Robot Learning

Assignment 1

Solutions are due on 19.04.2022 before the lecture.

Task 1.1)

In the game of Tic-Tac-Toe, two players alternate placing crosses and circles on a 3×3 grid, until one player has a row, column, or diagonal of three own pieces, which is a win.

In the following tasks, the game state will be encoded by denoting the symbol X by the digit 2, the symbol O by the digit 0 and the blank field by the digit 1. The board is then viewed as a 3×3 matrix and flattened to a vector with 9 entries. The entries of this vector are regarded as the digits of base 3 number and finally converted to base 10. Using this number, a table of state values V(s) can be maintained. For example:

$$s_{3\times3} = \begin{vmatrix} O & & O \\ O & X & \\ X & X & X \end{vmatrix} \leftrightarrow \begin{bmatrix} 0 & 1 & 0 \\ 0 & 2 & 1 \\ 2 & 2 & 2 \end{bmatrix} \leftrightarrow 010021222_3 \leftrightarrow 2402_{10} =: E(s_{3\times3}) =: s$$

Describe an inverse of this map and illustrate the individual steps in a different example leading to a legal board state!

4 Points

Answer

Lets take a number 3737_{10},

$$3737_{10} \leftrightarrow 020101023_3 \leftrightarrow \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 0 \\ 1 & 0 & 2 \end{bmatrix} \leftrightarrow \begin{vmatrix} O & & X \\ O & & O \\ & O & X \end{vmatrix}$$

Task 1.2)

Closely examine the script below, in which two random agents face off in a game of Tic-Tac-Toe.

Create a suitable structure for state values from the perspective of an agent playing crosses and initialize it for all states with $V(s) = 0.1$. Extend the script below to play $num\_episodes = 10000$ matches and set $V(s) = 1$ when there is a win or $V(s) = 0$ when there is a loss or the game ends in a draw, i.e. no further pieces can be placed.

At the end of each game, go through all states that your agent visited in reverse order and set

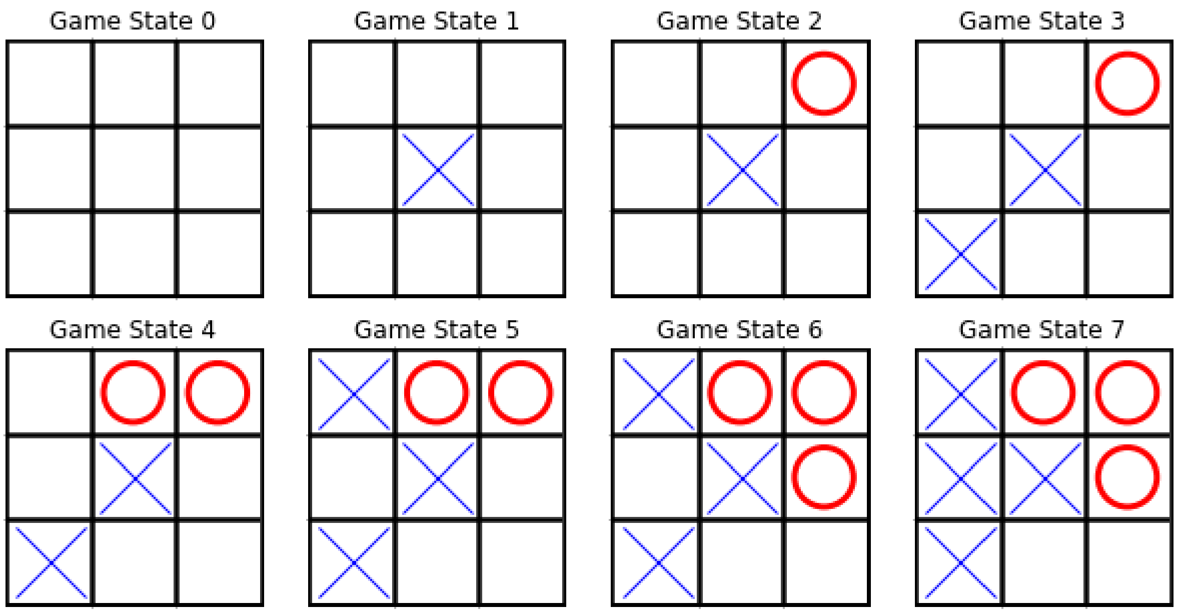
$V(s) \leftarrow V(s) + 0.2[V(s') - V(s)]$, where s' is the successor state.

Document $V(s)$ for all nine states where the agent playing crosses can place their first piece!

