Venkat Mohit Sornapudi - Student nr: 2721697
Laréb Fatima Ahmad - Student nr: 2737338

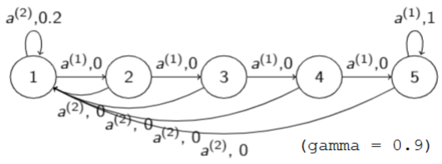
12-14-2021

Dynamic Programming and Reinforcement Learning

Assignment 4

Part 1

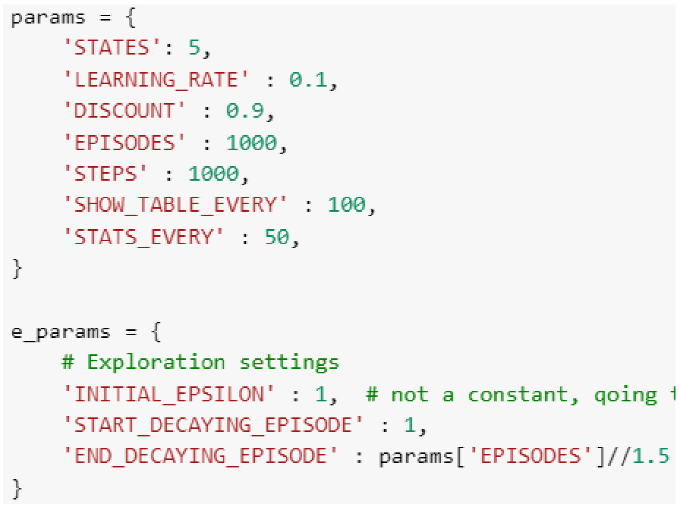
Implementation



For part 1, we consider The Chain Problem depicted above. We examine 5 and 10 states, and 2 actions. The action taken decides the next state. The discount value **γ** is 0.9. The found q-values are stored in a matrix of size states x actions (“q-table”).

In this section we explain the main program for solving the q-values: the q-learning function.

The q-learning function takes the following parameters:



```
params = {
    'STATES': 5,
    'LEARNING_RATE' : 0.1,
    'DISCOUNT' : 0.9,
    'EPISODES' : 1000,
    'STEPS' : 1000,
    'SHOW_TABLE_EVERY' : 100,
    'STATS_EVERY' : 50,
}

e_params = {
    # Exploration settings
    'INITIAL_EPSILON' : 1,  # not a constant, qoing t
    'START_DECAYING_EPISODE' : 1,
    'END_DECAYING_EPISODE' : params['EPISODES']//1.5
}
```

We define in “params” no. of states, learning-rate **α**, discount-value **γ**, total episodes to run, total steps in each episode, when to show the q-value table (e.g.: every 100th episode), and when to show the stats (e.g.: every 50th episode). Stats consist of episode number, reward in that episode, and ∈ in that episode. An *episode* is an iteration which contains *steps* in which the q-table is updated.

“e-params” define parameters that will change value in the course of the program run: initial ∈ and the range in which ∈ decays.

Each episode starts in state 0. The next state is decided by the action taken in the current state. In order to find the next state, we created a helper-function which does the following:

```
def take_step(state, action)
  if action == 1:
      new_state = 0
      if state == 0:
        reward = 0.2
      else:
        reward = 0
  else:
    if state == states[-1]:
      new_state = state
      reward = 1
    else:
      new_state = state+1
      reward = 0
  return new_state, reward
```

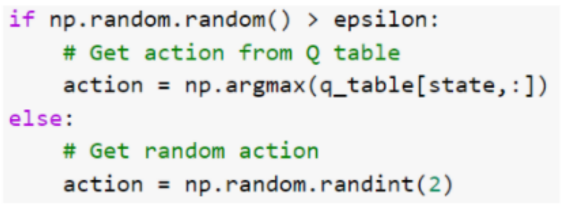
Given any state, if action = 0, the next state is always 0. If the given state is 0, then reward = 0.2, otherwise reward = 0. If action = 1, and the given state is the last state in the chain, then state remains unchanged, and reward = 1. If state = any other state, it transitions to the next one with a reward of 0.

$$Q(x_t, a_t) \leftarrow Q(x_t, a_t) + \alpha_t[r_t + \gamma \max_{a' \in \mathcal{A}} Q_t(x_{t+1}, a') - Q_t(x_t, a_t)],$$

```
# And here's our equation for a new Q value for current state and action
new_q = (1 - params['LEARNING_RATE']) * current_q + params['LEARNING_RATE'] * (reward + params['DISCOUNT'] * max_future_q)
```

The above figure displays the q-value formula and its implementation. **α** and **γ** are retrieved from the parameters to the q-learning function. The current q-value is taken from the q-table for current state and action. The maximum next q-value ‘max_future_q’ is the max value from the q-table corresponding to the *next* state.

