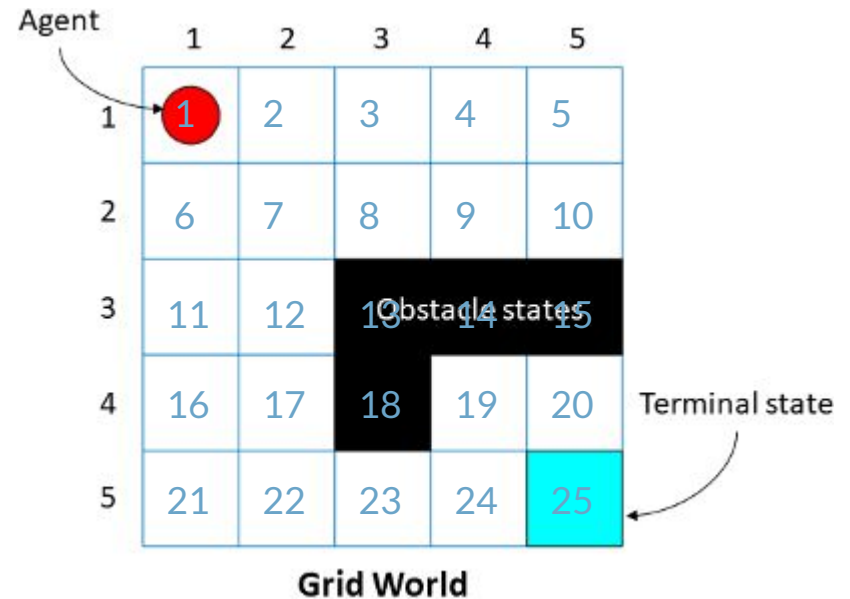
What do we need to define an environment?

A set of states: $\{s_1, s_2, \dots\} = \{(1,1), (1,2), (1,3) \dots\}$

A set of actions: $\{a_1, a_2, \dots\} = \{\text{Down, Up, Left, Right}\}$

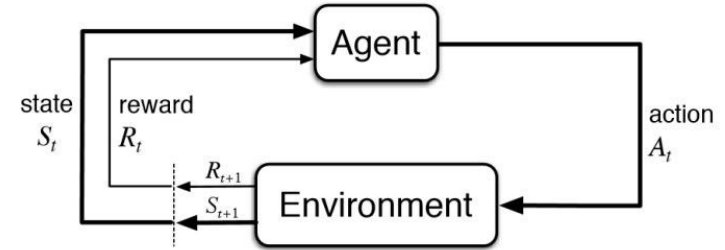
$T(s, s', a)$ - Rule for movement between states

$R(s)$ - Rule for how much reward a state is worth



Agent

- Takes info from environment
 - State / observation (some subset of info about state)
- Selects action to behave in environment
 - Rule for how to select action is called a policy (π)
- Acts on environment and gets info back:
 - Reward (how good was that action?)
 - Next state (what was the consequence of my behaviour?)





Policy = Rule for how to behave

The “rule” for how to act is described by a probability of taking different actions*

Ex.

- Behave randomly (all actions have same probability)
- Go left (action left has $p=1$, all others have $p=0$)

* If you have options a, b, and c then $P(a) + P(b) + P(c) = 1$
ie. the probability of all options must be 1.



Task 1: Define an environment

1. What size? (what is the set of unique states $\mathbf{S} = \{s_1, s_2 \dots, s_n\}$?)
2. What can you do? (what is the set of unique actions $\mathbf{A} = \{a_1, a_2 \dots, a_n\}$?)
3. What are the consequences of your actions?
 - a. Where do you go next? (what is the transition function $\mathbf{T}(s, s', a)$?)
 - b. What reward do you get? (what is the reward function $\mathbf{R}(s)$?)

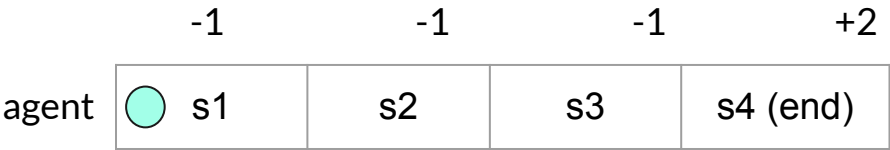


Defining the environment...

Starting with a simple environment,

- There are 4 states (s_1, s_2, s_3, s_4)
- There are 2 actions (left and right)
- Left and right actions allow the agent to move to adjacent states to the left or right of the current state.
- When the agent is at the first state, the left action will keep the agent in the same state since there is not an available state to move into.
- Each movement will give a certain reward, in this case each movement will give a reward of -1 and when a movement to the final state is made, the reward will be +2

So what does this look like?



Reward vector: $s1 \ s2 \ s3 \ s4$
 $[-1], [-1], [-1], [2]$

Rewards are assigned to each state, and the end state has a bigger reward.

Transition array:

- The transition vector to the left allows us to view the possible moves that the agent can make from each state of the environment.
- In this environment only right and left movements are allowed.
- Moving left from the first state keeps the agent in the first state.

	Left		Right		
s1	[[1. 0.]	[[1. 0.]	[[0. 0.]	[[0. 0.]	
s2	[0. 1.]	[0. 0.]	[1. 0.]	[0. 0.]	
s3	[0. 0.]	[0. 1.]	[0. 0.]	[0. 0.]	
s4	[0. 0.]	[0. 0.]	[0. 1.]	[0. 0.]	
	S1	S2	S3	S4	

Reward Function



```
91 def set_reward_function(self):  
92     # write how rewards are given by the environment  
93     # return an object R and then set self.R = set_reward_function()  
94     R = np.ones((self.nstates, 1))*-1  
95     R[-1] = 2  
96     return R  
97
```

This function sets all the values in the reward vector to -1, except for the last state, which is denoted by `nstates-1`, is set to 2. This function can work for any environment as long as it is desired for all states to have the same reward except the last one.

Transition Function for this environment



```
9  def set_transition_function(self):
10     # write how transitions should be handled by the environment
11     # return an object T and then set self.T = set_transition_function()
12     T = np.zeros((self.nstates,self.nstates,self.nactions))
13
14     T[0,0,0] = 1
15     T[0,1,1] = 1
16     T[1,0,0] = 1
17     T[1,2,1] = 1
18     T[2,1,0] = 1
19     T[2,3,1] = 1
20
21     return T
```

The transition array was created manually for this environment. First the entire array was set to 0, and then each location where the agent could have possibly moved to from each state was manually set to 1.

This function returns the transition array for the linear track environment created. Can not be used for any other environment.

Improved version of transition function



```
23 def linear_track_transition(self):
24     T = np.zeros((self.nstates, self.nstates, self.nactions))
25     for action in range(self.nactions):
26         if action == 0: # left
27             for state in range(self.nstates):
28                 if state == 0:
29                     T[state, state, action] = 1
30                 if state == self.nstates-1:
31                     T[state, state, action] = 0
32                 else:
33                     T[state, state-1, action] = 1
34         elif action == 1: # right
35             for state in range(self.nstates):
36                 if state == self.nstates-1:
37                     T[state, state, action] = 0
38                 else:
39                     T[state, state+1, action] = 1
40     return T
```

This transition function is specifically for linear tracks of any length. This allows the transition vector to be created automatically based on the user input of how many states there are, or in other words, how long the track is.

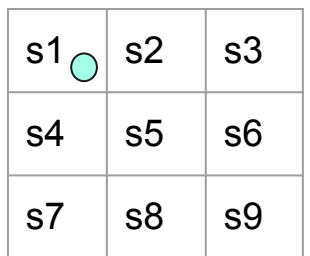
Same as before, this function assigns 0 to all the indices of the transition vector and then goes through each state and assigns 1 to the state before and the state after, indicating the possible moves from the current state.

Since the last state is the end, there are no possible moves from there.

Defining a grid environment

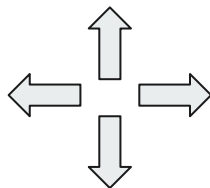
Defining the environment...

- There are n states ($s_1, s_2, s_3, s_4, \dots, s_n$)
- There are 4 actions (down, up, right and left)
- When there is no available space to move into with a specific action, taking that action keeps the agent in the same state
- Each movement will give a certain reward, in this case each movement will give a reward of -1 and when a movement to the final state is made, the reward will be +100 (same reward function will be used with minor changes)
- User inputs dimensions and number of states to create the environment



end

Possible actions

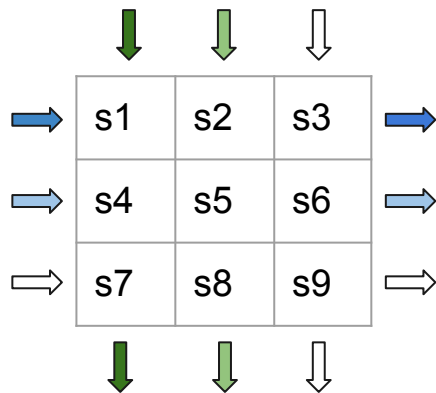


Transition Function for Grid Environments

```
76 def grid_transition(self):
77
78     # initialize
79     T = np.zeros((self.nstates, self.nstates, self.nactions)) # down, up, right, left
80
81     # add neighbor connections and jumps, remove for endlines
82     #T [starting_state, ending_state, action]
83     T[list(range(0, self.nstates-self.shape[1])), list(range(self.shape[1],self.nstates)), 0] = 1 #down
84     T[list(range(self.shape[1],self.nstates)), list(range(0, self.nstates-self.shape[1])),1] = 1 # up
85     T[list(range(0, self.nstates-1)), list(range(1, self.nstates)), 2] = 1 # right
86     T[list(range(1, self.nstates)), list(range(0, self.nstates-1)), 3] = 1 # left
87
88     #remove endlines
89     T[list(range(self.shape[1]-1, self.nstates-1, self.shape[1])), list(range(self.shape[1], self.nstates, self.shape[1])), 2 ] = 0
90     T[list(range(0,self.nstates-self.shape[1]+1, self.shape[1])) , list(range(-1,self.nstates-self.shape[1], self.shape[1])), 3] = 0
91     # include self transitions
92     #T[start_state, end_state, action]
93     T[list(range(self.nstates-self.shape[1],self.nstates)),list(range(self.nstates-self.shape[1], self.nstates)), 0] = 1
94     T[list(range(self.shape[1])),list(range(self.shape[1])), 1] = 1
95     T[list(range(self.shape[1]-1, self.nstates, self.shape[1])), list(range(self.shape[1]-1, self.nstates, self.shape[1])), 2 ] = 1
96     T[list(range(0,self.nstates-self.shape[1]+1, self.shape[1])) , list(range(0,self.nstates-self.shape[1]+1, self.shape[1])), 3] = 1
97
98     return T
```

This function sets all the values of the transition array to 0 and then goes through all the states and assigns 1 to all the possible states that can be achieved from the current state. This function also accounts for the outcome when performing an action which is not possible, such as trying to go down when at the bottom edge of the grid. This results in staying in the same state.