6 Points

In [1]:

```
# get the required helper class and visualization function
import numpy as np
from helpers.utils import env, draw_trajectory

# this makes sure that all experiments can be reproduced
np.random.seed(321)

# create a game environment
tictactoe = env()
# reset the game to the empty board - do this before each new episode
tictactoe.reset()

# create a flag which tracks whether the game is over
done = False

# create list to save the trajectory of the agent, start with empty board
states = [np.ones((1,3,3),dtype = int)]

# simulate until the episode is over
while not done:
    # which actions are possible for 'X' at the current state?
    # what are the indices of the states arising from these actions?
    next_state_indices, possible_actions = tictactoe.get_available_actions()
    # select one of the actions randomly
    a_x = tuple(possible_actions[np.random.choice(possible_actions.shape[0])]
    # execute the action using the step function, observe next state and rewa
    # the reward r is 1 when 'X' wins and 0 otherwise
    s_matrix, r, done = tictactoe.step(a=a_x)
    # save the state in 3x3 matrix form
    states.append(s_matrix)
    # HINT: call tictactoe.state_to_ind(s_matrix) to encode the state
    # we don't need to simulate for 'O' if the game is already over
    if done:
        break
    # simulate the random action of the 'O' player in the same manner
    _, possible_actions = tictactoe.get_available_actions(p=0)
    a_o = tuple(possible_actions[np.random.choice(possible_actions.shape[0])]
    s_matrix, r, done = tictactoe.step(a=a_o, player=0)
    states.append(s_matrix)

# use the helper function to display how the episode went
draw_trajectory(np.concatenate(states))
```

