



```
import warnings ; warnings.filterwarnings('ignore')

import itertools
import gym, gym_walk
import numpy as np
from tabulate import tabulate
from pprint import pprint
from tqdm import tqdm_notebook as tqdm

from itertools import cycle, count

import random
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
SEEDS = (12, 34, 56, 78, 90)

%matplotlib inline

pip install git+https://github.com/mimoralea/gym-walk#egg=gym-walk

Collecting gym-walk
  Cloning https://github.com/mimoralea/gym-walk to /tmp/pip-install-bl7ogslx/
  Running command git clone --filter=blob:none --quiet https://github.com/mimoralea/gym-walk to commit b915b94cf2ad16f883
  Resolved https://github.com/mimoralea/gym-walk to commit b915b94cf2ad16f883
  Preparing metadata (setup.py) ... done
Requirement already satisfied: gym in /usr/local/lib/python3.12/dist-packages
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.12/dist-packages
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.12/dist-packages
Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.12/dist-packages

plt.style.use('fivethirtyeight')
params = {
    'figure.figsize': (15, 8),
    'font.size': 24,
    'legend.fontsize': 20,
    'axes.titlesize': 28,
    'axes.labelsize': 24,
    'xtick.labelsize': 20,
    'ytick.labelsize': 20
}
pylab.rcParams.update(params)
np.set_printoptions(suppress=True)

def value_iteration(P, gamma=1.0, theta=1e-10):
    V = np.zeros(len(P), dtype=np.float64)
    while True:
        Q = np.zeros((len(P), len(P[0])), dtype=np.float64)
        for s in range(len(P)):
```

```

        for a in range(len(P[s])):
            for prob, next_state, reward, done in P[s][a]:
                Q[s][a] += prob * (reward + gamma * V[next_state] * (not done))
            if np.max(np.abs(V - np.max(Q, axis=1))) < theta:
                break
        V = np.max(Q, axis=1)
    pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
    return Q, V, pi

def print_policy(pi, P, action_symbols=('<', 'v', '>', '^'), n_cols=4, title='Policy'):
    print(title)
    arrs = {k:v for k,v in enumerate(action_symbols)}
    for s in range(len(P)):
        a = pi(s)
        print("| ", end="")
        if np.all([done for action in P[s].values() for _, _, _, done in action]):
            print("".rjust(9), end=" ")
        else:
            print(str(s).zfill(2), arrs[a].rjust(6), end=" ")
        if (s + 1) % n_cols == 0: print("|")

def print_state_value_function(V, P, n_cols=4, prec=3, title='State-value function'):
    print(title)
    for s in range(len(P)):
        v = V[s]
        print("| ", end="")
        if np.all([done for action in P[s].values() for _, _, _, done in action]):
            print("".rjust(9), end=" ")
        else:
            print(str(s).zfill(2), '{}'.format(np.round(v, prec)).rjust(6), end=" ")
        if (s + 1) % n_cols == 0: print("|")

def print_action_value_function(Q,
                                optimal_Q=None,
                                action_symbols=('<', '>'),
                                prec=3,
                                title='Action-value function:'):
    vf_types=('',) if optimal_Q is None else ('', '*', 'err')
    headers = ['s',] + [' '.join(i) for i in list(itertools.product(vf_types, action_symbols))]
    print(title)
    states = np.arange(len(Q))[..., np.newaxis]
    arr = np.hstack((states, np.round(Q, prec)))
    if not (optimal_Q is None):
        arr = np.hstack((arr, np.round(optimal_Q, prec), np.round(optimal_Q-Q, prec)))
    print(tabulate(arr, headers, tablefmt="fancy_grid"))

def get_policy_metrics(env, gamma, pi, goal_state, optimal_Q,
                      n_episodes=100, max_steps=200):
    random.seed(123); np.random.seed(123); env.seed(123)