Adding edge wrapping



In this environment, making an action when on the edges of the grid brings the agent back to the beginning of the row/column for the right and down actions at the right and bottom edge, and to the end of the row/column for the left and top edge.

```
#T[start_state, end_state, action]
T[list(range(self.nstates-self.shape[1],self.nstates)),list(range(self.shape[1])), 0] = 1
T[list(range(self.shape[1])),list(range(self.nstates-self.shape[1],self.nstates)), 1] = 1
T[list(range(self.shape[1]-1, self.nstates, self.shape[1])),list(range(0,self.nstates-self.shape[1]+1, self.shape[1])), 2] = 1
T[list(range(0,self.nstates-self.shape[1]+1, self.shape[1])), list(range(self.shape[1]-1, self.nstates, self.shape[1])), 3] = 1
```

The transition function for this environment is identical to the one for the normal environment, however instead of staying in the same state when performing an illegal action at an edge, this segment of code allows the agent to wrap around the edge back to the beginning of the row/column (depending on action and state).

The *move* function

Using the move function, we can determine the next state, reward and whether the end state has been reached.

```
137 def move(self, current_state, selected_action):
138     done=False
139     probabilities = self.T[current_state, :,selected_action]
140     next_state = np.random.choice(np.arange(self.nstates), p=probabilities)
141     reward=self.R[next_state,0]
142
143     if reward != -1:
144         done=True
145
146     return next_state, reward, done
```



```
1 print(test_env.move(2,0))
```

```
(5, -1.0, False)
```

For example, when 2 is the state and 0 is the action representing down, the function returns the state that the agent will end up in along with the reward that comes with it. It will also output true or false depending on whether the next state is the terminal state.

s1	s2	s3
s4	s5	s6
s7	s8	s9

Testing the environment

```

7 def printMoves(state):
8     done=False
9
10    if test_env.R[state,0]!=-1:
11        done=True
12        print("(" ,state," ", " ,test_env.R[state,0]," ", " , True,")" )
13
14    while not done:
15        action = np.random.choice(np.arange(test_env.nactions))
16        result=test_env.move(state, action)
17        print(f"state={state} action={action} next state={result[0]} reward={result[1]} end={result[2]}")
18        new_state = result[0]
19        reward = result[1]
20        done = result[2]
21        state = new_state

```

This function goes through the environment through random movements and prints out the conditions at each step until the terminal state is reached.



1 printMoves(0)

```

state=0 action=0 next state=3 reward=-1.0 end=False
state=3 action=1 next state=0 reward=-1.0 end=False
state=0 action=3 next state=2 reward=-1.0 end=False
state=2 action=3 next state=1 reward=-1.0 end=False
state=1 action=2 next state=2 reward=-1.0 end=False
state=2 action=1 next state=8 reward=100.0 end=True

```



Through a series of random actions the terminal state was reached in 6 steps

Checking probabilities in the transition array

```

12 def check_transition_prob(self, transition_matrix):
13     action=0
14     check= True
15     for action in range(self.nactions):
16         for state in range(self.nstates):
17             sum=0
18             for state2 in range(self.nstates):
19                 sum+=transition_matrix[state, state2, action]
20                 if state2==self.nstates-1:
21                     if "%.1f" % (1,sum) != "%.1f" % (1,1):
22                         check=False
23                     break
24
25     return check

```

Sometimes there may be probabilities associated with the possible states you can end up in for a certain action. This function checks to make sure that all the probabilities of ending up in certain states for a certain action add up to 1.

For example if there is a windy day in the linear track environment taking the action “right” has 3 possibilities. Each possibility is assigned its own probability and all the probabilities for that action are equal to 1.

```

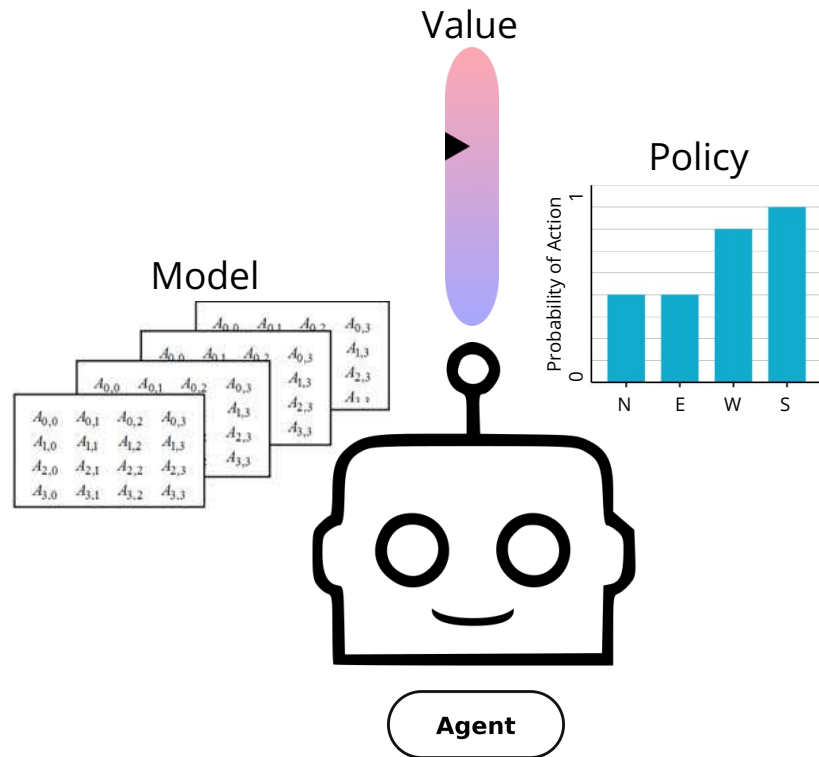
T[0,0,0] = 1
T[0,1,1] = 0.6
T[0,2,1] = 0.3
T[0,3,1] = 0.1
T[1,0,0] = 1
T[1,2,1] = 1
T[2,1,0] = 1
T[2,3,1] = 1
T[3,2,0] = 1
T[3,3,1] = 1

```

What's In An Agent?

An agent is made up of one or more of these pieces:

- Model
 - agent's representation of the environment
- Value Function
 - estimate how much reward is expected over time for each state and/or action
- Policy
 - function which maps states onto actions
 - A (conditional) probability distribution
 - gives the probability of selecting each action given current state





Value Function

Can estimate value being in a specific state i.e. $V(s)$

Or can estimate value of selecting a particular action from a specific state $Q(s,a)$



Policies

- A policy is a rule for how to behave
- We use the greek letter pi (π) to denote the policy
- π describes a probability distribution over actions
 - Ex $\pi(s) = [0.25, 0.25, 0.25, 0.25]$ = all actions equally likely
- How do we learn good policies?
 - Greedy / Epsilon-Greedy Policies: learn good estimates of value, then say that our policy is “select action with best value”
 - Can we learn the policy on its own?

Creating an agent...



- Behaviour of agent?
 - Random or
 - Based on action that has highest value
- Learning rate:
 - Controls speed at which the model learns. It controls the amount of how much the weights/values of the model are updated after each step.
- Discount rate:
 - Determines how much the agent values rewards in the immediate future in comparison to rewards in the distant future.

Creating the agent class

```

1 class agent(object):
2     def __init__(self, action_type, env):
3         self.action_space= np.arange(env.nactions)
4         self.action_t=action_type
5         self.q_table = np.random.uniform(low=-2, high=0, size=(env.nstates, env.nactions))
6
7         self.LEARNING_RATE=0.1
8         self.DISCOUNT=0.95
9
10        if self.action_t=="random":
11            self.choose_action=self.choose_random_action
12        elif self.action_t=="max_value":
13            self.choose_action=self.choose_max_value_action
14
15        def choose_random_action(self, state):
16            a=np.random.choice(self.action_space)
17            return a
18
19        def choose_max_value_action(self, state):
20            action= np.argmax(self.q_table[state,:])
21            return action
22
23        def q_update(self, current_state, current_action, reward, new_state):
24            current_q = self.q_table[ current_state, current_action]
25            max_future_q = np.max(self.q_table[new_state,:])
26
27            new_q = (1-self.LEARNING_RATE)*current_q + self.LEARNING_RATE*(reward + self.DISCOUNT*max_future_q)
28            self.q_table[current_state, current_action] = new_q

```

The agent class includes 3 functions. The first one causes the agent to take actions randomly. Calling the second function allows the agent to choose actions based on value. The third function is used to update the q table.

At the beginning, the q table is assigned random values, however, as the agent explores the environment it updates the q table using the third function called “q_update”. The q table basically holds the information of how much value a certain action holds at a certain state.

It can also be noticed that the learning rate is set to 0.1 and the discount factor is set to 0.95,



Q-learning procedure

1. Based on current state, the agent chooses the action that has the highest value based on q-table which has been initialized with random values
2. The next state and reward are returned
3. q-table is updated
4. Process continues repeating until terminal state is reached
5. When terminal state is reached, value of terminal state is set to 0 in q-table

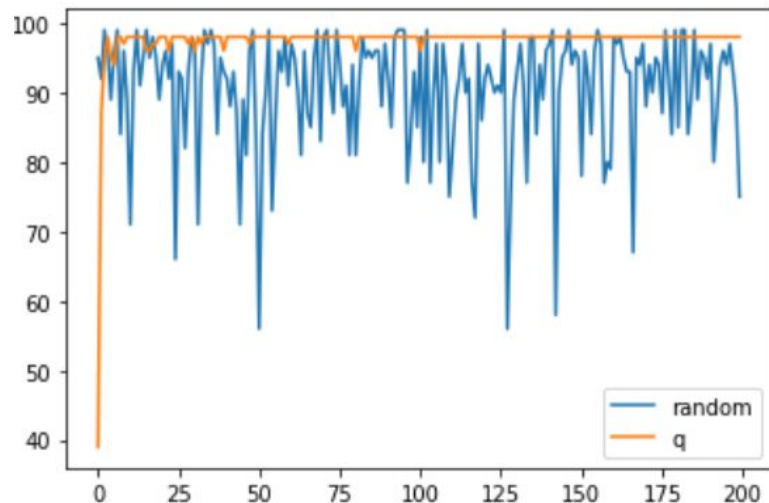
Training the agent multiple times creates a q-table that the agent can use to make the most optimal actions to reach the end in the lowest number of moves.