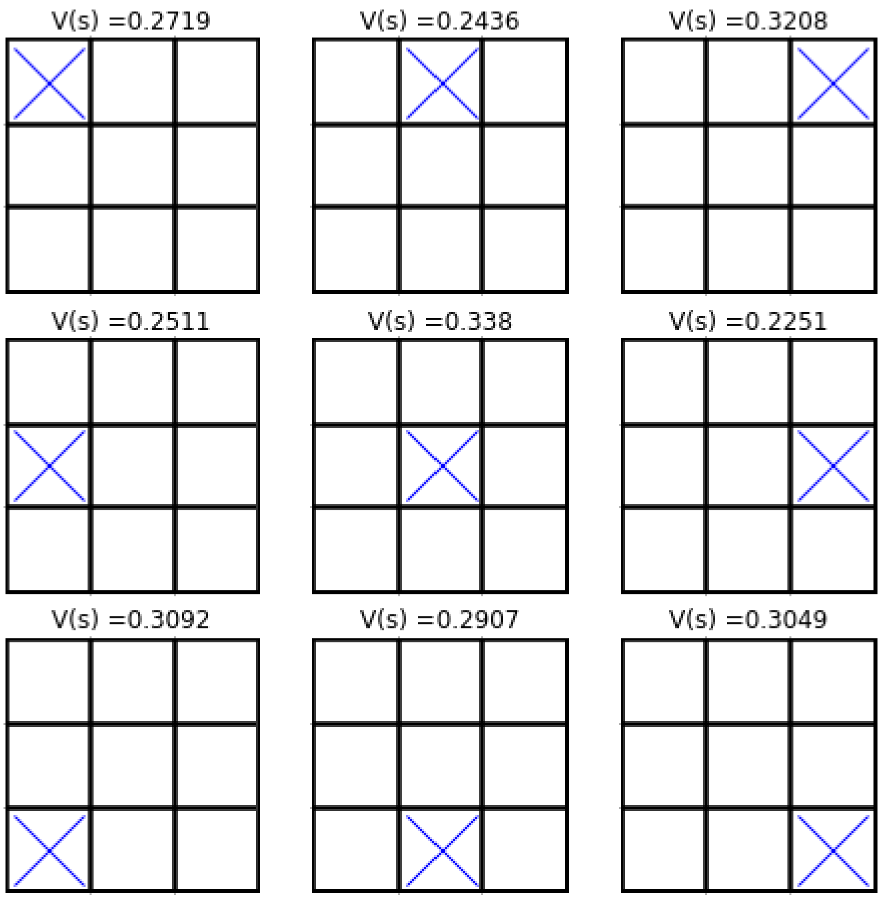


In [2]:

```
# Helper class and visualization function
import numpy as np
from helpers.utils import env, draw_trajectory

# Creating game environment
tictactoe = env()

# State-value function
V = np.empty((19602)) #Equivalent to 222220000-Base-3 (maximum possible)
V.fill(0.1)

num_episodes = 10000 # Matches
np.random.seed(321)

for episode in range(num_episodes):
    # Game reset
    tictactoe.reset()
    # Trajectory of the agent starting from empty board
    states = [np.ones((1,3,3),dtype = int)]
    # Game over flag reset
    done = False
    while not done:
        # simulating the random action of the 'X' player
        next_state_indices, possible_actions = tictactoe.get_available_actior
        a_x = tuple(possible_actions[np.random.choice(possible_actions.shape[
        s_matrix, r, done = tictactoe.step(a=a_x)
        states.append(s_matrix)
        if done:
            break
        # simulating the random action of the 'O' player
        _, possible_actions = tictactoe.get_available_actions(p=0)
        a_o = tuple(possible_actions[np.random.choice(possible_actions.shape[
        s_matrix, r, done = tictactoe.step(a=a_o, player=0)
        states.append(s_matrix)
    # Last state of the game
    s = tictactoe.state_to_ind(s_matrix)
    V[s] = r # Set 1 or 0 based on success or failure
    successor_state = int(s) # set current state as successor
    # Updating the State-value function
    for state in reversed(states[:-1]):
        # Iterating over the states in reverse order
        s = tictactoe.state_to_ind(state)
        # Updating State-value function at 's'
```

```python
        V[s] = V[s] + 0.2 * (V[successor_state] - V[s])
        successor_state = int(s) # set current state as successor

# Nine diffrent starting positions
starting_moves = np.array([[[2 if i*3+j==n else 1 for j in range(3)] for i in
s = tictactoe.state_to_ind(starting_moves)
results = V[s]
# Printing Results
for n,result in enumerate(results):
    print("V(s), If 'x' is starting at position "+str([n//3+1,n%3+1])+" is ",

# Plotting the values
import matplotlib.pyplot as plt
fig, axes = plt.subplots(nrows = 3, ncols = 3, squeeze = False, figsize = (2.
for ind, state in enumerate(starting_moves):
    crosses = np.argwhere(state == 2)
    axes[ind // 3, ind % 3].matshow(np.zeros((3,3)), cmap = 'Greys')
    axes[ind // 3, ind % 3].set_xticks([x-0.5 for x in range(1,3)],minor=True
    axes[ind // 3, ind % 3].set_yticks([y-0.5 for y in range(1,3)],minor=True
    axes[ind // 3, ind % 3].grid(color = 'k',which="minor",ls="-",lw=2.5)
    axes[ind // 3, ind % 3].set_xticks([])
    axes[ind // 3, ind % 3].set_yticks([])
    axes[ind // 3, ind % 3].patch.set_edgecolor('black')
    axes[ind // 3, ind % 3].patch.set_linewidth('3')
    # To plot V(s)
    s = tictactoe.state_to_ind(state)
    axes[ind // 3, ind % 3].set_title('V(s) =' +str(round(V[s][0],4)) )
    axes[ind // 3, ind % 3].plot(crosses[:,1],crosses[:,0],c = 'b', marker='x
plt.show()
```

```
V(s), If 'x' is starting at position [1, 1] is  0.2719474741678269
V(s), If 'x' is starting at position [1, 2] is  0.2435687495419921
V(s), If 'x' is starting at position [1, 3] is  0.3208356967456222
V(s), If 'x' is starting at position [2, 1] is  0.25107114203333003
V(s), If 'x' is starting at position [2, 2] is  0.33800429089316264
V(s), If 'x' is starting at position [2, 3] is  0.22506821026947688
V(s), If 'x' is starting at position [3, 1] is  0.30921198426718166
V(s), If 'x' is starting at position [3, 2] is  0.29068967120274664
V(s), If 'x' is starting at position [3, 3] is  0.3049391885752953
```



Task 1.3)

Replace your agent's arbitrary action selection by an epsilon-greedy strategy that with probability 0.9 places your piece such that $V(s)$ is maximized (break ties randomly) and with probability 0.1 places your piece uniformly among the empty fields.