```

```

reached_goal, episode_reward, episode_regret = [], [], []
for _ in range(n_episodes):
    state, done, steps = env.reset(), False, 0
    episode_reward.append(0.0)
    episode_regret.append(0.0)
    while not done and steps < max_steps:
        action = pi(state)
        regret = np.max(optimal_Q[state]) - optimal_Q[state][action]
        episode_regret[-1] += regret

        state, reward, done, _ = env.step(action)
        episode_reward[-1] += (gamma**steps * reward)

        steps += 1

    reached_goal.append(state == goal_state)
results = np.array((np.sum(reached_goal)/len(reached_goal)*100,
                    np.mean(episode_reward),
                    np.mean(episode_regret)))
return results

def get_metrics_from_tracks(env, gamma, goal_state, optimal_Q, pi_track, coverage):
    total_samples = len(pi_track)
    n_samples = int(total_samples * coverage)
    samples_e = np.linspace(0, total_samples, n_samples, endpoint=True, dtype=int)
    metrics = []
    for e, pi in enumerate(tqdm(pi_track)):
        if e in samples_e:
            metrics.append(get_policy_metrics(
                env,
                gamma=gamma,
                pi=lambda s: pi[s],
                goal_state=goal_state,
                optimal_Q=optimal_Q))
        else:
            metrics.append(metrics[-1])
    metrics = np.array(metrics)
    success_rate_ma, mean_return_ma, mean_regret_ma = np.apply_along_axis(moving_average, 0, metrics, n=100)
    return success_rate_ma, mean_return_ma, mean_regret_ma

def rmse(x, y, dp=4):
    return np.round(np.sqrt(np.mean((x - y)**2)), dp)

def moving_average(a, n=100) :
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n

def plot_value_function(title, V_track, V_true=None, log=False, limit_value=0

```

```

np.random.seed(123)
per_col = 25
linecycler = cycle(["-", "--", ":", "-."])
legends = []

valid_values = np.argwhere(V_track[-1] > limit_value).squeeze()
items_idx = np.random.choice(valid_values,
                             min(len(valid_values), limit_items),
                             replace=False)

# draw the true values first
if V_true is not None:
    for i, state in enumerate(V_track.T):
        if i not in items_idx:
            continue
        if state[-1] < limit_value:
            continue

        label = 'v*({})'.format(i)
        plt.axhline(y=V_true[i], color='k', linestyle='-', linewidth=1)
        plt.text(int(len(V_track)*1.02), V_true[i]+.01, label)

# then the estimates
for i, state in enumerate(V_track.T):
    if i not in items_idx:
        continue
    if state[-1] < limit_value:
        continue
    line_type = next(linecycler)
    label = 'V({})'.format(i)
    p, = plt.plot(state, line_type, label=label, linewidth=3)
    legends.append(p)

```

```
legends.reverse()
```

```

ls = []
for loc, idx in enumerate(range(0, len(legends), per_col)):
    subset = legends[idx:idx+per_col]
    l = plt.legend(subset, [p.get_label() for p in subset],
                  loc='center right', bbox_to_anchor=(1.25, 0.5))
    ls.append(l)
[plt.gca().add_artist(l) for l in ls[:-1]]
if log: plt.xscale('log')
plt.title(title)
plt.ylabel('State-value function')
plt.xlabel('Episodes (log scale)' if log else 'Episodes')
plt.show()

```

```

def decay_schedule(init_value, min_value, decay_ratio, max_steps, log_start=-1)
    decay_steps = int(max_steps * decay_ratio)

```

```

rem_steps = max_steps - decay_steps
values = np.logspace(log_start, 0, decay_steps, base=log_base, endpoint=True)
values = (values - values.min()) / (values.max() - values.min())
values = (init_value - min_value) * values + min_value
values = np.pad(values, (0, rem_steps), 'edge')
return values