The navigate function

```
1 def navigate(env, agent, num_episodes):
2     reward_tracking=[]
3     max_steps= 1000
4     for episode in range(num_episodes):
5         total_reward=0
6         state=0 #np.random.choice(env.nstates)
7         #print(f"agent starts in state:{state}")
8         for step in range(max_steps):
9             action = agent.choose_action(state)
10            next_state, reward, done = env.move(state, action)
11            #print(f"state={state} action={action} next state={next_state} reward={reward} end={done}")
12            total_reward+=reward
13            if not done:
14                agent.q_update(state, action, reward, next_state)
15            else:
16                agent.q_table[state, action] = 0
17                break
18            state=next_state
19        #print(total_reward)
20        reward_tracking.append(total_reward)
21    return reward_tracking
```

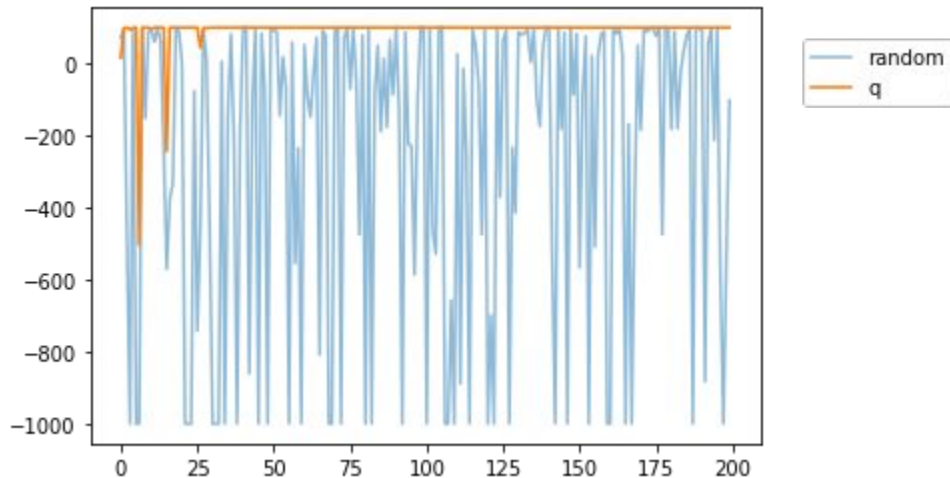
This is the function that performs all the steps stated in the previous slide. The q-table is updated in each step. There is a maximum of 1000 steps allowed if the terminal state is not reached before. The agent is trained like this for a set amount of episodes specified by the user. At the end, all the total rewards from each episode are output allowing us to view how the agent became more efficient each time.

Comparing the results



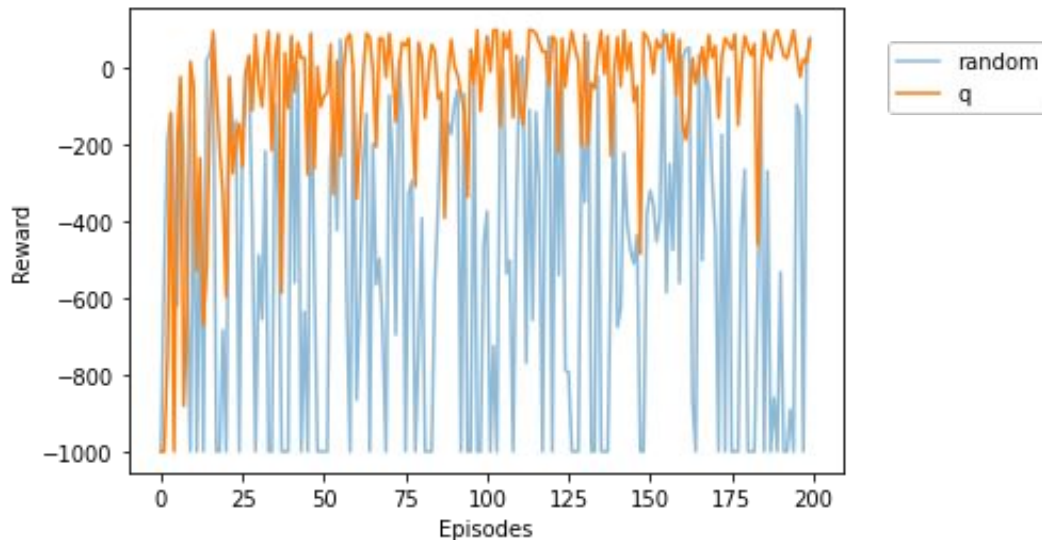
As can be seen in the graph to the left, q-learning spends a little time exploring its environment, and then is able to make the best choices to consistently achieve the end state in the least number of states possible. This can be viewed by the orange line which remains steady once the exploration phase is complete. On the other hand, randomly going through the environment is very inefficient.

Increasing environment size



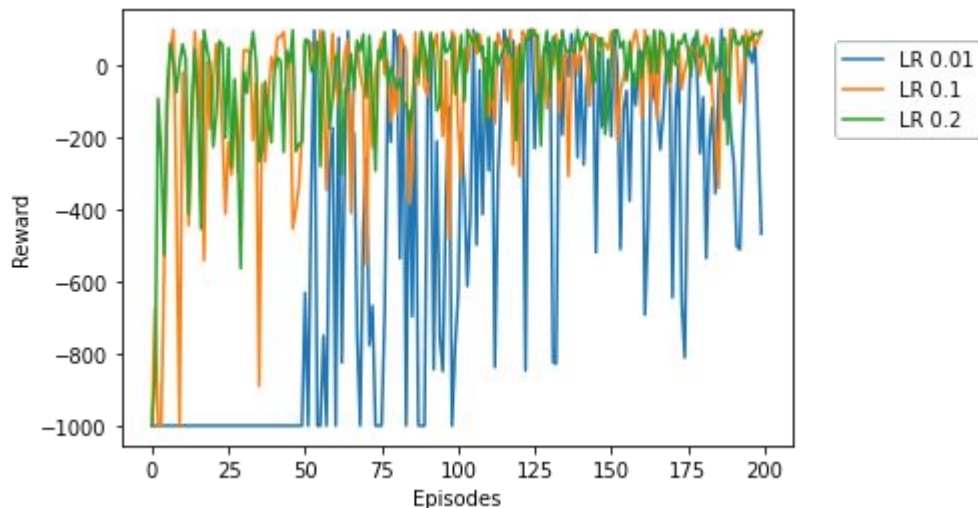
When the environment is increased in size from a 3x3 to a 20x20 grid, it can be seen that the randomly behaving agent is not able to reach the terminal state within the limit of 1000 steps anymore. This is where q-learning is seen to be significantly more advantageous than attempting to perform the same task randomly. Another benefit of q-learning is that once the agent has found the optimal path, it is able to consistently achieve the terminal state.

Starting at Random States



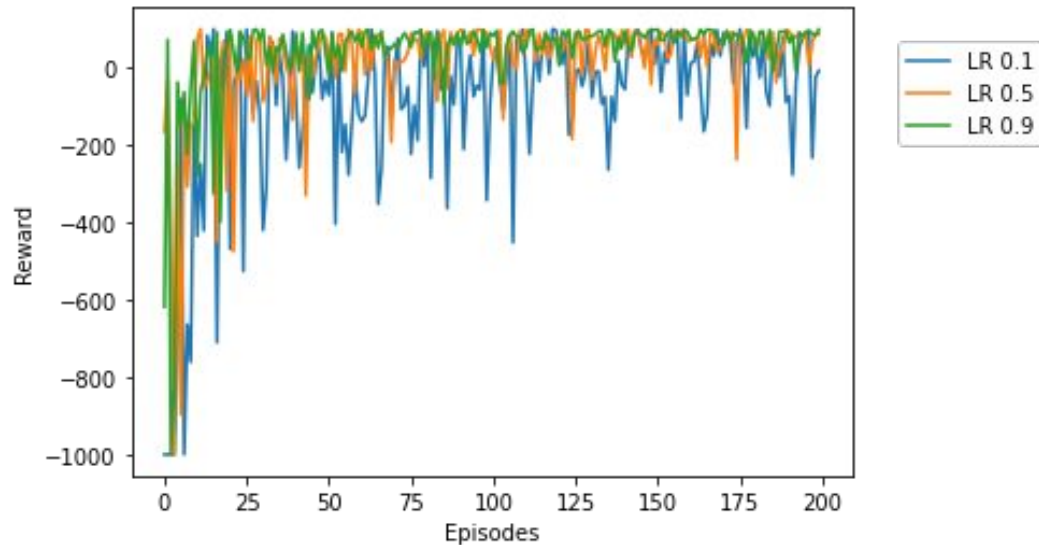
In this case, the agent began from a random state at the beginning of each episode. As can be seen, it took longer for the agent to learn from its environment because it was no longer following the same path each time. Now, the agent had to explore the environment from different starting points.

Effect of Learning rates



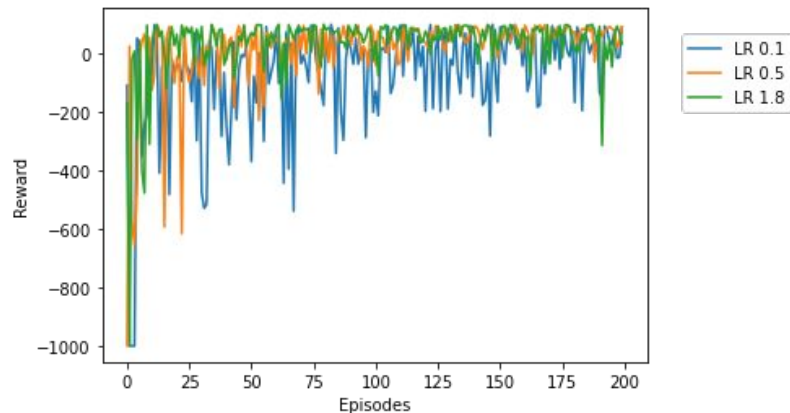
From the plot to the left it can be seen that a greater learning rate proves to be more efficient for this certain environment. The agent is able to learn more quickly in comparison to other agents with lower learning rates. For example, the agent with a learning rate of 0.01 doesn't show any significant improvement until 100 episodes have been completed.

Effect of Learning rates

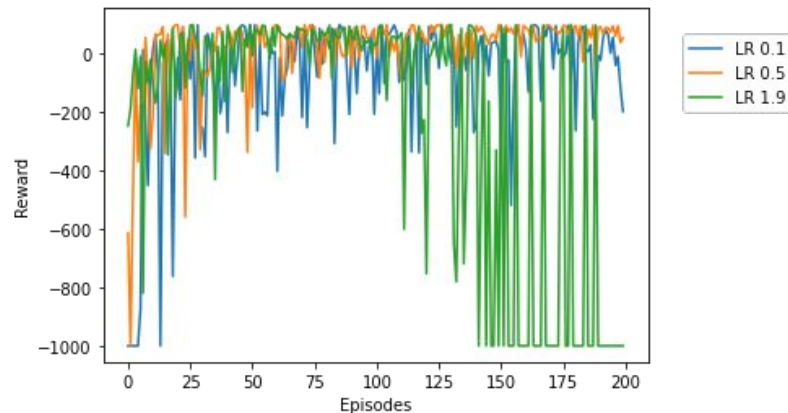


As the learning rate is increased, the plot of the agent becomes less “noisy” and tends to have minimal variation in comparison to lower learning rates which tend to have significant discrepancy in their plots.

High learning rates

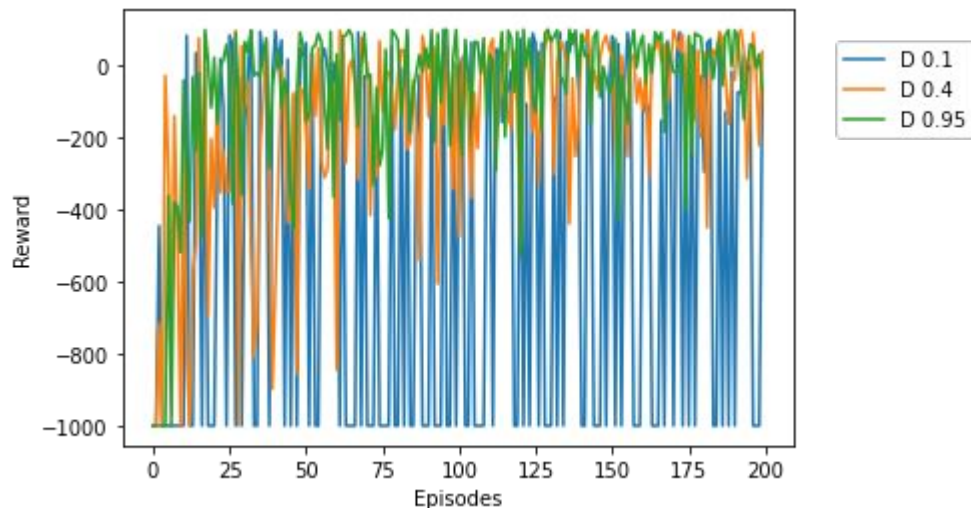


However, when the learning rate is really high, the agent does not perform as expected.