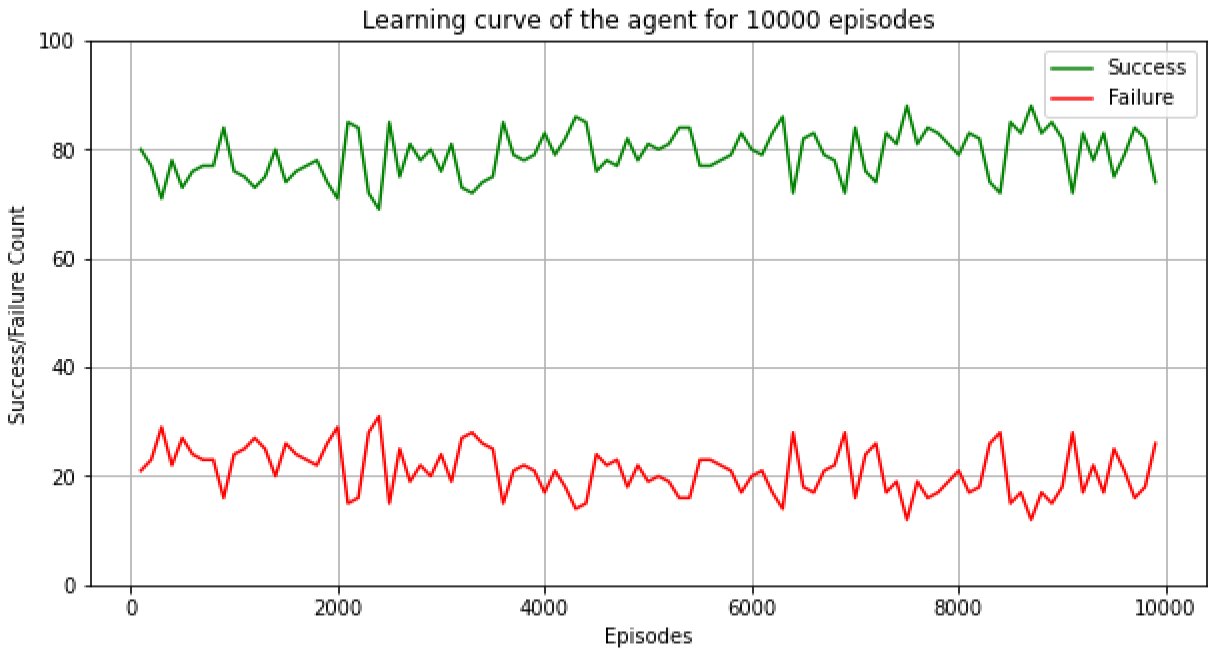
Play 10000 automated games. For every 100 games compute how often your player won and document this learning curve!

Again document $V(s)$ for all nine states where you can place your first piece!