In the solution, action is chosen using an ∈-greedy algorithm. This is conducted by taking a random value from the range [0,1] and comparing it to ∈. Optimal action (action with highest q value corresponding to the state) is chosen from the q-table if said randomly value is > ∈, otherwise a random action is taken. This is shown in the figure below. At the end of each episode, ∈ is decayed if the episode number is within the decaying range. This range is explained in the results section.

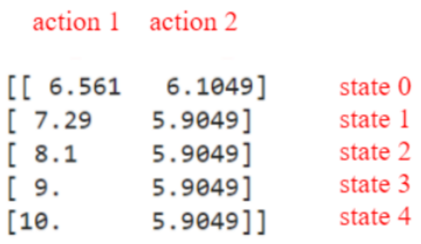


```python
if np.random.random() > epsilon:
    # Get action from Q table
    action = np.argmax(q_table[state,:])
else:
    # Get random action
    action = np.random.randint(2)
```

Figure: ∈-greedy algorithm

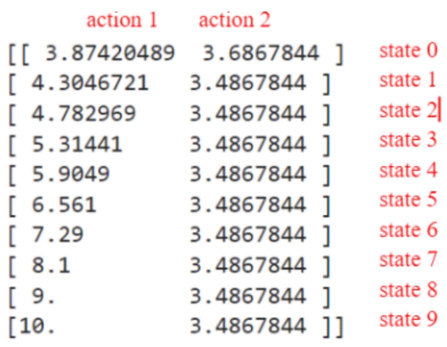
Results

When this part of the q-value-formula $[r_t + \gamma \max_{a' \in \mathcal{A}} Q_t(x_{t+1}, a') - Q_t(x_t, a_t)]$, goes to 0, the formula essentially says: q-value + 0. This yields the fixed q-values shown below. The last 5 q-values for 10 states are the same as the q-values for 5 states.



```
     action 1   action 2

[[ 6.561    6.1049]      state 0
 [ 7.29     5.9049]      state 1
 [ 8.1      5.9049]      state 2
 [ 9.       5.9049]      state 3
 [10.       5.9049]]     state 4
```

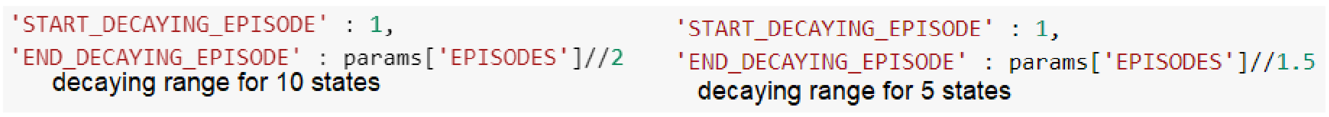
Fixed q-values for 5 states



```
         action 1     action 2
[[ 3.87420489  3.6867844 ]     state 0
 [ 4.3046721   3.4867844 ]     state 1
 [ 4.782969    3.4867844 ]     state 2
 [ 5.31441     3.4867844 ]     state 3
 [ 5.9049      3.4867844 ]     state 4
 [ 6.561       3.4867844 ]     state 5
 [ 7.29        3.4867844 ]     state 6
 [ 8.1         3.4867844 ]     state 7
 [ 9.          3.4867844 ]     state 8
 [10.          3.4867844 ]]    state 9
```

Fixed q-values for 10 states.