At 1.8 the performance of the agent seems to get worse as the learning rate increases. At 1.9 a significant change is visible in the plot.

Discount



A greater discount is seen to have better results as the agent is able to make smarter choices. A higher discount means that the agent values future rewards more than immediate rewards. This allows the agent to be able to look into the future and determine which states will be more beneficial to move to.

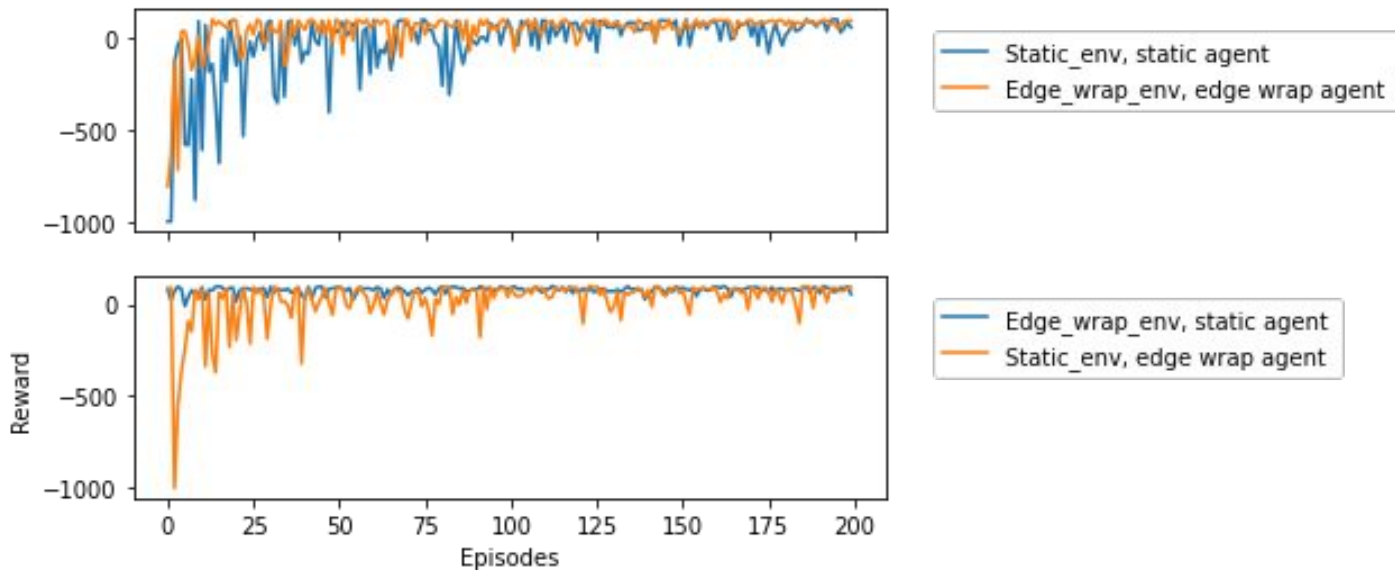
Predicting the effect of introducing trained agents to new environments



- We have 2 environments
 - A static environment where the edges act as boundaries
 - An edge wrapping environment where moving towards the edge brings the agent back to the beginning of that row/column
- The agent trained in the static environment should be able to do just fine in the edge wrapping environment because even though this new environment allows the agent to wrap around edges, the agent has been trained to go through the states in a regular path. This means that the agent will perform as it was previously trained and will be able to perform pretty well.
- The agent trained in the edge wrapping environment will struggle in the static environment because its training allowed it to become efficient by taking advantage of the edge wrapping property of the environment. When introduced to the static environment the agent will no longer be able to use the training it went through as an advantage. Therefore, this agent will take longer to adjust itself to this new environment.

Comparing results

The predictions made have proven to be correct. As expected, the agent trained in the edge wrapping environment struggled in its new environment at the beginning and then slowly adjusted to it and got better. It is also clearly visible that the static agent was able to use its training to do well in the new environment.





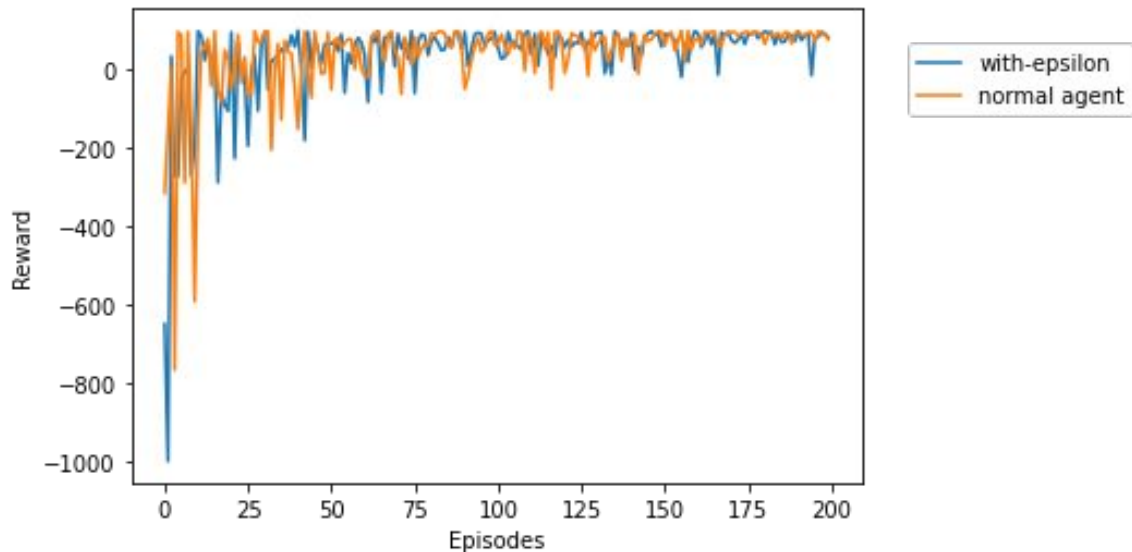
Adding Epsilon

The epsilon factor determines how greedy the agent is. Meaning that a lower value represents an agent that will be more likely to make an action that gives the highest reward, rather than taking a random action.

This concept allows us to predict that a more greedy agent will make the best actions because it will be more reward oriented, and on the other hand, a less greedy agent will most likely take random actions which won't be as rewarding and may not be most efficient at learning.

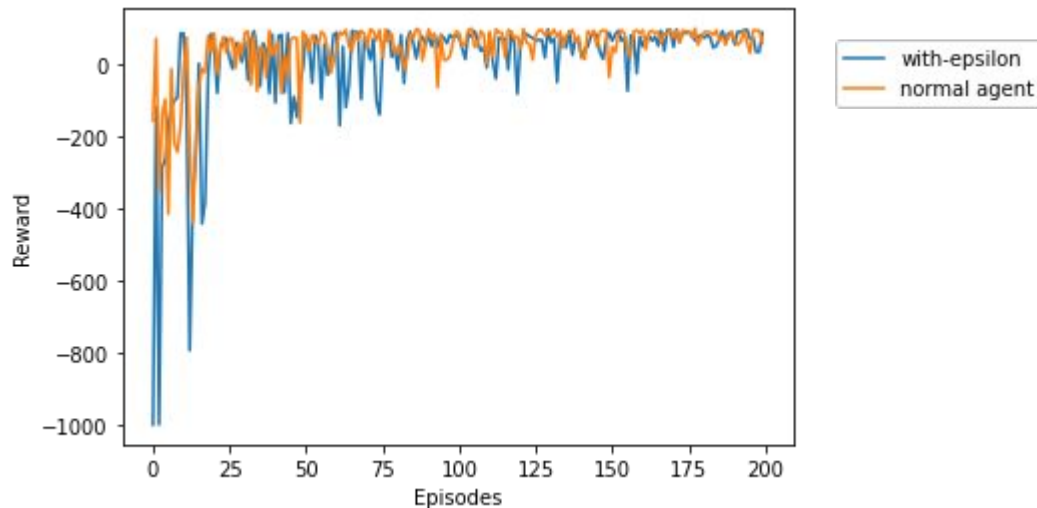
However, taking random actions allows the agent to explore the environment for possible strategies that are more advantageous.

Epsilon = 0.1



Since this epsilon value is low, this means that the agent is greedy and will take a random action 10% of the time. This is why this agent is seen to perform slightly worse than an agent with an epsilon of 0, but still performs pretty well in the bigger picture. In the graph, the agent with an epsilon factor is seen to have these random spikes which represent the episodes where the agent took random moves that were not beneficial.

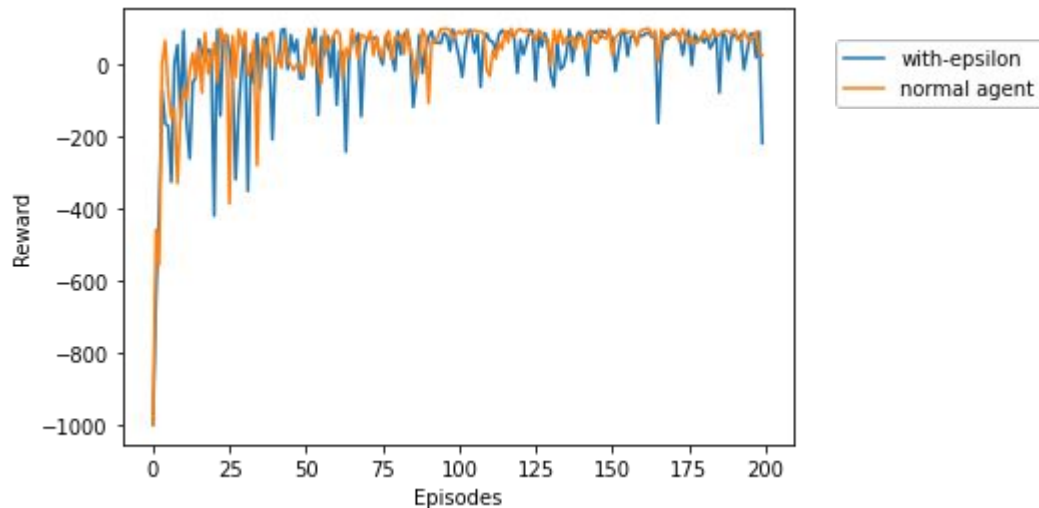
Epsilon = 0.5



With an epsilon of 0.5 this means that 50% of the time a random action is taken. This is an interesting result because it would be expected that the agent would not be able to perform well if half the time it is taking random actions. However, as it can be seen in the graph, the agent performs worse in the first half of the training period than the regular agent, and then ends up performing very similarly to the normal agent later on. This may indicate that the agent discovered a more beneficial strategy through a random action that it took.