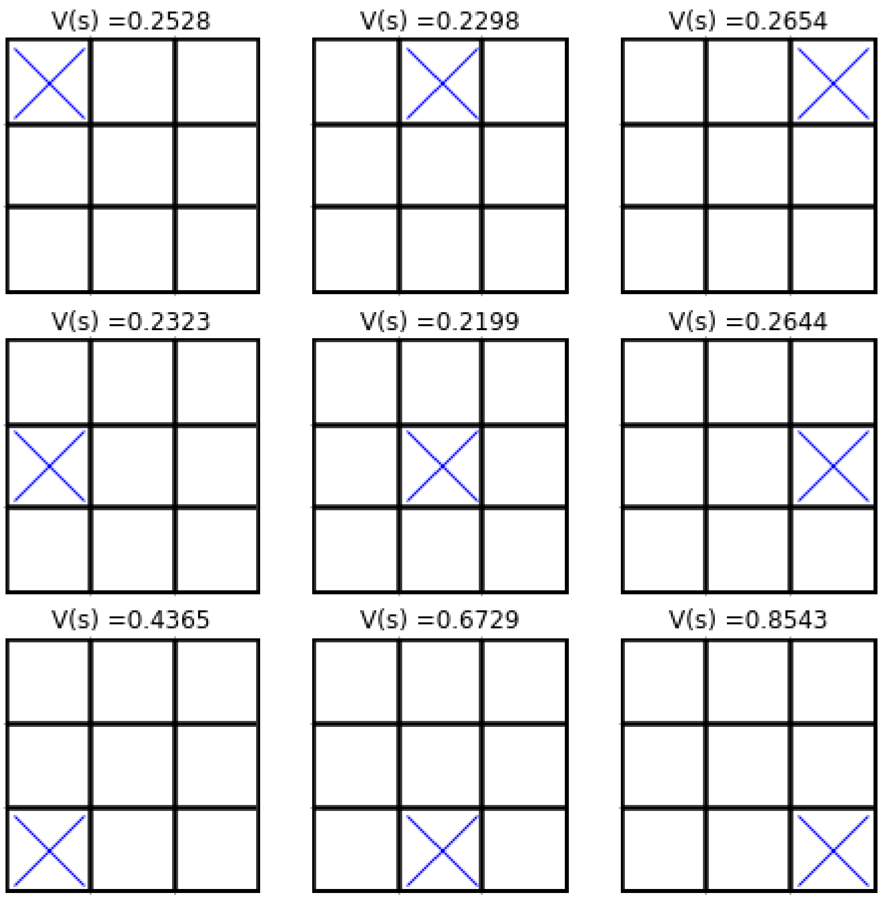
6 Points

In [3]:
```
# Helper class and visualization function
import numpy as np
from helpers.utils import env, draw_trajectory

# Creating game environment
tictactoe = env()

# State-value function
V = np.empty((19602)) #Equivalent to 222220000-Base-3 (maximum possible)
V.fill(0.1)

# Action-value function Q(a)
Q = np.zeros((19602,1,3,3))
epsilon = 0.1

num_episodes = 10000 # Matches
np.random.seed(321)
success,failure = 0,0
```

```python
learning_curve = {'episode':list(),
                  'success':list(),
                  'failure':list(),}

for episode in range(num_episodes):
    # Game reset
    tictactoe.reset()
    # Trajectory of the agent starting from empty board
    states = [np.ones((1,3,3),dtype = int)]
    # Game over flag reset
    done = False
    # Action count reset
    k=0
    while not done:
        [current_state] = tictactoe.state_to_ind(states[-1])
        # simulating the random action of the 'X' player
        next_state_indices, possible_actions = tictactoe.get_available_action
        # randomness - Action selection
        if np.random.random() <= epsilon : # explore
            a_x = tuple(possible_actions[np.random.choice(possible_actions.sh
        else:  # exploit
            max_value = 0
            for possible_action in possible_actions:
                if Q[current_state][tuple(possible_action)] >= max_value:
                    max_value = Q[current_state][tuple(possible_action)]
                    a_x = tuple(possible_action) # epsilon-greedy policy
        s_matrix, r, done = tictactoe.step(a=a_x)
        k+=1 # Increasing Action count
        states.append(s_matrix)
        # Updating Action-value function Q(a)
        Q[current_state][a_x]  =  Q[current_state][a_x] + 1/(k+1) * (r - Q[cu
        if done:
            break
        # simulating the random action of the 'O' player
        _, possible_actions = tictactoe.get_available_actions(p=0)
        a_o = tuple(possible_actions[np.random.choice(possible_actions.shape[
        s_matrix, r, done = tictactoe.step(a=a_o, player=0)
        states.append(s_matrix)
    # Last state of the game
    s = tictactoe.state_to_ind(s_matrix)
    V[s] = r # Set 1 or 0 based on success or failure
    successor_state = int(s) # set current state as successor
    # Updating the State-value function
    for state in reversed(states[:-1]):
        # Iterating over the states in reverse order
        s = tictactoe.state_to_ind(state)
        # Updating State-value function at 's'
        V[s] = V[s] + 0.2 * (V[successor_state] - V[s])