Experiments



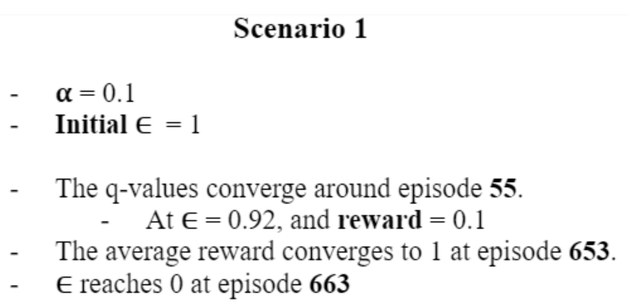
```python
'START_DECAYING_EPISODE' : 1,
'END_DECAYING_EPISODE' : params['EPISODES']//2
```

decaying range for 10 states

```python
'START_DECAYING_EPISODE' : 1,
'END_DECAYING_EPISODE' : params['EPISODES']//1.5
```

decaying range for 5 states

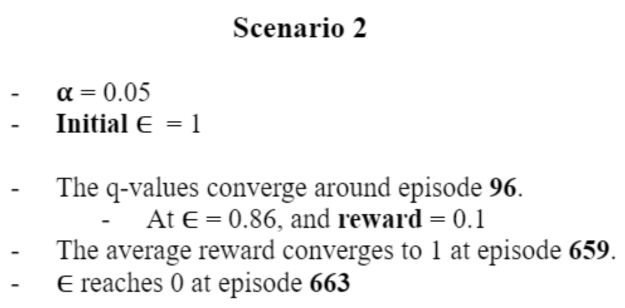
To discuss ∈ and **α**, we experimented with varying ∈ and **α** values. The figure above shows the decaying ranges (closed intervals), which were chosen as they deemed best fit after some experimenting with the values. We run the program for 1000 steps in 1000 episodes, and track the rewards, q-values and ∈. The following table shows the results given **5 states**. The table with each scenario's corresponding reward-plots in each episode can be seen in the appendix.



Scenario 1

- **α** = 0.1
- **Initial** ∈ = 1

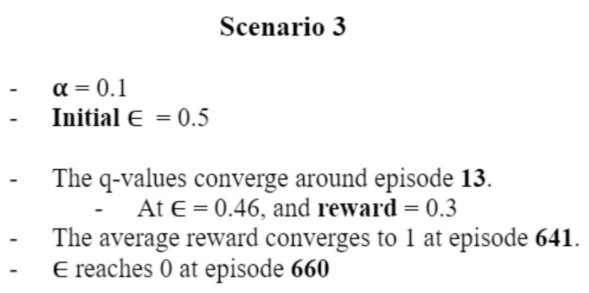
- The q-values converge around episode **55**.
  - At ∈ = 0.92, and **reward** = 0.1
- The average reward converges to 1 at episode **653**.
- ∈ reaches 0 at episode **663**



Scenario 2

- **α** = 0.05
- **Initial** ∈ = 1

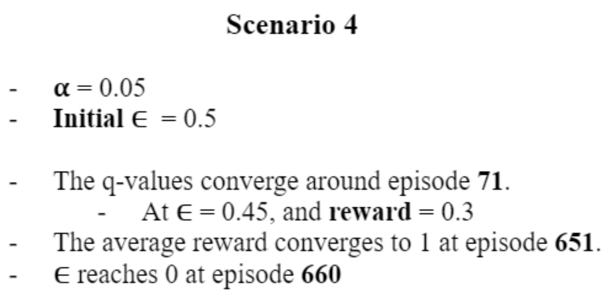
- The q-values converge around episode **96**.
  - At ∈ = 0.86, and **reward** = 0.1
- The average reward converges to 1 at episode **659**.
- ∈ reaches 0 at episode **663**



Scenario 3

- **α** = 0.1
- **Initial** ∈ = 0.5

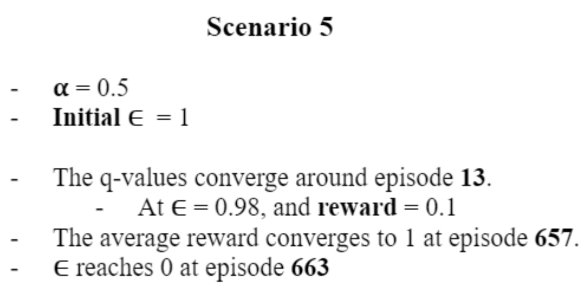
- The q-values converge around episode **13**.
  - At ∈ = 0.46, and **reward** = 0.3
- The average reward converges to 1 at episode **641**.
- ∈ reaches 0 at episode **660**



Scenario 4

- **α** = 0.05
- **Initial** ∈ = 0.5

- The q-values converge around episode **71**.
  - At ∈ = 0.45, and **reward** = 0.3
- The average reward converges to 1 at episode **651**.
- ∈ reaches 0 at episode **660**