```

```

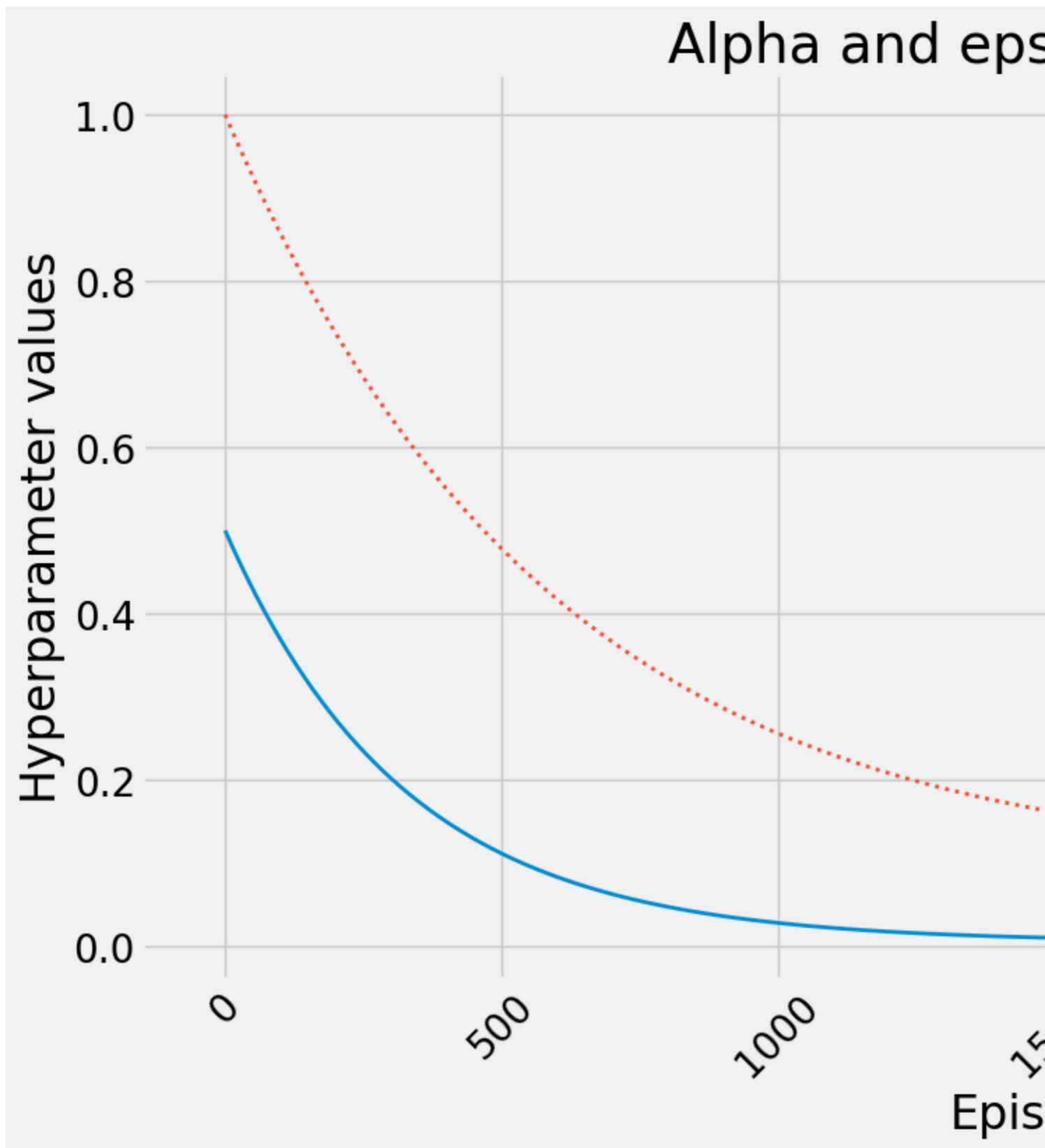
env = gym.make('FrozenLake-v1')
init_state = env.reset()
goal_state = 15
gamma = 0.99
n_episodes = 3000
P = env.env.P
n_cols, svf_prec, err_prec, avf_prec=4, 4, 2, 3
action_symbols=('<', 'v', '>', '^')
limit_items, limit_value = 5, 0.0
cu_limit_items, cu_limit_value, cu_episodes = 10, 0.0, 100

plt.plot(decay_schedule(0.5, 0.01, 0.5, n_episodes),
         '-', linewidth=2,
         label='Alpha schedule')
plt.plot(decay_schedule(1.0, 0.1, 0.9, n_episodes),
         ':', linewidth=2,
         label='Epsilon schedule')
plt.legend(loc=1, ncol=1)

plt.title('Alpha and epsilon schedules')
plt.xlabel('Episodes')
plt.ylabel('Hyperparameter values')
plt.xticks(rotation=45)

plt.show()

