Sarsa

March 13, 2023

1 Assignment 2

1.1 Importing packages

[1]:
```
import gymnasium as gym
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
import random
import os


os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
rnd = np.random.default_rng(112233)
```

1.2 Building Frozen lake (SARSA)

[2]:
```
env = gym.make('FrozenLake-v1', desc=None, map_name="4x4", is_slippery=True,␣
 ↪render_mode='ansi')

matrix = np.zeros((env.observation_space.n, env.action_space.n))
```

1.3 Building Sarsa class

[3]:
```
class sarsa():
    def __init__(self, decision_matrix, alpha=.85, gamma=.95, temperature=.05,␣
 ↪expected=False):
        self.a = alpha
        self.g = gamma
        self.q = decision_matrix
        self.temp = temperature
        self.expected = expected

        return
```

```
    def update(self, reward, state, action, next_state, next_action=None):  #␣
↪next action can be none in the expected

        if self.expected:
            self.q[state, action] = self.q[state, action] + self.a * (
                    reward + self.g * np.sum(self.q[next_state, :] * self.
↪boltzmann(next_state))
                    - self.q[state, action])
        else:
            self.q[state, action] = self.q[state, action] + self.a * (
                    reward + self.g * self.q[next_state, next_action] - self.
↪q[state, action])

        return None

    def choose(self, env, state, greedy):

        if np.max(self.q[state]) == 0:
            # random sampling
            chosen = rnd.choice(list(range(env.action_space.n)))
        elif greedy or (self.temp <= 0):  # temp 0 means greedy, and cannot go␣
↪to boltzmann to avoid division by 0
            # greedy choice
            chosen = np.argmax(self.q[state])
        else:
            # boltzmann probability
            prob = self.boltzmann(state)
            chosen = rnd.choice(list(range(env.action_space.n)), p=prob)

        return chosen

    def boltzmann(self, state):
        actions = np.divide(self.q[state], self.temp)
        upper = np.exp(actions)
        lower = np.sum(upper)
        return upper / lower
```

1.4 Building the training process

[4]:
```
# defining one episode
def episode(model, env, greedy=0):
    env.reset()
    state = 0  # initializing the state
    ended = False
    reward = 0
```

```
    if not model.expected:
        # Choose A from S
        action = model.choose(env, state, greedy)

    while not ended:

        if model.expected:
            # Choose A from S
            action = model.choose(env, state, greedy)

        # take A from S and get S'
        new_state, reward, ended, time_limit, prob = env.step(action)

        if model.expected:
            model.update(reward, state, action, new_state, None)
        else:
            # choose A' from S'
            new_action = model.choose(env, new_state, greedy)

            model.update(reward, state, action, new_state, new_action)
            # A <- A'
            action = new_action

        # S <- S'
        state = new_state

        if time_limit:
            break

    return reward, greedy
```

[5]:
```
# defining process for each of the segments
def segment(model, env, training, verbose):
    rewards = []
    modes = []

    for i, mode in enumerate(training):
        if verbose:
            print(f"-{i + 1}", end='')
        episode_result, episode_mode = episode(model, env, mode)
        rewards.append(episode_result)
        modes.append(episode_mode)

    return rewards, modes
```

[6]:
```
# defining process for each of the runs
def run(model, env, segments_n=500, training=np.append(np.zeros(10), [1]),␣
 ↪verbose=True):
    rewards = []
    modes = []

    for i, mode in enumerate(range(segments_n)):
        if verbose:
            print(f"\n{i + 1}th Segment:", end='')

        a, b = segment(model, env, training, verbose)
        rewards += a
        modes += b

    return rewards, modes
```

1.5 Running the model

[7]:
```
# configurations

temperatures = [.0, .1, .01]
learning_rates = [.15, .5, .85]
model_name = ['classic', 'expected']
n_runs = 10

training_size = 10
testing_size = 1
```

[ ]:
```
# Running the model
df = None
for type_ in model_name:
    for alpha in learning_rates:
        for temp in temperatures:
            bool_type = True if type_ == 'expected' else False
            for i in range(n_runs):
                print(f"{type_} - {alpha} - {temp} - {i}")
                n_model = sarsa(matrix.copy(), alpha=alpha, temperature=temp,␣
 ↪expected=bool_type)
                rewards, modes = run(n_model, env, training=np.append(np.
 ↪zeros(training_size), [testing_size]),
                                     verbose=False)
                df_ = pd.DataFrame({'reward': rewards, "mode": modes})
                df_['run'] = i
                df_['alpha'] = alpha
                df_['temp'] = temp
                df_['type'] = type_
                if df is None:
```

```
                df = df_
            else:
                df = pd.concat([df, df_])
```