Scenario 5

- **α** = 0.5
- **Initial** ∈ = 1

- The q-values converge around episode **13**.
  - At ∈ = 0.98, and **reward** = 0.1
- The average reward converges to 1 at episode **657**.
- ∈ reaches 0 at episode **663**

Observations

From the experiments we make the following observations about ∈ and **α**:

1. The initial ∈-value affects when the q-values converge. When initial ∈ = 1, in scenarios 1,2 and 5 we observe that the convergence happens when the reward is 0.1. In scenarios 3 and 4 it is when the reward is 0.3. This is regardless of the learning rate.
2. The learning rate **α** affects when the q-values converge. In scenario 1, 2 and 5, ∈ is 1 while **α** are 0.1, 0.05 and 0.5, respectively. We observe that the higher the **α**, the less episodes it takes to reach convergence.

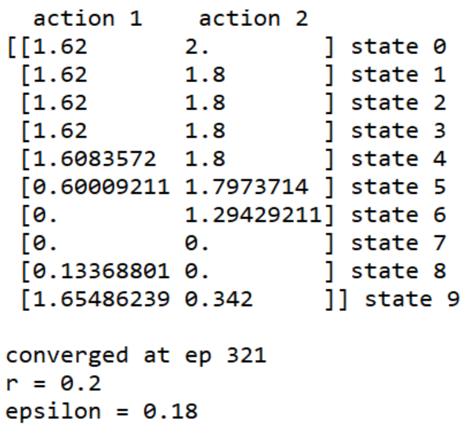
Results and observations for 10 states

Running the same scenarios with **10 states** yielded somewhat different results. The obseravtions still hold for scenrios 1, 2 and 5. The q-values in scenario 2 only converged after increasing the steps and episodes. Generally, the q-values converge later, while reward and ∈ converge earlier, with 10 states.

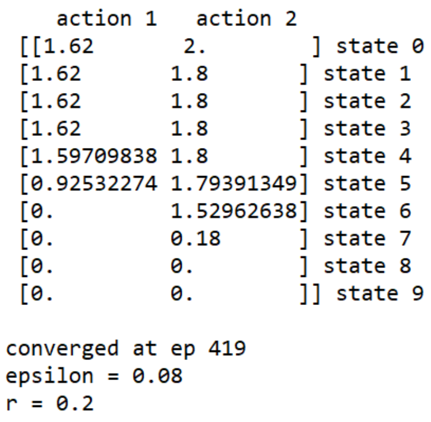
Scenario 1	**Scenario 2**	**Scenario 3**	**Scenario 4**
Q-values converge in 1000 eps 1000 steps, around **episode 219** with r = 0.0, ∈= 0.57	Q-values converge in **2000** steps in **2000** episodes, around **episode 200**, with r = 0.1 and ∈ = 0.56	Q-values converge in 1000 eps 1000 steps, around **episode 220** with r = 0.1, ∈ = 0.56	Q-values converge in 1000 steps in 1000 eps, around **episode 407,** with r = 0.2, ∈ = 0.09

In scenarios 3 and 4, where initial ∈ = 0.5, the observations do not hold. In these scenarios, the q-values converge to slightly different values each time at different episodes, while reward never reaches 1, regardless of increased steps/episodes. The q-values in these scenarios are shown below. After further experimenting with the ∈-values, we can conclude that when ∈ < 1, q-values are different and max reward = 0.2.

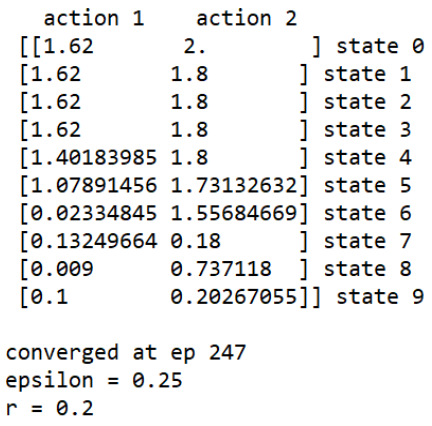
Q-values in scenario 3:



```
  action 1    action 2
[[1.62        2.        ] state 0
 [1.62        1.8       ] state 1
 [1.62        1.8       ] state 2
 [1.62        1.8       ] state 3
 [1.6083572   1.8       ] state 4
 [0.60009211 1.7973714 ] state 5
 [0.          1.29429211] state 6
 [0.          0.        ] state 7
 [0.13368801 0.         ] state 8
 [1.65486239 0.342     ]] state 9

converged at ep 321
r = 0.2
epsilon = 0.18
```