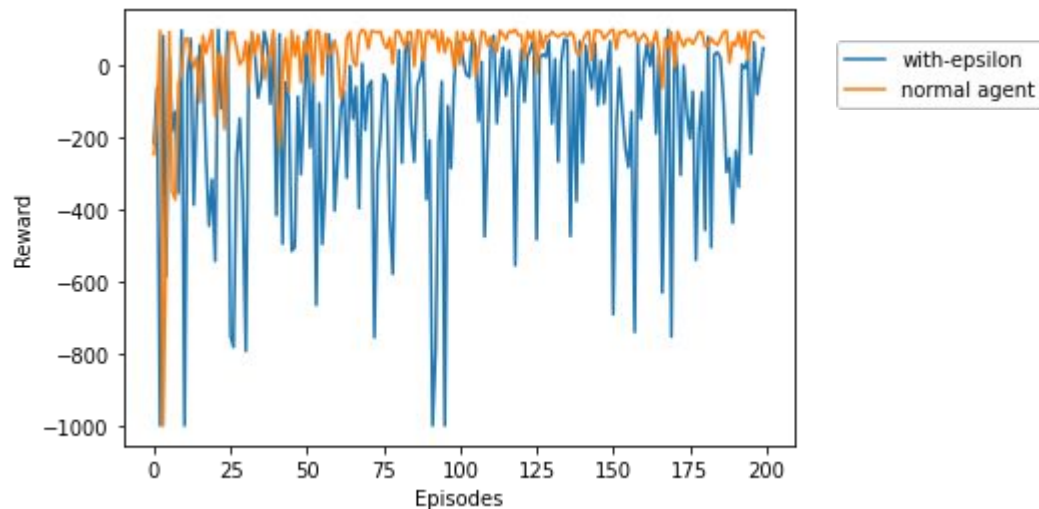


Epsilon = 0.5



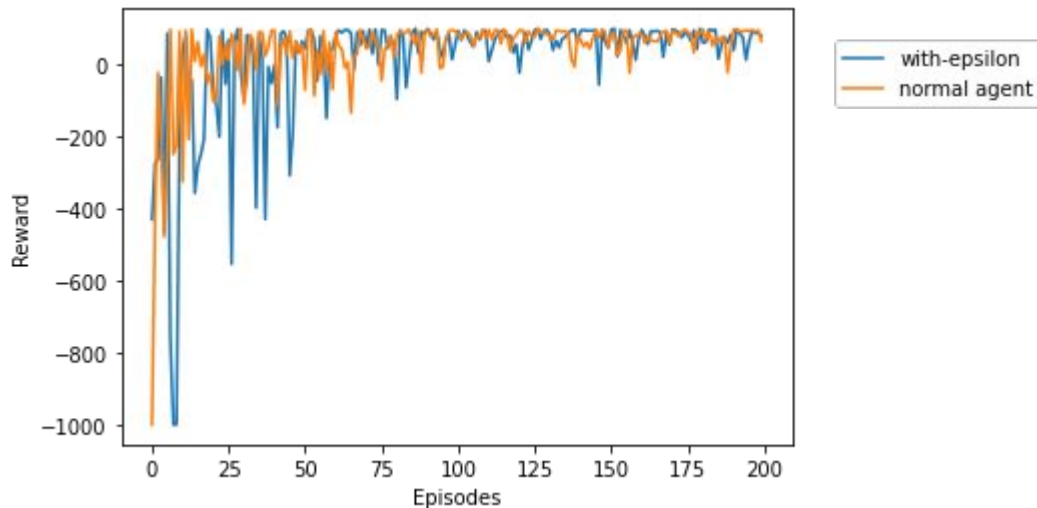
Since the agent with epsilon takes random actions, it's not guaranteed that the random actions it takes will be beneficial. The graph to the left is an example where the agent with epsilon does not really benefit from taking random actions.

Epsilon = 0.9



As expected, a high epsilon value results in the agent taking random actions majority of the time and almost not being greedy for rewards at all. It is clear that an epsilon value over 0.5 is not beneficial, however there are visible benefits for having a low epsilon value, as it allows the agent to discover better strategies.

Epsilon = 0.9 (only for first half of episodes with decay overtime)



The code was changed so that the agent was only affected by epsilon for the first half of the training period. As can be seen, this made a huge difference in the results. In comparison to the previous slide, this time the agent was able to perform significantly better. It is hypothesized that the random movements in the first half, allow the agent to explore the environment which can prove to be beneficial later on when the



What does a Q value represent?

What does the quantity $Q(s,a)$ mean?

How do we get from random Q (initialization) to Q values that actually mean something?



Neural Networks - Universal Function Approximators

- Given some input, learn to produce a desired output
 - Learn by minimizing an error signal
- In DQN, what does each unit in the output layer represent? How do we tune the weights so that we get the desired output?



Neural Networks to Learn Value - DQN

Deep Q networks (DQN) learn to approximate the $Q(s,a)$ for a given input s

Policy is then: take the action which corresponds to the highest Q value for that s



Neural Networks to Learn Policies

- Neural networks can be trained to learn any function
- Policy is a function of state
- Neural network can learn a policy more complicated than just “select highest value action”
- Depends on having a good loss function (i.e. measurement of error)
 - What is the target for learning a good policy?



Neural Networks to Learn BOTH Value and Policy

- These are called “actor-critic” architectures:
 - An “actor” learns a good policy (i.e. how to act)
 - A “critic” learns an estimate of value (i.e. how good/bad a state or state/action pair is)



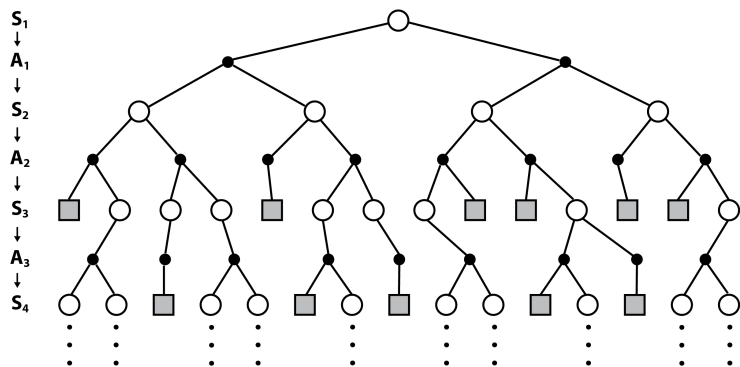
Comparing Different Learning Styles

In the same environment, try:

- Tabular Q learning
- Deep Q network
- Policy network
- Actor critic network

Which works best? Does this change in a different type of environment? Why do you think this happens?

- Control: Bellman optimality equations



Bellman expectation equations:

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s'} \mathcal{P}_{ss'}^a (\mathcal{R}_{ss'}^a + \gamma v_{\pi}(s'))$$

$$q_{\pi}(s, a) = \sum_{s'} \mathcal{P}_{ss'}^a \left[\mathcal{R}_{ss'}^a + \gamma \sum_{a'} \pi(s', a') q_{\pi}(s', a') \right]$$

Bellman optimality equations:

$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

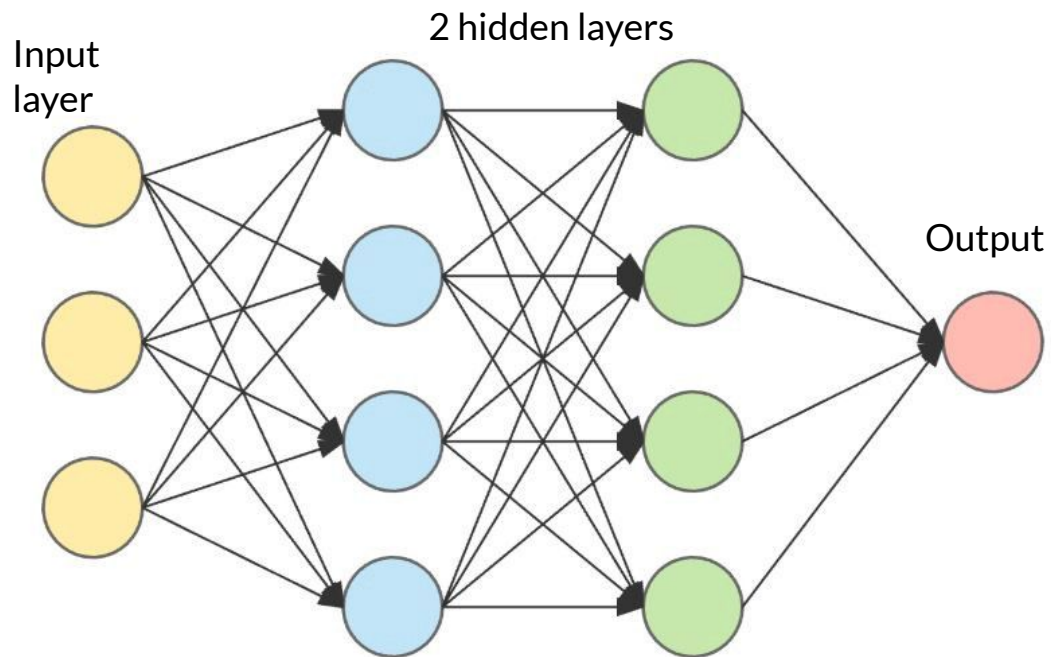
$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a)$$



What is a Neural Network

- Neural networks consist of a large number of nodes which are densely interconnected. They are organized into layers and data moves through them in a single direction, “forward”. Each node has a weight associated with it. When data arrives at a node from the hundreds of connected nodes from the layer beneath, it is multiplied by the weight and then summed together. If this value is greater than the threshold value, data is passed to the next layer.
- At the beginning of training periods, all the weights and threshold values are set randomly. As the agent trains, the weights are adjusted depending on how well the network performed and whether the output of the network was correct or far from the desired output.

Neural Network



As can be seen in this diagram, input is sent forward through the layers of the network. After the data has been processed an output is given. According to how accurate the output is, the network adjusts the weights. It keeps doing this to train itself, until a point is reached where the weights have reached the most optimum value.

The Deep-Q network class

```

1 class DQN(nn.Module):
2     def __init__(self, input_dims, fc1_dims, fc2_dims, n_actions, lr):
3         super(DQN, self).__init__()
4         self.input_dims = input_dims
5         self.fc1_dims = fc1_dims
6         self.fc2_dims = fc2_dims
7         self.output_dims = n_actions
8
9         self.fc1 = nn.Linear(*self.input_dims, self.fc1_dims) # *inputdims unpacks a list of items
10        self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
11        self.fc3 = nn.Linear(self.fc2_dims, self.output_dims)
12
13        self.lr = lr
14        self.optimizer = optim.Adam(self.parameters(), lr=self.lr)
15        self.loss = nn.MSELoss()
16        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
17
18    def forward(self, state):
19        x = F.relu(self.fc1(state))
20        x = F.relu(self.fc2(x))
21        actions = self.fc3(x)
22
23        return actions

```

In this class, all the layers of the network are connected. In this case there are 2 hidden layers called fc1 and fc2 and there is a layer for input and a layer for output. It is important to note that the dimensions of the matrices must match to allow for multiplication. For example if the input layer is 1x25 the first dimension of the first hidden layer, fc1, must be 25.