```



png

```
optimal_Q, optimal_V, optimal_pi = value_iteration(P, gamma=gamma)
print_state_value_function(optimal_V, P, n_cols=n_cols, prec=svf_prec, title=
```

```

print()

print_action_value_function(optimal_Q,
                            None,
                            action_symbols=action_symbols,
                            prec=avf_prec,
                            title='Optimal action-value function:')

print()
print_policy(optimal_pi, P, action_symbols=action_symbols, n_cols=n_cols)
success_rate_op, mean_return_op, mean_regret_op = get_policy_metrics(
    env, gamma=gamma, pi=optimal_pi, goal_state=goal_state, optimal_Q=optimal_Q)
print('Reaches goal {:.2f}%. Obtains an average return of {:.4f}. Regret of {
    success_rate_op, mean_return_op, mean_regret_op))

```

Optimal state-value function:

```

| 00 0.542 | 01 0.4988 | 02 0.4707 | 03 0.4569 |
| 04 0.5585 |          | 06 0.3583 |          |
| 08 0.5918 | 09 0.6431 | 10 0.6152 |          |
|          | 13 0.7417 | 14 0.8628 |          |

```

Optimal action-value function:

s	<	v	>	^
0	0.542	0.528	0.528	0.522
1	0.343	0.334	0.32	0.499
2	0.438	0.434	0.424	0.471
3	0.306	0.306	0.302	0.457
4	0.558	0.38	0.374	0.363
5	0	0	0	0
6	0.358	0.203	0.358	0.155
7	0	0	0	0
8	0.38	0.408	0.397	0.592
9	0.44	0.643	0.448	0.398
10	0.615	0.497	0.403	0.33
11	0	0	0	0
12	0	0	0	0

13	0.457	0.53	0.742	0.497
14	0.733	0.863	0.821	0.781
15	0	0	0	0

Policy:

00	<	01	^	02	^	03	^
04	<			06	<		
08	^	09	v	10	<		
		13	>	14	v		

Reaches goal 69.00%. Obtains an average return of 0.4808. Regret of 0.0000

```
def generate_trajectory(select_action, Q, epsilon, env, max_steps=200):
    done, trajectory = False, []
    while not done:
        state = env.reset()
        for t in count():
            action = select_action(state, Q, epsilon)
            next_state, reward, done, _ = env.step(action)
            experience = (state, action, reward, next_state, done)
            trajectory.append(experience)
            if done:
                break
            if t >= max_steps - 1:
                trajectory = []
                break
            state = next_state
    return np.array(trajectory, object)

def mc_control(env,
               gamma=1.0,
               init_alpha=0.5,
               min_alpha=0.01,
               alpha_decay_ratio=0.5,
               init_epsilon=1.0,
               min_epsilon=0.1,
               epsilon_decay_ratio=0.9,
               n_episodes=3000,
               max_steps=200,
               first_visit=True):
    nS, nA = env.observation_space.n, env.action_space.n
    discounts = np.logspace(0,
                             max_steps,
                             num=max_steps,
                             base=gamma,
                             endpoint=False)
```



```

alphas = decay_schedule(init_alpha,
                        min_alpha,
                        alpha_decay_ratio,
                        n_episodes)
epsilons = decay_schedule(init_epsilon,
                          min_epsilon,
                          epsilon_decay_ratio,
                          n_episodes)

pi_track = []
Q = np.zeros((nS, nA), dtype=np.float64)
Q_track = np.zeros((n_episodes, nS, nA), dtype=np.float64)
select_action = lambda state, Q, epsilon: np.argmax(Q[state]) \
    if np.random.random() > epsilon \
    else np.random.randint(len(Q[state]))

for e in tqdm(range(n_episodes), leave=False):

    trajectory = generate_trajectory(select_action,
                                    Q,
                                    epsilons[e],
                                    env,
                                    max_steps)

    visited = np.zeros((nS, nA), dtype=bool)
    for t, (state, action, reward, _, _) in enumerate(trajectory):
        if visited[state][action] and first_visit:
            continue
        visited[state][action] = True

        n_steps = len(trajectory[t:])
        G = np.sum(discounts[:n_steps] * trajectory[t:, 2])
        Q[state][action] = Q[state][action] + alphas[e] * (G - Q[state][a])

    Q_track[e] = Q
    pi_track.append(np.argmax(Q, axis=1))

V = np.max(Q, axis=1)
pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
return Q, V, pi, Q_track, pi_track