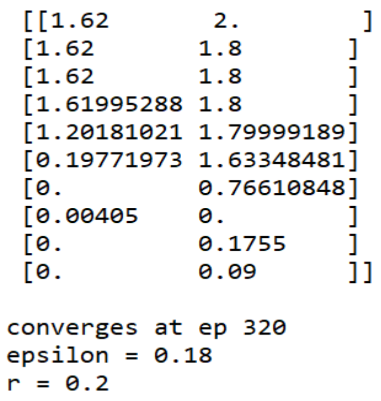


```
  action 1   action 2
[[1.62        2.        ] state 0
 [1.62        1.8       ] state 1
 [1.62        1.8       ] state 2
 [1.62        1.8       ] state 3
 [1.59709838 1.8        ] state 4
 [0.92532274 1.79391349] state 5
 [0.          1.52962638] state 6
 [0.          0.18      ] state 7
 [0.          0.        ] state 8
 [0.          0.        ]] state 9

converged at ep 419
epsilon = 0.08
r = 0.2
```

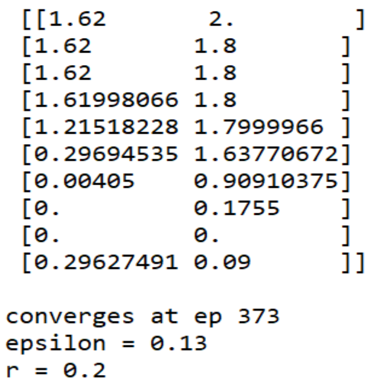


```
  action 1    action 2
[[1.62        2.        ] state 0
 [1.62        1.8       ] state 1
 [1.62        1.8       ] state 2
 [1.62        1.8       ] state 3
 [1.40183985 1.8        ] state 4
 [1.07891456 1.73132632] state 5
 [0.02334845 1.55684669] state 6
 [0.13249664 0.18      ] state 7
 [0.009       0.737118  ] state 8
 [0.1         0.20267055]] state 9

converged at ep 247
epsilon = 0.25
r = 0.2
```

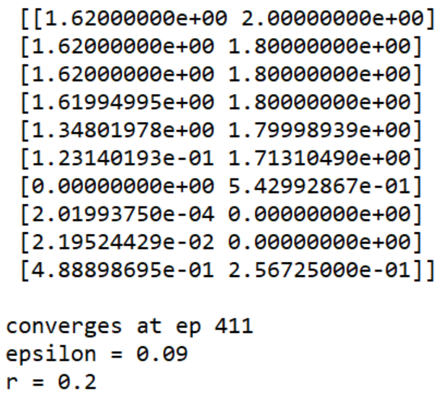
Q-values in scenario 4:



```
[[1.62        2.        ]
 [1.62        1.8       ]
 [1.62        1.8       ]
 [1.61995288 1.8        ]
 [1.20181021 1.79999189]
 [0.19771973 1.63348481]
 [0.          0.76610848]
 [0.00405     0.        ]
 [0.          0.1755    ]
 [0.          0.09      ]]

converges at ep 320
epsilon = 0.18
r = 0.2
```



```
[[1.62        2.        ]
 [1.62        1.8       ]
 [1.62        1.8       ]
 [1.61998066 1.8        ]
 [1.21518228 1.7999966 ]
 [0.29694535 1.63770672]
 [0.00405     0.90910375]
 [0.          0.1755    ]
 [0.          0.        ]
 [0.29627491 0.09      ]]

converges at ep 373
epsilon = 0.13
r = 0.2
```



```
[[1.62000000e+00 2.00000000e+00]
 [1.62000000e+00 1.80000000e+00]
 [1.62000000e+00 1.80000000e+00]
 [1.61994995e+00 1.80000000e+00]
 [1.34801978e+00 1.79998939e+00]
 [1.23140193e-01 1.71310490e+00]
 [0.00000000e+00 5.42992867e-01]
 [2.01993750e-04 0.00000000e+00]
 [2.19524429e-02 0.00000000e+00]
 [4.88898695e-01 2.56725000e-01]]

converges at ep 411
epsilon = 0.09
r = 0.2
```

Part 2

Implementation

Part 2 is solved similarly to part 1, with a few differences. Action and next-states are selected identically, but the states now exist in a tensor with the states one-hot encoded.
In this part, we utilize replay memory by using a buffer (“replay_buffer”) to save an “experience” which consists of current state, action, reward and next state. We operate on a base-model to be trained (‘model’) and a target-model (‘target_model’) which takes the base-model’s weights for each gradient update.
We used one-hot-encoding as the span of state-space is known, and neural-networks (NN) work better with one-hot-encodings. Here, states are the input to the NN. Low dimensional state representations (compared to one-hot-encodings) as input to the NN would hurt the performance of the network.