Comparing action selection

```
def choose_action(self, observation):
    # epsilon greedy action selection
    if np.random.random() > self.epsilon:
        # take best action
        vector= np.zeros(self.nstates)
        vector[observation]=1
        state = T.Tensor([vector]).to(self.Q_eval.device)
        actions = self.Q_eval(state)
        action = T.argmax(actions).item()
    else:
        action = np.random.choice(self.action_space)

    return action
```

```
def choose_random_action(self, state):
    a=np.random.choice(self.action_space)
    return a

def choose_max_value_action(self, state):
    action= np.argmax(self.q_table[state,:])
    return action

def choose_epsilon_action(self, state):
    if np.random.random()>self.EPSILON:
        action = np.argmax(self.q_table[state])
    else:
        action=np.random.choice(self.action_space)
    return action
```

The main difference between the way actions are selected by an agent using a neural network and an agent using q-learning is that the q-learning agent uses a table of q-values to make the decision based on max value, while the neural network agent passes the current observation (state) through the network and then receives q-values to allow it to make the decision based on max value. Storing q-values in a table takes up a lot of space, which makes neural networks a better option when dealing with larger environments. In neural networks, simply the weights are adjusted to output better q-values.



Loss

- With neural networks, we aim to minimize the error.
- The loss function calculates a value known as “loss” and this value tells us how well the network performed with the current weights.
- In other words it represents the error in the model and this allows the agent to learn and update the weights based on how much error there was in the model.
- A better model of weights is represented by a lower loss value.

```
loss = self.Q_eval.loss(q_target, q_eval).to(self.Q_eval.device)
loss.backward() # backprop the error
self.Q_eval.optimizer.step() #weight update
```

SARSA

- State Action Reward State(new) Action(new)
- Very similar to the q-learning agent from before
- Q values are updated slightly differently
 - Sarsa updates q values acknowledging the policy that will be used to select actions
 - Q agent uses the max future value to update q values which does not represent the epsilon-greedy policy that will be used during action selection

```
def q_update(self, current_state, current_action, reward, new_state):
    current_q = self.q_table[ current_state, current_action]
    max_future_q = np.max(self.q_table[new_state,:])

    new_q = (1-self.LEARNING_RATE)*current_q + self.LEARNING_RATE*(reward + self.DISCOUNT*max_future_q)
    self.q_table[current_state, current_action] = new_q
```

SARSA q-update

```
def q_update(self, current_state, current_action, reward, next_state, next_action):
    current_q = self.q_table[ current_state, current_action]
    future_q = self.q_table[ next_state, next_action] # np.max(self.q_table[new_state,:])

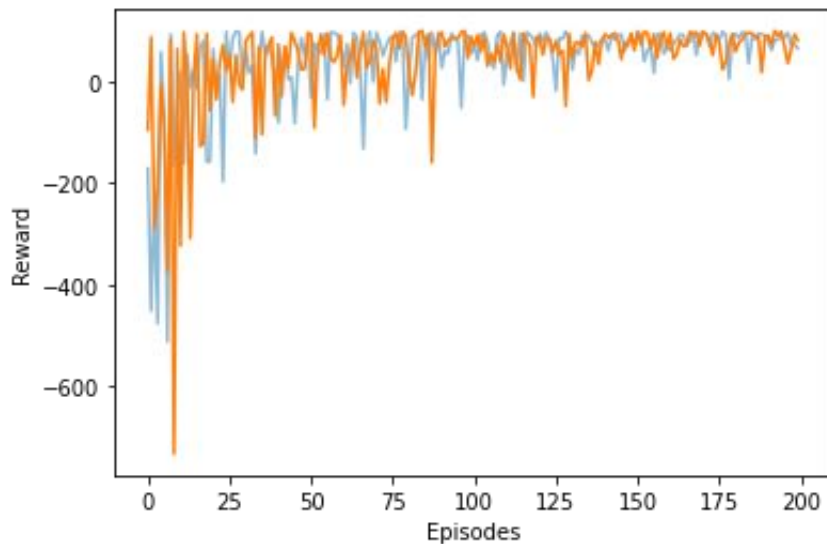
    new_q = (1-self.LEARNING_RATE)*current_q + self.LEARNING_RATE*(reward + self.DISCOUNT*future_q)
    self.q_table[current_state, current_action] = new_q
```



Difference between SARSA agent and Q-agent

- Sarsa is an *on-policy* learning algorithm, and Q-learning is an *off-policy* learning algorithm.
- this means that Sarsa and Q-learning converge to different solutions / "optimal" policies
- While SARSA and Q agents both choose actions using the same policy, the way how the Q values are updated is different.
- Q agents use the max future value to update their Q tables, while SARSA agents follow the same epsilon-greedy policy that is used to select actions to update the Q values.
- The effects of this are only visible in certain environments.

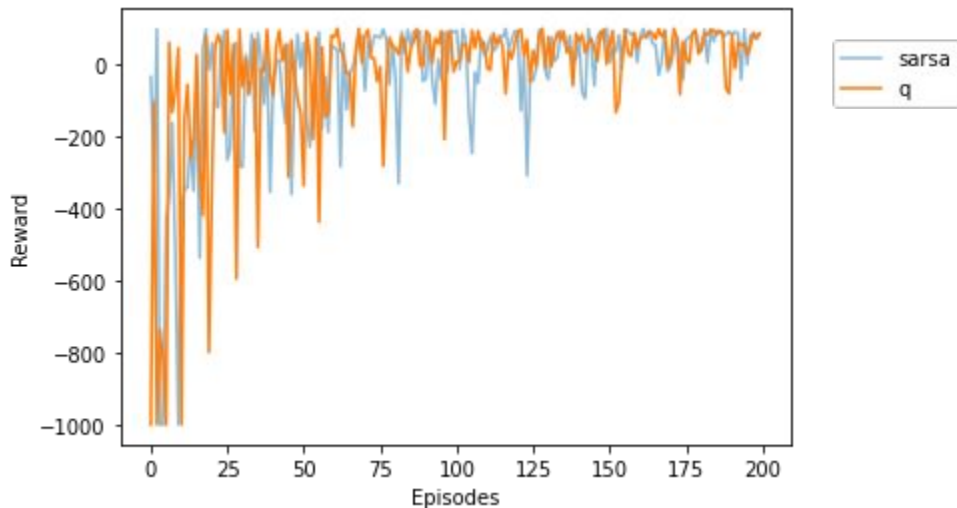
Comparing performance in the edge-wrap environment



sarsa
q

Both agents perform very similarly in the regular edge-wrap environment. Therefore there is no benefit of using either agent over the other in this type of environment.

Comparing performance in the static environment




Both agents perform very similarly in the regular static environment as well. Therefore there is no benefit of using either agent over the other in this type of environment.



When should SARSA be used?

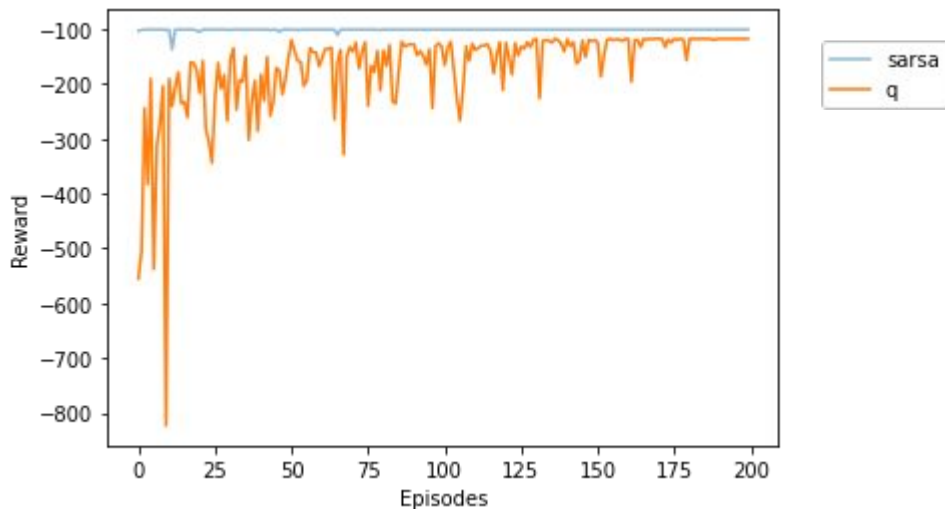
- An algorithm like Sarsa is typically preferable in situations where we care about the agent's performance during the process of learning / generating experience
- For example if the agent was an expensive robot that is being trained near a cliff, we don't want it to fall and break.
- Since we care about the agents performance during the training process, and acknowledge that we will be using an epsilon-greedy policy, we don't want the agent to be close to the cliff since at random times the agent can fall off. We want it to quickly learn that it's dangerous to be close to the cliff
- A Q agent however, would be used in situations where performance doesn't matter in the learning process, and some loss/damage due to randomness is ok.
- A Sarsa agent will reach a solution that is optimal for the particular policy that is being used to select actions, while the q agent reaches a solution assuming that the greedy policy will be used to select actions instead of epsilon-greedy.
- It can be predicted that a q- agent would perform much worse in the learning phase than the sarsa agent.

Creating the cliff environment

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	
Start											+100	End

In this environment there is a cliff state which gives a huge penalty for moving into it, and it also directs the agent back to the start. It is predicted that the Q - agent will try to take the most fastest path right along the cliff, however since the q updates did not account for the epsilon-greedy policy, the agent faces large penalties for everytime it falls off the cliff. On the other hand, the sarsa agent takes a safer path and reaches more optimum rewards.

Comparing performance in the cliff environment

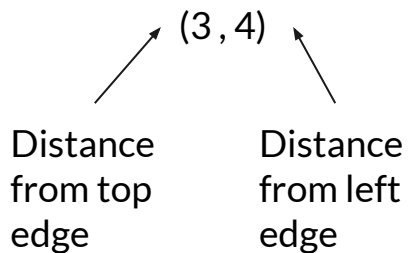


The results for this experiment are very interesting and show clear evidence that in this type of environment the SARSA agent performs significantly better. As can be seen in the graph to the left, the SARSA agent remains steady throughout the training period while the regular agent takes much longer to optimize itself, and is still unable to do better than the SARSA agent. After episode 175 it is visible that although both agents reached stability, the SARSA agent was able to find a much more optimum route. It did so really quickly as well while the q-agent couldn't get there despite taking a longer time to reach stability. This mainly has to do with the difference in how q values are updated.



State features

- The agent can have information about the exact distance it is from the edges of the grid.
- This can create a coordinate system that allows the agent to know exactly where it is in the environment and where it needs to be to reach the terminal state.