        successor_state = int(s) # set current state as successor
    # Success and failure count
    if r == 1 : success+=1
    else : failure+=1
    if episode%100==0 and episode!=0:
        learning_curve['episode'].append(episode)
        learning_curve['success'].append(success)
        learning_curve['failure'].append(failure)
        success,failure = 0,0

# Plotting Learning curve
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1,1, figsize = (10,5))
ax.set_title("Learning curve of the agent for 10000 episodes")
ax.plot(learning_curve['episode'],learning_curve['success'],label="Success",c
ax.plot(learning_curve['episode'],learning_curve['failure'],label="Failure",c
plt.ylim([0, 100])
plt.xlabel("Episodes")
plt.ylabel("Success/Failure Count")
ax.legend()
ax.grid()
plt.show()

# Nine diffrent starting positions
starting_moves = np.array([[[2 if i*3+j==n else 1 for j in range(3)] for i in
s = tictactoe.state_to_ind(starting_moves)
results = V[s]
# Printing Results
for n,result in enumerate(results):
    print("V(s), If 'x' is starting at position "+str([n//3+1,n%3+1])+" is ",

# Plotting the values
import matplotlib.pyplot as plt
fig, axes = plt.subplots(nrows = 3, ncols = 3, squeeze = False, figsize = (2.
for ind, state in enumerate(starting_moves):
    crosses = np.argwhere(state == 2)
    axes[ind // 3, ind % 3].matshow(np.zeros((3,3)), cmap = 'Greys')
    axes[ind // 3, ind % 3].set_xticks([x-0.5 for x in range(1,3)],minor=True
    axes[ind // 3, ind % 3].set_yticks([y-0.5 for y in range(1,3)],minor=True
    axes[ind // 3, ind % 3].grid(color = 'k',which="minor",ls="-",lw=2.5)
    axes[ind // 3, ind % 3].set_xticks([])
    axes[ind // 3, ind % 3].set_yticks([])
    axes[ind // 3, ind % 3].patch.set_edgecolor('black')
    axes[ind // 3, ind % 3].patch.set_linewidth('3')
    # To plot V(s)
    s = tictactoe.state_to_ind(state)
    axes[ind // 3, ind % 3].set_title('V(s) =' +str(round(V[s][0],4)) )
    axes[ind // 3, ind % 3].plot(crosses[:,1],crosses[:,0],c = 'b', marker='x
plt.show()
```



```
V(s), If 'x' is starting at position [1, 1] is  0.2527565647692102
V(s), If 'x' is starting at position [1, 2] is  0.22975949291669417
V(s), If 'x' is starting at position [1, 3] is  0.26535827742770535
V(s), If 'x' is starting at position [2, 1] is  0.23225190010934116
V(s), If 'x' is starting at position [2, 2] is  0.21985805305427336
V(s), If 'x' is starting at position [2, 3] is  0.26442195941534474
V(s), If 'x' is starting at position [3, 1] is  0.43650681662428503
V(s), If 'x' is starting at position [3, 2] is  0.6729075151064435
V(s), If 'x' is starting at position [3, 3] is  0.8543332353276696
```