Replay_buffer has a size limit. If it’s not full, current experience is saved to it. Otherwise, first value is replaced by current experience:

```
if len(replay_buffer) < params['REPLAY_MEMORY_SIZE']:
    replay_buffer.append([state, action, reward, new_state])
elif len(replay_buffer) == params['REPLAY_MEMORY_SIZE']:
    del replay_buffer[0]
    replay_buffer.append([state, action, reward, new_state])
```

Then, if the buffer is full, a random experience is sampled from the buffer. The sample size is a parameter to the q_learn function (“batch_size”). From the experience, the individual items (lists) are extracted and saved to variables:

```
sampled_transitions = np.array(random.sample(replay_buffer, params['BATCH_SIZE']))

state_s = sampled_transitions[:,0]
action_s = sampled_transitions[:,1]
reward_s = sampled_transitions[:,2]
new_state_s = sampled_transitions[:,3]
```

2D-Tensors are created from the variables. In the state-tensors, all the values in the columns of indexes from the state-lists are replaced with 1:

```
state_tensor_s = torch.zeros(params['BATCH_SIZE'], params['STATES'])
state_tensor_s[torch.arange(params['BATCH_SIZE']), state_s] = 1
new_state_tensor_s = torch.zeros(params['BATCH_SIZE'], params['STATES'])
new_state_tensor_s[torch.arange(params['BATCH_SIZE']), new_state_s] = 1
reward_s = torch.tensor(reward_s) # tensor of reward_s
```

Next, q-values are retrieved by solving this formula:

$$Y_k^Q = r + \gamma \max_{a' \in \mathcal{A}} Q(x', a'; \theta_k).$$

in the code:

```
next_q, _ = torch.max(target_model(new_state_tensor_s).detach(),1) # max q given new_state and appro
target_q = reward_s + params['DISCOUNT'] * next_q # r + gamma * max q
target_q = target_q.type(torch.FloatTensor)
model = train(model, state_tensor_s, target_q, action_s, criterion, optimizer, params['BATCH_SIZE'])
```

where ‘next_q’ is the max value from the tensor at a given state. The q-value is saved to a tensor and sent for training.

The model is trained with the help of MSEloss loss function (‘criterion’) for measuring mean-squared-error (MSE), and RMSprop optimizer (‘optimizer’) for normalizing the gradient update-steps, from PyTorch library, along with a given batch_size.

In the train-function, the loss criterion measures MSE from the model-tensor at a given index to Y, where Y is the q-value-tensor. The ‘backward’ function computes the gradients of the weights of the NN and updates the weights, with regards to the loss criterion and batch_size.

```
def train(model, state_vector, Y, actions, criterion, optimizer, BATCH_SIZE):
  """Training is done using target Q value (coming from target model)"""
  out = model(state_vector)
  loss = criterion(out[range(BATCH_SIZE),actions], Y)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
  return model
```

Results

We operate with 5 states, $\in = 1$, and varying **α**-values and decay-ranges. The program is run for 1000 steps in 500 episodes, with batch-size 1000 and replay-memory-limit 50000. The fixed-q-values are:

```
action 1   action 2
[[6.5610, 6.1049]] state 0
[[7.2900, 5.9049]] state 1
[[8.1000, 5.9049]] state 2
[[9.0000, 5.9049]] state 3
[[10.0000,  5.9049]] state 4
```

These are the same as the q-values in part 1 for 5 states.