```

```

Q_mcs, V_mcs, Q_track_mcs = [], [], []
for seed in tqdm(SEEDS, desc='All seeds', leave=True):
    random.seed(seed); np.random.seed(seed); env.seed(seed)
    Q_mc, V_mc, pi_mc, Q_track_mc, pi_track_mc = mc_control(env, gamma=gamma,
    Q_mcs.append(Q_mc) ; V_mcs.append(V_mc) ; Q_track_mcs.append(Q_track_mc)
Q_mc, V_mc, Q_track_mc = np.mean(Q_mcs, axis=0), np.mean(V_mcs, axis=0), np.mean(Q_track_mcs, axis=0)
del Q_mcs ; del V_mcs ; del Q_track_mcs

```

```

All seeds: 0%|          | 0/5 [00:00<?, ?it/s]

```

0%| | 0/3000 [00:00<?, ?it/s]

0%| | 0/3000 [00:00<?, ?it/s]

0%| | 0/3000 [00:00<?, ?it/s]

0%| | 0/3000 [00:00<?, ?it/s]

0%| | 0/3000 [00:00<?, ?it/s]

```
print('Name:JEEVANESH S   Register Number:212222243002')
print_state_value_function(V_mc, P, n_cols=n_cols,
                           prec=svf_prec, title='State-value function found by FVMC')
print_state_value_function(optimal_V, P, n_cols=n_cols,
                           prec=svf_prec, title='Optimal state-value function found by FVMC')
print_state_value_function(V_mc - optimal_V, P, n_cols=n_cols,
                           prec=err_prec, title='State-value function errors: FVMC')
print('State-value function RMSE: {}'.format(rmse(V_mc, optimal_V)))
print()
print_action_value_function(Q_mc,
                            optimal_Q,
                            action_symbols=action_symbols,
                            prec=avf_prec,
                            title='FVMC action-value function:')
print('Action-value function RMSE: {}'.format(rmse(Q_mc, optimal_Q)))
print()
print_policy(pi_mc, P, action_symbols=action_symbols, n_cols=n_cols)
success_rate_mc, mean_return_mc, mean_regret_mc = get_policy_metrics(
    env, gamma=gamma, pi=pi_mc, goal_state=goal_state, optimal_Q=optimal_Q)
print('Reaches goal {:.2f}%. Obtains an average return of {:.4f}. Regret of {:.2f}%'.format(
    success_rate_mc, mean_return_mc, mean_regret_mc))
```

Name:JEEVANESH S Register Number:212222243002

State-value function found by FVMC:

00 0.2212	01 0.1538	02 0.1508	03 0.0766
04 0.2402	05 0.1538	06 0.1753	07 0.1538
08 0.2797	09 0.3622	10 0.3707	11 0.3707
12 0.3707	13 0.5094	14 0.695	15 0.695

Optimal state-value function:

00	0.542	01	0.4988	02	0.4707	03	0.4569
04	0.5585			06	0.3583		
08	0.5918	09	0.6431	10	0.6152		
		13	0.7417	14	0.8628		

State-value function errors:

00	-0.32	01	-0.34	02	-0.32	03	-0.38
04	-0.32			06	-0.18		
08	-0.31	09	-0.28	10	-0.24		
		13	-0.23	14	-0.17		