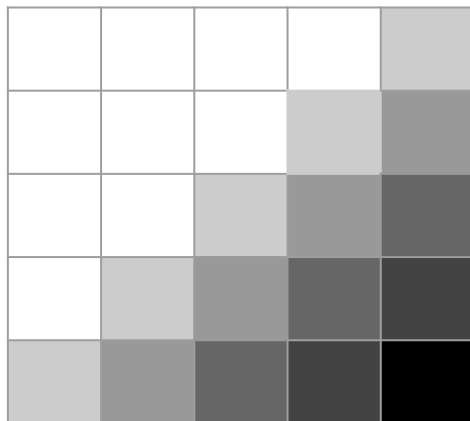


(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)



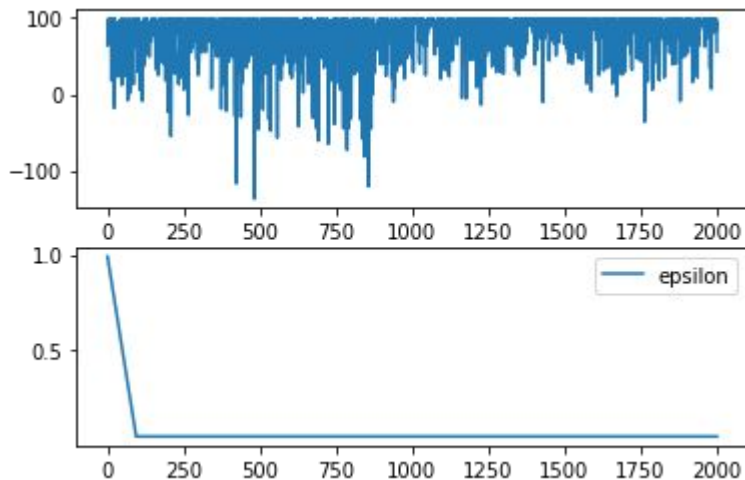
State features

- The environment can have a color gradient which becomes most dense at states closest to the terminal state and starts fading away as it gets further away from the terminal state.



← Terminal state

Coordinate representation results (Parameter tuning)



Env(5,5)

Learning rate = 0.002

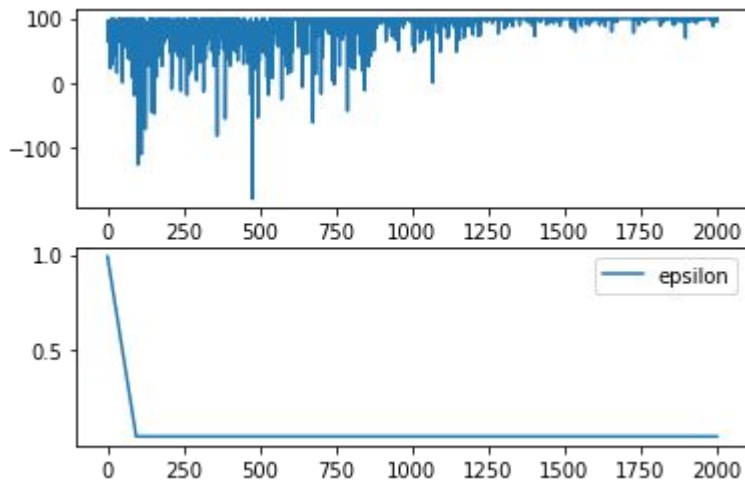
Discount = 0.99

Hidden layers = 50, 10

2000 episodes , 300 max steps

Epsilon end = 0.05

Coordinate representation results (Parameter tuning)



Env(5,5)

Start at 0

Learning rate = 0.002

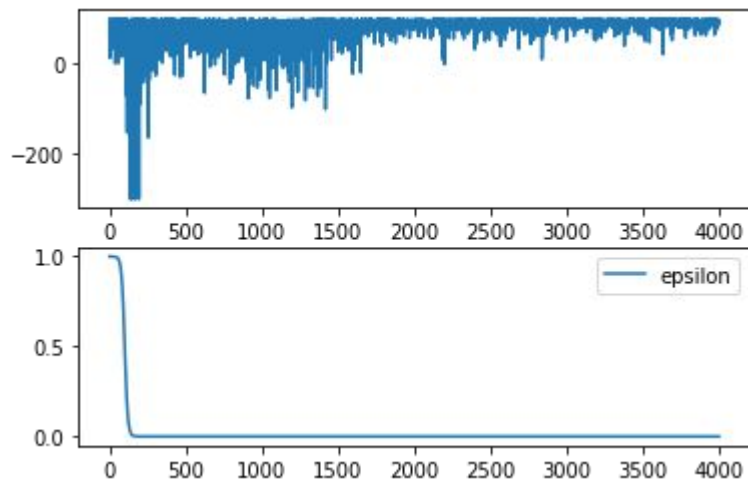
Discount = 0.99

Hidden layers = 100, 50

2000 episodes , 300 max steps

Epsilon end = 0.05

Coordinate representation results (Parameter tuning)



Env(5,5)

Random start

Learning rate = 0.0005

Discount = 0.99

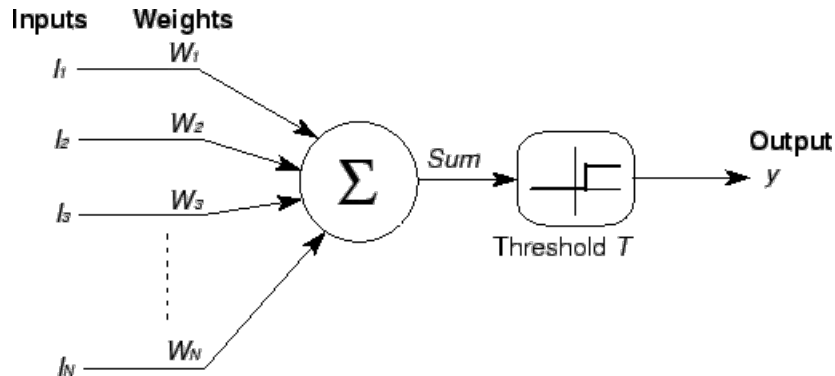
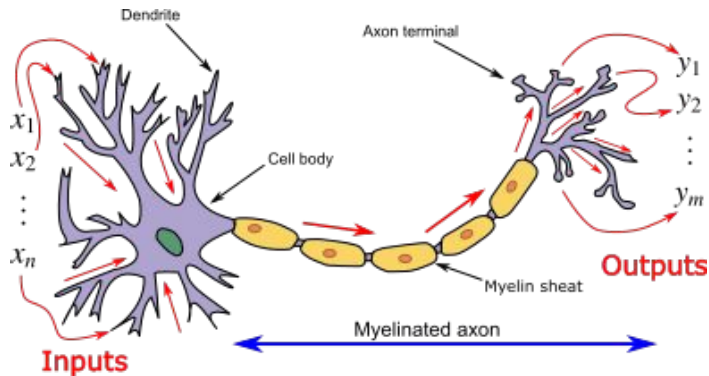
Hidden layers = 100, 50

2000 episodes , 300 max steps

Epsilon end = 0.00001

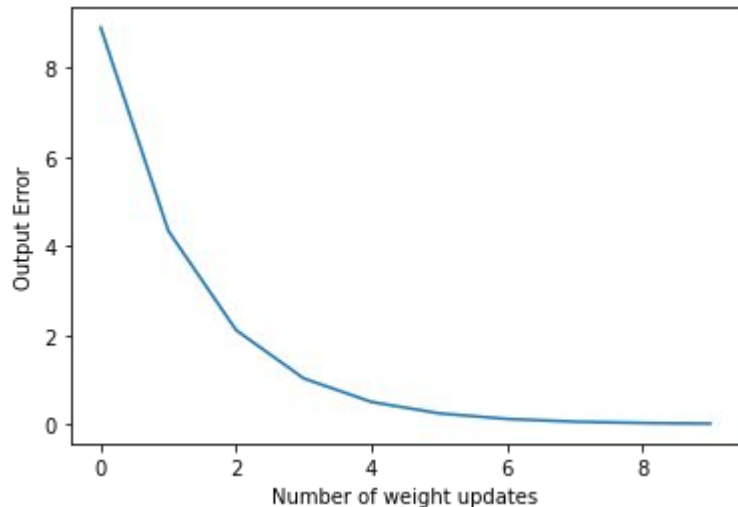
Perceptrons

Perceptrons are basically artificial neurons. The structure of a perceptron is inspired by the neuron. The inputs and weights are analogous to the dendrites and synapses of the neuron. In addition, similar to the neuron, the output can only be sent to the next unit if a certain threshold value is passed.





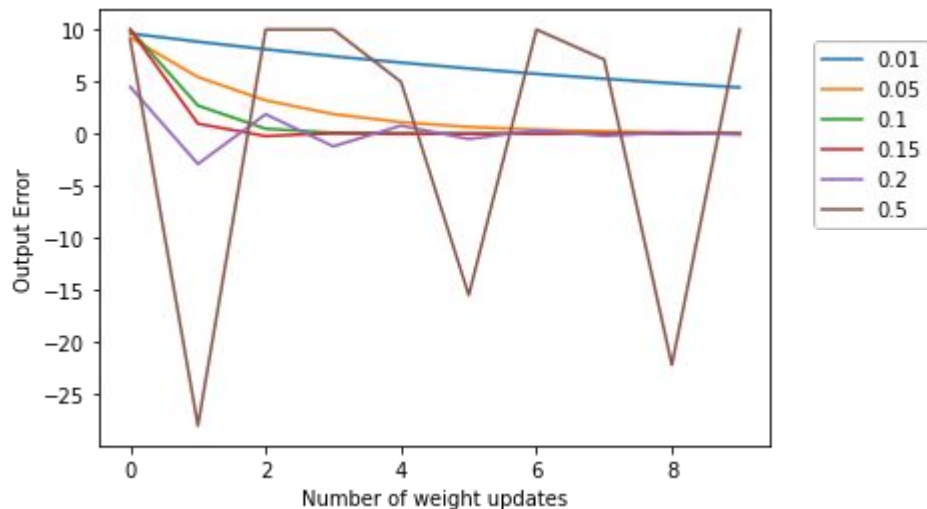
Perceptron Learning algorithm



The way how the perceptron learning algorithm works is by reducing the amount of error in each trial by updating the weights according to the amount of error resulting from the current set of weights. The graph to the left portrays the results of how learning occurs in this type of learning algorithm. It shows how the output error gets lower and lower with each weight update to achieve the target value.

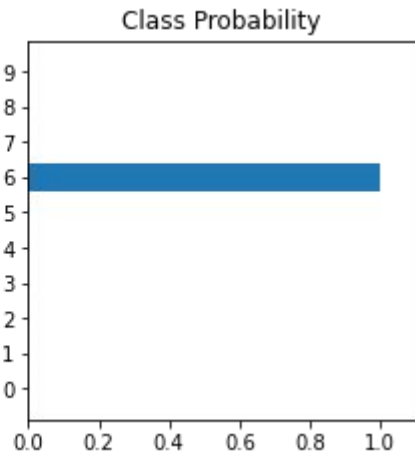
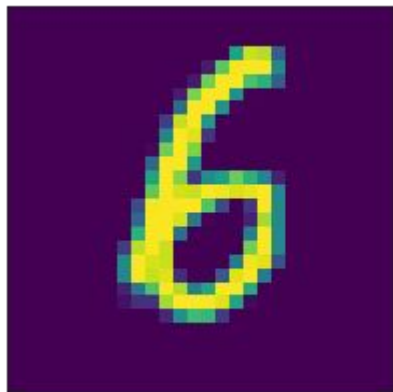
The algorithm updates the weights by multiplying the learning rate by the error times the inputs plus the current weights.