Regardless of the decay range and **α**-value, all scenarios converge on episode 101:

Epsilon = 1
Learning rate = 0.99
Decay range = [1, 445]
 	
Values already converged at:
`Episode:  101, average reward:  0.1, current epsilon: 0.78`
 	
`At the end of episode # 101:`
`tensor([[6.5610, 6.1049]])`
`tensor([[7.2900, 5.9049]])`
`tensor([[8.1000, 5.9049]])`
`tensor([[9.0000, 5.9049]])`
`tensor([[10.0000,  5.9049]])`
	Epsilon = 1
	Learning rate = 0.99
	Decay range = [1, 1000]
	
	Values already converged at:
	`Episode:  101, average reward:  0.1, current epsilon: 0.90`
	
	`At the end of episode # 101:`
	`tensor([[6.5610, 6.1049]])`
	`tensor([[7.2900, 5.9049]])`
	`tensor([[8.1000, 5.9049]])`
	`tensor([[9.0000, 5.9049]])`
	`tensor([[10.0000,  5.9049]])`

Epsilon 1
Learning rate = 0.05
Decay range = [1,1000]
 	
Values already converged at:
`Episode:  101, average reward:  0.1, current epsilon: 0.90`
 	
`At the end of episode # 101 :`
`tensor([[6.5610, 6.1049]])`
`tensor([[7.2900, 5.9049]])`
`tensor([[8.1000, 5.9049]])`
`tensor([[9.0000, 5.9049]])`
`tensor([[10.0000,  5.9049]])`
	Epsilon = 1
	Learning rate = 0.05
	Decay range = [1,250]
	
	Values already converged at:
	`Episode:  101, average reward:  0.2, current epsilon: 0.60`
	
	`At the end of episode # 101:`
	`tensor([[6.5610, 6.1049]])`
	`tensor([[7.2900, 5.9049]])`
	`tensor([[8.1000, 5.9049]])`
	`tensor([[9.0000, 5.9049]])`
	`tensor([[10.0000,  5.9049]])`

Epsilon = 1
Learning rate = 0.1
Decaying range = [1, 250]
 	
Values already converged at:
`Episode:  101, average reward: 0.2, current epsilon: 0.60`
 	
`At the end of episode # 101:`
`tensor([[6.5610, 6.1049]])`
`tensor([[7.2900, 5.9049]])`
`tensor([[8.1000, 5.9049]])`
`tensor([[9.0000, 5.9049]])`
`tensor([[10.0000,  5.9049]])`
	Epsilon = 1
	Learning rate = 0.1
	Decaying range = [1, 416]
	
	`Episode:  101, average reward:  0.1, current epsilon: 0.80`
	
	`At the end of episode # 101 :`
	`tensor([[6.5610, 6.1049]])`
	`tensor([[7.2900, 5.9049]])`
	`tensor([[8.1000, 5.9049]])`
	`tensor([[9.0000, 5.9049]])`
	`tensor([[10.0000,  5.9049]])`

Observations

For part 2, the following observations about **α** and decay-range were made:

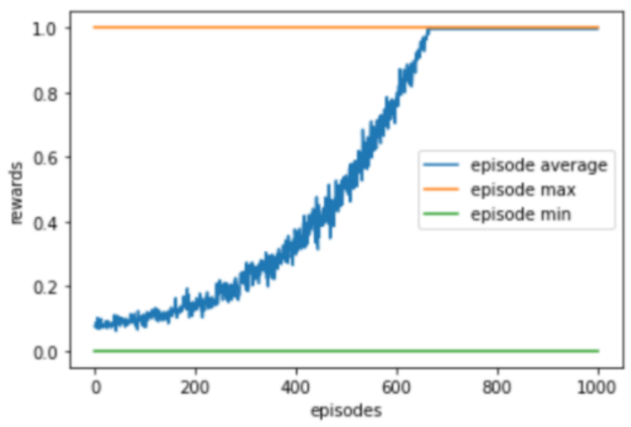
1. We know **α** affects how fast q-values converge. DQN ensures a steady learning curve regardless.
2. The same decay-range decayes ∈ at the same rate, regardless of **α**.

Appendix:

Scenario 1

- **α** = 0.1
- **Initial** ∈ = 1

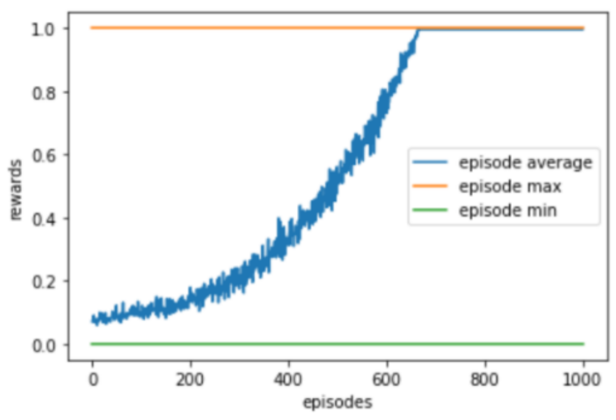
- The q-values converge around episode **55**.
  - At ∈ = 0.92, and **reward** = 0.1
- The average reward converges to 1 at episode **653**.
- ∈ reaches 0 at episode **663**