Task 1.4)

Your trained agent from the previous task should beat the random opponent in most, but not all games. Give two reasons why this is the case. Explain whether your trained agent could plausibly consistently beat a human expert.

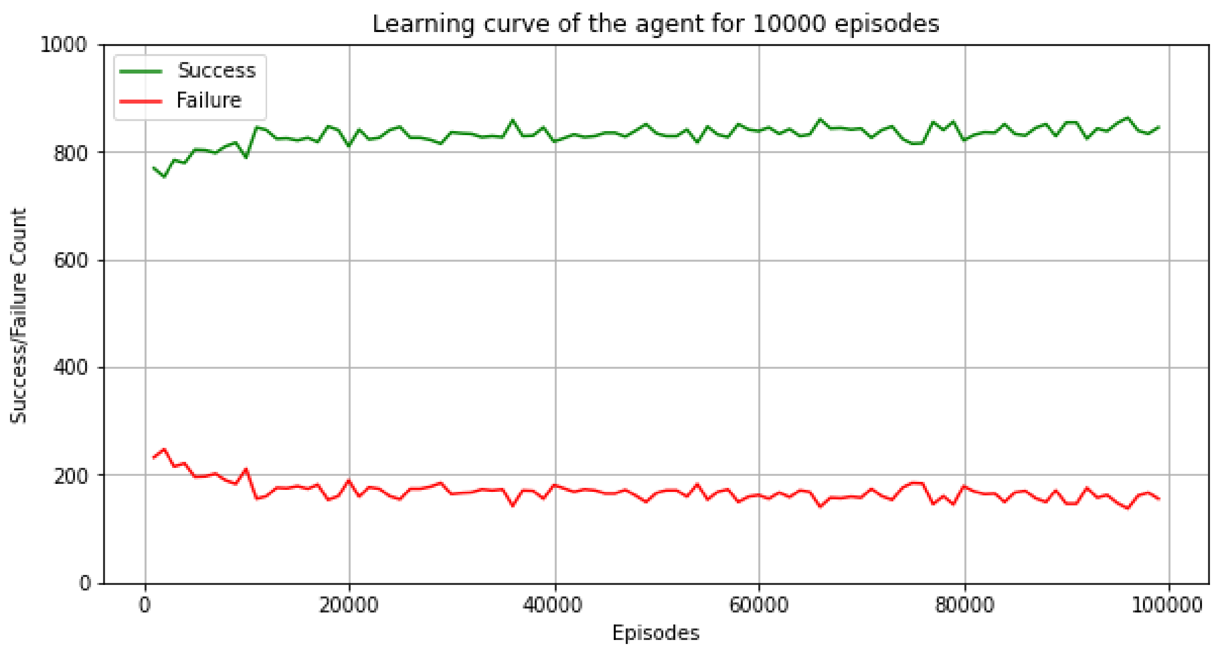
4 Points

Answer

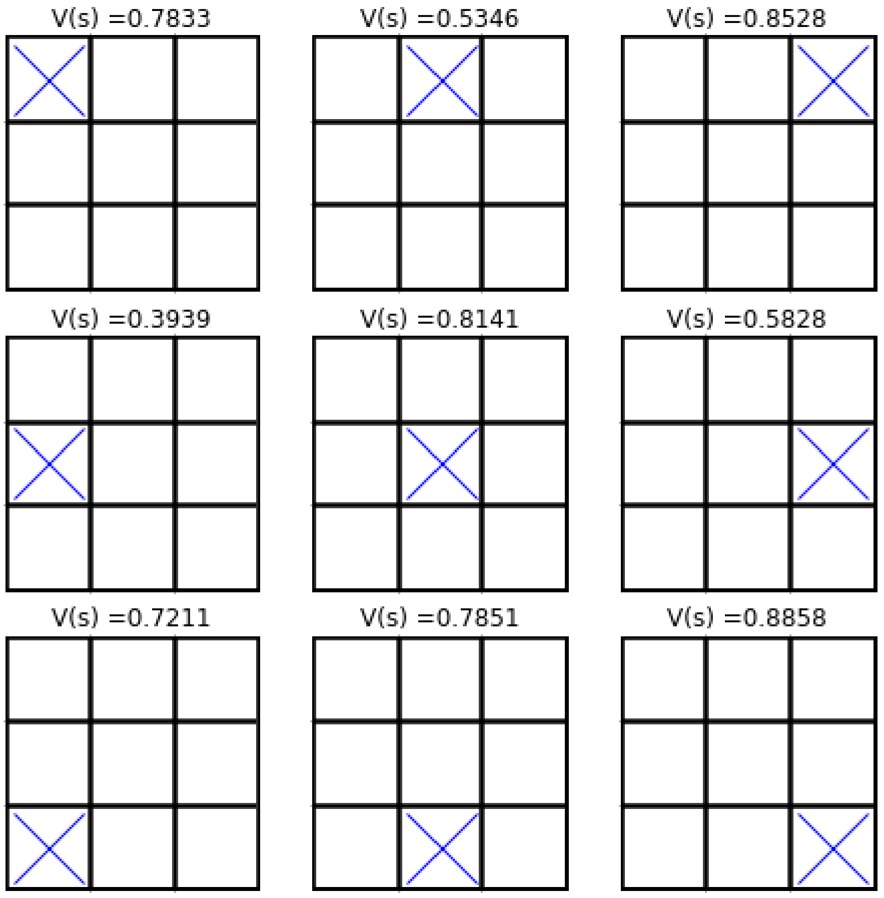
- Epsilon value \ 1) Training with this small ϵ value at the early episodes, leads to over exploitation. Eg. In Task-1.3 The $V(s)$ value for the state where agent strating its first move at $(3x3)$ is significantly very high compared to others. This move might be selected at random during initial exporation steps and the agent chooses it repeatly during exploitation. \ 2) Even after learning all the values for the Action-value function, Our agent will make random moves one in ten times ($\epsilon = 0.1$). This may lead to failures in some cases. The Ideal solution is have a high epsilon value at the initial episodes and diminish it at the later episodes so that our agent will expore more at the beginning and reduce random actions when it is trained \
- Number of Episodes \ 3) Training more episodes will provide more chances for exporation and eventually improves the success rate. In the below cell, the agent is trained for 100000 matches, The success rate was improved after around 20000 training episodes.

The trained agent could not beat a human expert because It was trained only against a random agent. The learned action-values will not be sufficient to make competitive moves. Training again with the human expert can improve the agent action selection but trainig for 10000 matches will be highly time consuming.