Effects of Learning rate on performance



As can be seen in the graph to the left, the learning rate must be in the goldilocks zone where it is not too high or too low. When the rate is too low it takes much longer for the algorithm to converge. When the learning rate is too high, the algorithm overshoots and is not able to converge since the step size is too large. For this particular example the learning rate of 0.15 seems to be the optimal value which allows it to converge to the target most efficiently.

Using neural networks to identify images



- Aside from the grid environment tasks, we can also use neural networks to identify objects from images.
- In the example to the left, an agent was trained with multiple images of handwritten digits. Over time, the network's weights were optimized to reach a state where it was almost 98% accurate when tested with 10,000 images.
- In the diagram we can see the image that was input as well as the output from the network of the probability of which digit it is.



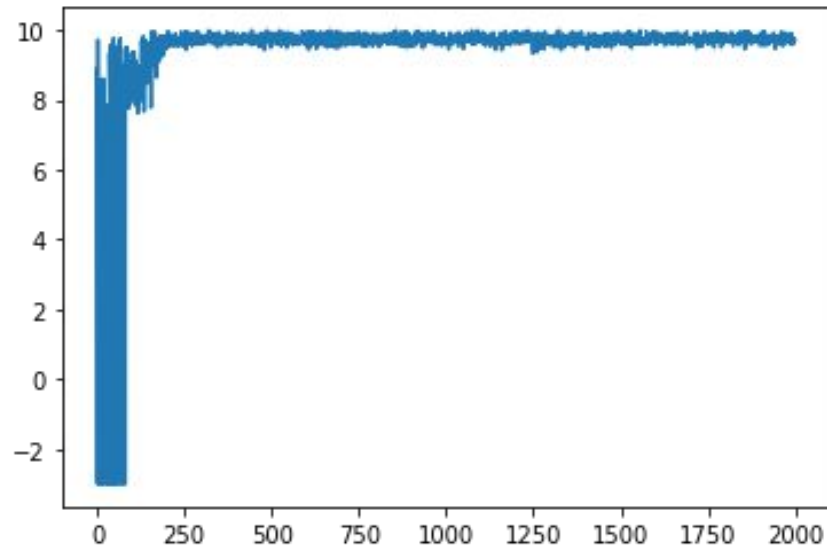
Actor - Critic Networks

An actor-critic network is a merge between policy based and value based reinforcement learning methods.

- Actor: decides which action to take (policy based)
- Critic : provides feedback to actor of how good the action was and how it should adjust. (value based)

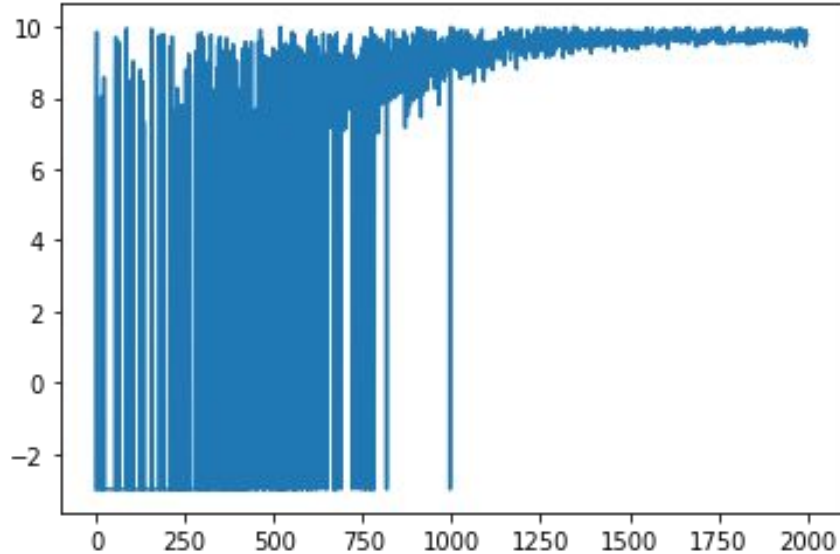
The actor takes as input the state and outputs the best action. It essentially controls how the agent behaves by learning the optimal policy (policy-based). The critic, on the other hand, evaluates the action by computing the value function (value based). Those two models participate in a game where they both get better in their own role as the time passes. The result is that the overall architecture will learn to play the game more efficiently than the two methods separately.

Results



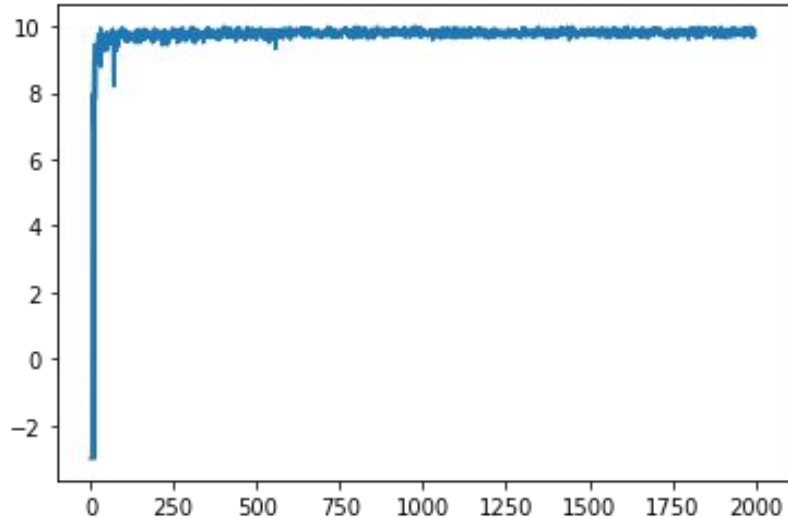
In this 20x20 grid environment the actor-critic network performed really well and was quickly able to stabilize to achieve high scores in each episode. In the first 100 episodes it is visible that the agent was exploring the environment and did not achieve the optimal set of weights yet to make the best decisions. However, at around 200 episodes there is a quick improvement in performance and the agent seems to have achieved the optimal state and continues to perform consistently throughout the experiment.

Lower Learning rate



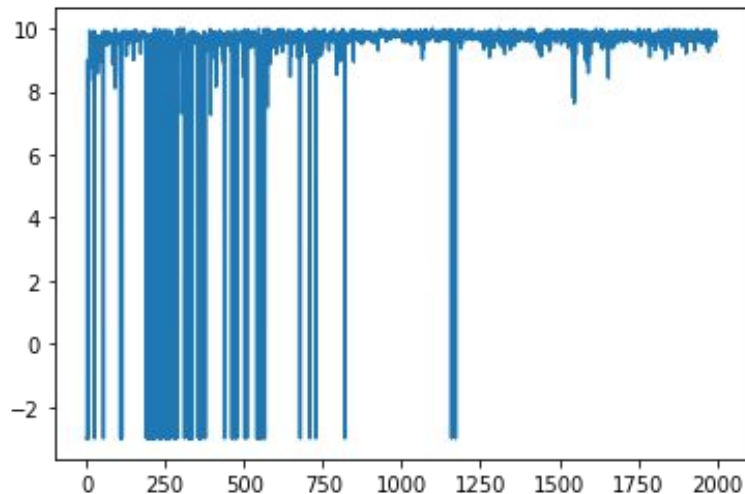
When the learning rate is decreased by a factor of 10 it is clearly visible that it takes longer for the agent to reach the optimal state, however is still able to stabilize within 2000 episodes.

Higher Learning rate



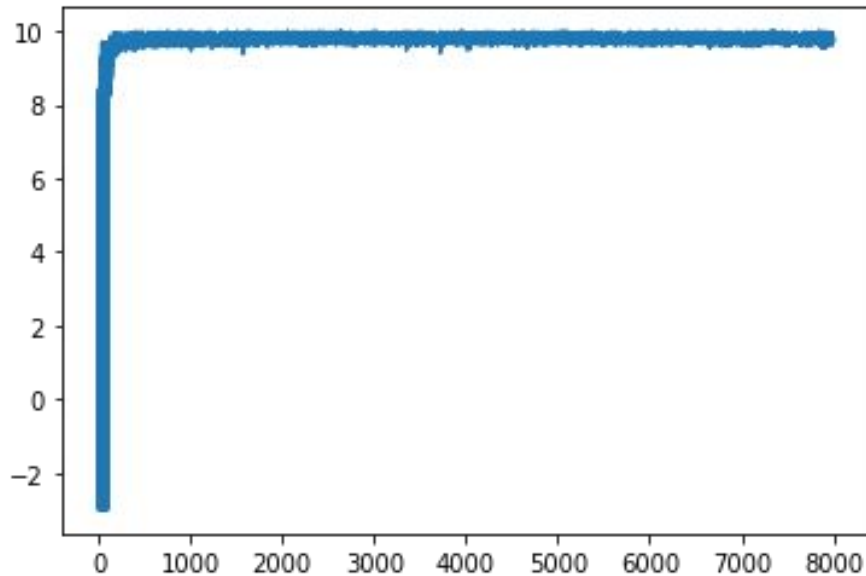
When the learning rate is increased by a factor of 10 it is clearly visible that it takes much less time for the agent to reach the optimality.

Very high Learning rate



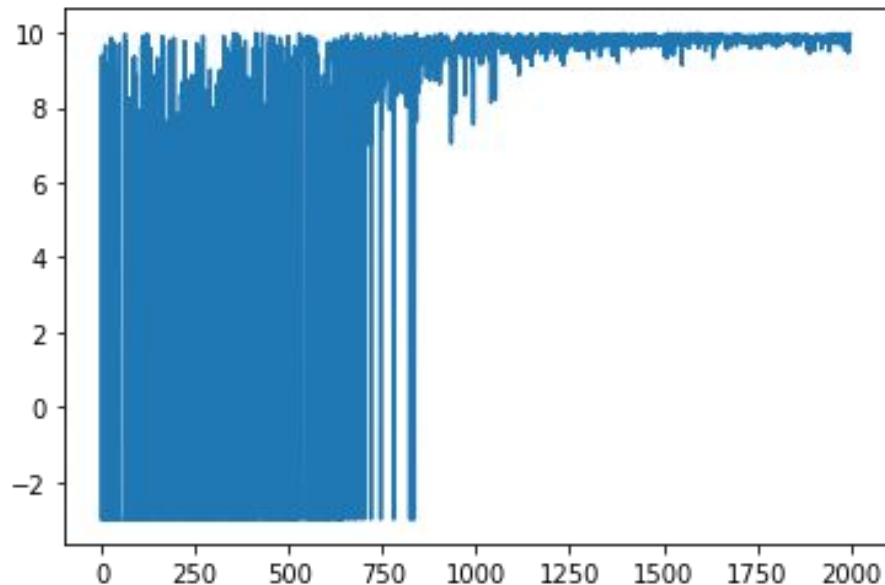
Learning rate was increased by a factor of 10 again. As expected when the learning rate is too high the agent begins to overshoot the target since the weights are updated drastically with each update. This causes the network to not be precise enough to give the best decision.

Reducing Discount (0.5)



There is no visible effect of reducing the discount factor to 0.5

Changing Terminal state



When the terminal state is changed to a position at the center of the grid, the results are different. It now takes longer for the agent to find the optimal path. Since this is an edge wrapping environment, it makes it much more likely for the agent to discover an easy shortcut to reach the terminal state. When the terminal state is in the center of the grid, it takes longer for the agent to learn the optimal path since the likelihood of the agent discovering a shortcut are low.