Scenario 2

- **α** = 0.05
- **Initial** ∈ = 1

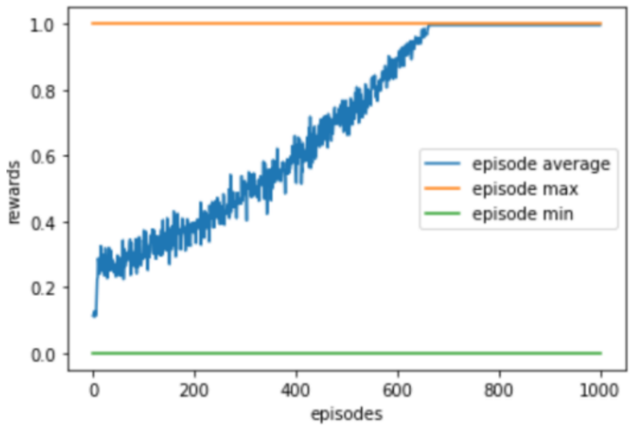
- The q-values converge around episode **96**.
  - At ∈ = 0.86, and **reward** = 0.1
- The average reward converges to 1 at episode **659**.
- ∈ reaches 0 at episode **663**



Scenario 3

- **α** = 0.1
- **Initial** ∈ = 0.5

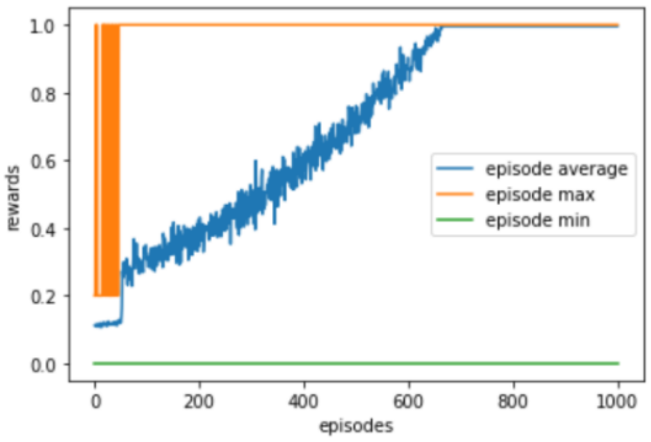
- The q-values converge around episode **13**.
    - At ∈ = 0.46, and **reward** = 0.3
- The average reward converges to 1 at episode **641**.
- ∈ reaches 0 at episode **660**



Scenario 4

- **α** = 0.05
- **Initial** ∈ = 0.5

- The q-values converge around episode **71**.
    - At ∈ = 0.45, and **reward** = 0.3
- The average reward converges to 1 at episode **651**.
- ∈ reaches 0 at episode **660**



Scenario 5

- **α** = 0.5
- **Initial** ∈ = 1

- The q-values converge around episode **13**.
    - At ∈ = 0.98, and **reward** = 0.1
- The average reward converges to 1 at episode **657**.
- ∈ reaches 0 at episode **663**

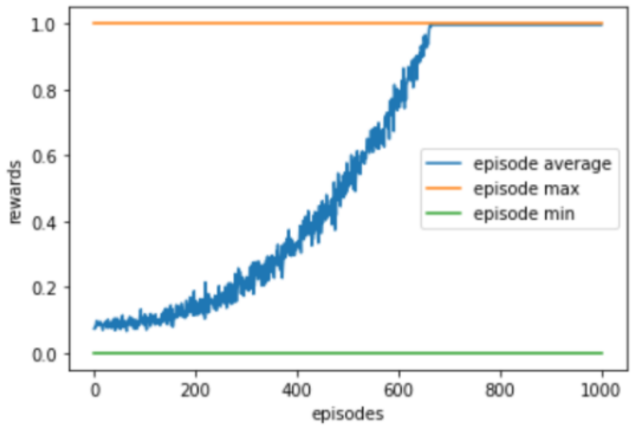


Table 1: Experiment with 5 states results