In [4]:
```
# Helper class and visualization function
import numpy as np
from helpers.utils import env, draw_trajectory

# Creating game environment
tictactoe = env()

# State-value function
V = np.empty((19602)) #Equivalent to 222220000-Base-3 (maximum possible)
V.fill(0.1)

# Action-value function Q(a)
Q = np.zeros((19602,1,3,3))
epsilon = 0.1

num_episodes = 100000 # Matches
np.random.seed(321)
success,failure = 0,0
learning_curve = {'episode':list(),
                  'success':list(),
                  'failure':list(),}

for episode in range(num_episodes):
    # Game reset
    tictactoe.reset()
    # Trajectory of the agent starting from empty board
    states = [np.ones((1,3,3),dtype = int)]
    # Game over flag reset
    done = False
    # Action count reset
    k=0
    while not done:
        [current_state] = tictactoe.state_to_ind(states[-1])
        # simulating the random action of the 'X' player
        next_state_indices, possible_actions = tictactoe.get_available_action
        # randomness - Action selection
        if np.random.random() <= epsilon : # explore
            a_x = tuple(possible_actions[np.random.choice(possible_actions.sh
        else:  # exploit
            max_value = 0
            for possible_action in possible_actions:
                if Q[current_state][tuple(possible_action)] >= max_value:
                    max_value = Q[current_state][tuple(possible_action)]
                    a_x = tuple(possible_action) # epsilon-greedy policy
        s_matrix, r, done = tictactoe.step(a=a_x)
        k+=1 # Increasing Action count
        states.append(s_matrix)
        # Updating Action-value function Q(a)
        Q[current_state][a_x]  =  Q[current_state][a_x] + 1/(k+1) * (r - Q[cu
        if done:
            break
        # simulating the random action of the 'O' player
        _, possible_actions = tictactoe.get_available_actions(p=0)
        a_o = tuple(possible_actions[np.random.choice(possible_actions.shape[
        s_matrix, r, done = tictactoe.step(a=a_o, player=0)
        states.append(s_matrix)
    # Last state of the game
    s = tictactoe.state_to_ind(s_matrix)
    V[s] = r # Set 1 or 0 based on success or failure
    successor_state = int(s) # set current state as successor
    # Updating the State-value function
    for state in reversed(states[:-1]):
        # Iterating over the states in reverse order
        s = tictactoe.state_to_ind(state)
        # Updating State-value function at 's'
        V[s] = V[s] + 0.2 * (V[successor_state] - V[s])
        successor_state = int(s) # set current state as successor
    # Success and failure count
    if r == 1 : success+=1
    else : failure+=1
    if episode%1000==0 and episode!=0:
        learning_curve['episode'].append(episode)
        learning_curve['success'].append(success)
        learning_curve['failure'].append(failure)
        success,failure = 0,0

# Plotting Learning curve
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1,1, figsize = (10,5))
ax.set_title("Learning curve of the agent for 10000 episodes")
ax.plot(learning_curve['episode'],learning_curve['success'],label="Success",c
ax.plot(learning_curve['episode'],learning_curve['failure'],label="Failure",c
plt.ylim([0, 1000])
plt.xlabel("Episodes")
plt.ylabel("Success/Failure Count")
ax.legend()
ax.grid()
plt.show()

# Nine diffrent starting positions
starting_moves = np.array([[[2 if i*3+j==n else 1 for j in range(3)] for i in
s = tictactoe.state_to_ind(starting_moves)
results = V[s]
# Printing Results
for n,result in enumerate(results):
    print("V(s), If 'x' is starting at position "+str([n//3+1,n%3+1])+" is ",

# Plotting the values
import matplotlib.pyplot as plt
fig, axes = plt.subplots(nrows = 3, ncols = 3, squeeze = False, figsize = (2.
for ind, state in enumerate(starting_moves):
    crosses = np.argwhere(state == 2)
    axes[ind // 3, ind % 3].matshow(np.zeros((3,3)), cmap = 'Greys')
    axes[ind // 3, ind % 3].set_xticks([x-0.5 for x in range(1,3)],minor=True
    axes[ind // 3, ind % 3].set_yticks([y-0.5 for y in range(1,3)],minor=True
    axes[ind // 3, ind % 3].grid(color = 'k',which="minor",ls="-",lw=2.5)
    axes[ind // 3, ind % 3].set_xticks([])
    axes[ind // 3, ind % 3].set_yticks([])
    axes[ind // 3, ind % 3].patch.set_edgecolor('black')
    axes[ind // 3, ind % 3].patch.set_linewidth('3')
    # To plot V(s)
    s = tictactoe.state_to_ind(state)
    axes[ind // 3, ind % 3].set_title('V(s) =' +str(round(V[s][0],4)) )
    axes[ind // 3, ind % 3].plot(crosses[:,1],crosses[:,0],c = 'b', marker='x
plt.show()
```



```
V(s), If 'x' is starting at position [1, 1] is  0.7833160671171214
V(s), If 'x' is starting at position [1, 2] is  0.5346420671343061
V(s), If 'x' is starting at position [1, 3] is  0.8528039403230424
V(s), If 'x' is starting at position [2, 1] is  0.3938590462940303
V(s), If 'x' is starting at position [2, 2] is  0.8140719771141364
V(s), If 'x' is starting at position [2, 3] is  0.5827892004318934
V(s), If 'x' is starting at position [3, 1] is  0.721125036542853
V(s), If 'x' is starting at position [3, 2] is  0.7851385059674804
V(s), If 'x' is starting at position [3, 3] is  0.8857868665741699
```



```
In [ ]:
```