State-value function RMSE: 0.24

FVMC action-value function:

s	<	v	>	^	* <	* v	* >	* ^	err
0	0.216	0.18	0.17	0.178	0.542	0.528	0.528	0.522	0.3
1	0.082	0.071	0.068	0.152	0.343	0.334	0.32	0.499	0.2
2	0.125	0.086	0.091	0.077	0.438	0.434	0.424	0.471	0.3
3	0.049	0.029	0.009	0.036	0.306	0.306	0.302	0.457	0.2
4	0.24	0.149	0.139	0.143	0.558	0.38	0.374	0.363	0.3
5	0	0	0	0	0	0	0	0	0
6	0.175	0.044	0.065	0.016	0.358	0.203	0.358	0.155	0.1
7	0	0	0	0	0	0	0	0	0
8	0.128	0.161	0.137	0.28	0.38	0.408	0.397	0.592	0.2
9	0.143	0.332	0.238	0.159	0.44	0.643	0.448	0.398	0.2
10	0.318	0.24	0.177	0.093	0.615	0.497	0.403	0.33	0.2
11	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0
13	0.176	0.249	0.509	0.215	0.457	0.53	0.742	0.497	0.2
14	0.339	0.573	0.614	0.438	0.733	0.863	0.821	0.781	0.3
15	0	0	0	0	0	0	0	0	0

Action-value function RMSE: 0.2383

Policy:

00	<	01	v	02	>	03	<
04	<			06	<		
08	^	09	v	10	v		
		13	>	14	>		

Reaches goal 53.00%. Obtains an average return of 0.3942. Regret of 0.2561

```
def sarsa(env,
          gamma=1.0,
          init_alpha=0.5,
          min_alpha=0.01,
          alpha_decay_ratio=0.5,
          init_epsilon=1.0,
          min_epsilon=0.1,
          epsilon_decay_ratio=0.9,
          n_episodes=3000,
          max_steps=200):
    nS, nA = env.observation_space.n, env.action_space.n
    pi_track = []
    Q = np.zeros((nS, nA), dtype=np.float64)
    Q_track = np.zeros((n_episodes, nS, nA), dtype=np.float64)

    # decay schedules
    alphas = decay_schedule(init_alpha, min_alpha, alpha_decay_ratio, n_episodes)
    epsilons = decay_schedule(init_epsilon, min_epsilon, epsilon_decay_ratio, n_episodes)

    # epsilon-greedy action selection
    def select_action(state, epsilon):
        if np.random.rand() < epsilon:
            return np.random.randint(nA)
        else:
            return np.argmax(Q[state])

    for e in tqdm(range(n_episodes), leave=False):
        state = env.reset()
        action = select_action(state, epsilons[e])

        for t in range(max_steps):
            next_state, reward, done, _ = env.step(action)
            next_action = select_action(next_state, epsilons[e])

            # TD target and update
            td_target = reward + gamma * Q[next_state][next_action] * (not done)
            td_error = td_target - Q[state][action]
            Q[state][action] += alphas[e] * td_error

            state, action = next_state, next_action
```

```

        if done:
            break

        Q_track[e] = Q
        pi_track.append(np.argmax(Q, axis=1))

    V = np.max(Q, axis=1)
    pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
    return Q, V, pi, Q_track, pi_track

Q_sarsas, V_sarsas, Q_track_sarsas = [], [], []
for seed in tqdm(SEEDS, desc='All seeds', leave=True):
    random.seed(seed); np.random.seed(seed); env.seed(seed)
    Q_sarsa, V_sarsa, pi_sarsa, Q_track_sarsa, pi_track_sarsa = sarsa(env, gamma)
    Q_sarsas.append(Q_sarsa); V_sarsas.append(V_sarsa); Q_track_sarsas.append(Q_track_sarsa)
Q_sarsa = np.mean(Q_sarsas, axis=0)
V_sarsa = np.mean(V_sarsas, axis=0)
Q_track_sarsa = np.mean(Q_track_sarsas, axis=0)
del Q_sarsas; del V_sarsas; del Q_track_sarsas

All seeds:   0%|          | 0/5 [00:00<?, ?it/s]

0%|          | 0/3000 [00:00<?, ?it/s]

0%|          | 0/3000 [00:00<?, ?it/s]

0%|          | 0/3000 [00:00<?, ?it/s]

0%|          | 0/3000 [00:00<?, ?it/s]

0%|          | 0/3000 [00:00<?, ?it/s]

print('Name:JEEVANESH S                               Register Number:212222243002')
print_state_value_function(V_sarsa, P, n_cols=n_cols,
                           prec=svf_prec, title='State-value function found by')
print_state_value_function(optimal_V, P, n_cols=n_cols,
                           prec=svf_prec, title='Optimal state-value function')
print_state_value_function(V_sarsa - optimal_V, P, n_cols=n_cols,
                           prec=err_prec, title='State-value function errors:

```

```
print('State-value function RMSE: {}'.format(rmse(V_sarsa, optimal_V)))
print()
print_action_value_function(Q_sarsa,
                             optimal_Q,
                             action_symbols=action_symbols,
                             prec=avf_prec,
                             title='Sarsa action-value function:')
print('Action-value function RMSE: {}'.format(rmse(Q_sarsa, optimal_Q)))
print()
print_policy(pi_sarsa, P, action_symbols=action_symbols, n_cols=n_cols)
success_rate_sarsa, mean_return_sarsa, mean_regret_sarsa = get_policy_metrics(
    env, gamma=gamma, pi=pi_sarsa, goal_state=goal_state, optimal_Q=optimal_Q)
print('Reaches goal {:.2f}%. Obtains an average return of {:.4f}. Regret of {
    success_rate_sarsa, mean_return_sarsa, mean_regret_sarsa))
```

Name: JEEVANESH S Register Number: 212222243002

State-value function found by Sarsa:

00 0.1502	01 0.0999	02 0.0874	03 0.0508
04 0.1709		06 0.1215	
08 0.2129	09 0.2922	10 0.3188	
	13 0.4269	14 0.6352	

Optimal state-value function:

00 0.542	01 0.4988	02 0.4707	03 0.4569
04 0.5585		06 0.3583	
08 0.5918	09 0.6431	10 0.6152	
	13 0.7417	14 0.8628	

State-value function errors:

00	-0.39	01	-0.4	02	-0.38	03	-0.41
04	-0.39			06	-0.24		
08	-0.38	09	-0.35	10	-0.3		
		13	-0.31	14	-0.23		

State-value function RMSE: 0.289

Sarsa action-value function:

[illegible]

6	0.082	0.055	0.095	0.01	0.358	0.203	0.358	0.155	0.2
7	0	0	0	0	0	0	0	0	0
8	0.083	0.115	0.107	0.213	0.38	0.408	0.397	0.592	0.2
9	0.127	0.277	0.175	0.113	0.44	0.643	0.448	0.398	0.3
10	0.285	0.237	0.152	0.066	0.615	0.497	0.403	0.33	0.3
11	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0
13	0.103	0.199	0.427	0.191	0.457	0.53	0.742	0.497	0.3
14	0.253	0.466	0.635	0.392	0.733	0.863	0.821	0.781	0.4
15	0	0	0	0	0	0	0	0	0

Action-value function RMSE: 0.2741

Policy:

00	>	01	^	02	>	03	^	
04	<			06	>			
08	^	09	v	10	<			
		13	>	14	>			

Reaches goal 40.00%. Obtains an average return of 0.3180. Regret of 0.2629

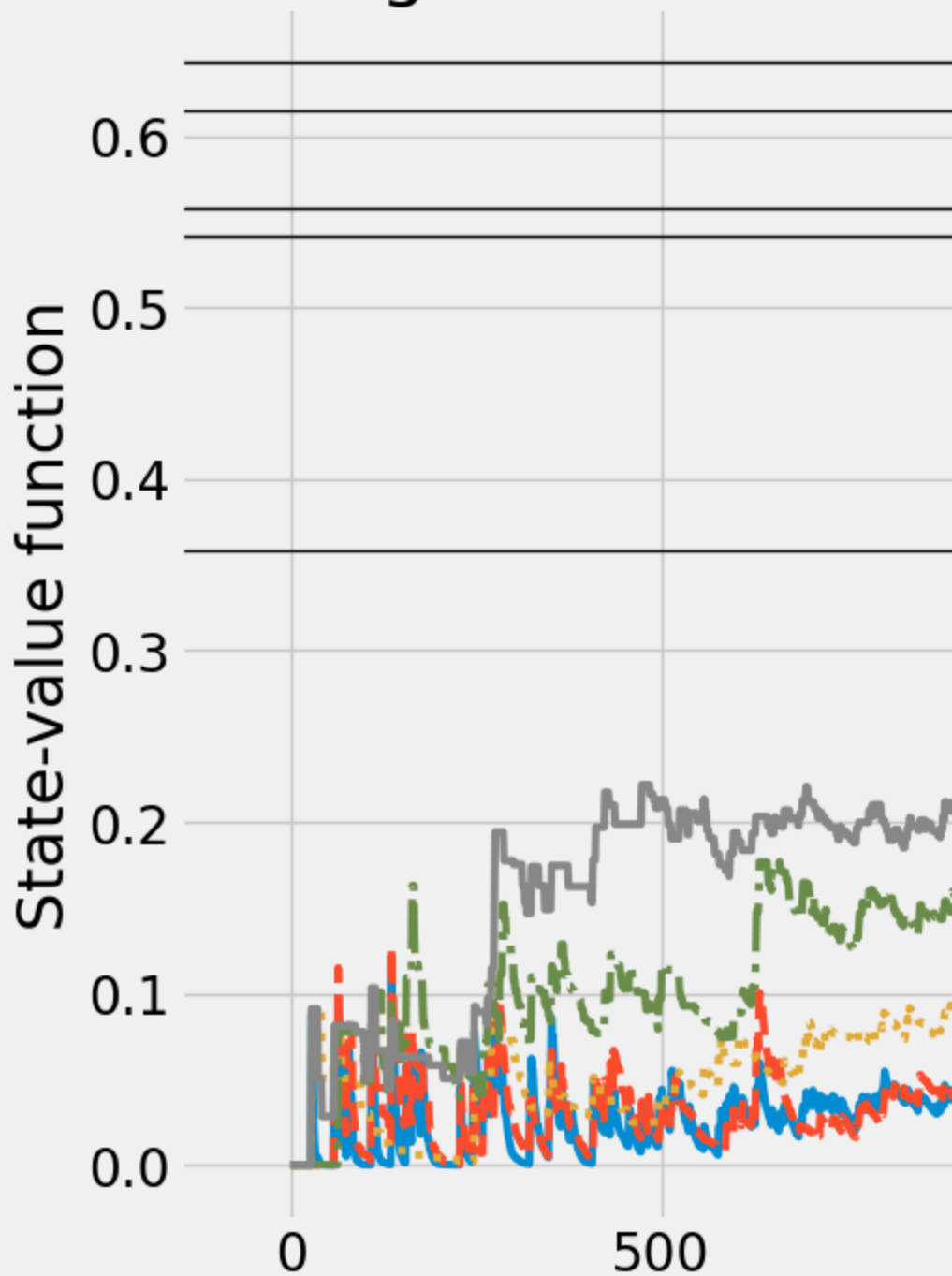
plot_value_function(

```

    'FVMC estimates through time vs. true values Name:JEEVANESH S    Register I
    np.max(Q_track_mc, axis=2),
    optimal_V,
    limit_items=limit_items,
    limit_value=limit_value,
    log=False)

```

FVMC estimates through time vs. true



png

```
plot_value_function(  
    'Sarsa estimates through time vs. true values' Name:JEEVANESH S Register I  
    np.max(Q_track_sarsa, axis=2),  
    optimal_V,  
    limit_items=limit_items,
```



```
limit_value=limit_value,  
log=False)
```

Sarsa estimates through time vs. true