1.6 Tabular RL

Pick 3 settings of the temperature parameter used in the exploration and 3 settings of the learning rate. You need to plot:

1.6.1 First plot

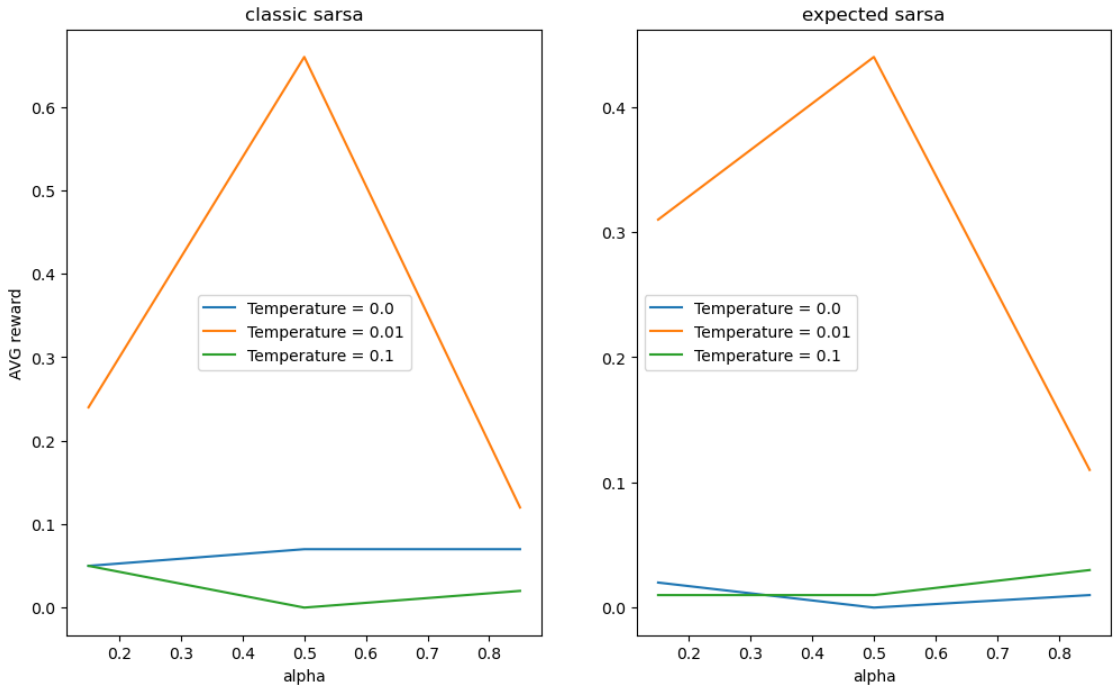
One u-shaped graph that shows the effect of the parameters on the final training performance, expressed as the return of the agent (averaged over the last 10 training episodes and the 10 runs); note that this will typically end up as an upside-down u.

[9]:
```
df_training = df[df['mode'] == 0]  # getting training

reward_runs = {'type': [], 'alpha': [], 'temp': [], 'reward': []}
for model in df_training['type'].unique():
    df_ = df_training[df_training['type'] == model].copy()
    df_ = df_[['alpha', 'temp', 'reward', 'run']]

    for i, group in df_.groupby(['alpha', 'temp', 'run']):  # each config result
        reward = group['reward'].tail(10).mean()
        a = list(group['alpha'])[0]
        t = list(group['temp'])[0]

        reward_runs['type'].append(model)
        reward_runs['alpha'].append(a)
        reward_runs['temp'].append(t)
        reward_runs['reward'].append(reward)

df_ = pd.DataFrame.from_dict(reward_runs)
df_ = df_.groupby(['type', 'alpha', 'temp']).mean().reset_index()
```

[11]:
```
fig, ax = plt.subplots(1, 2, figsize=(12, 7))

for m, model in enumerate(df_['type'].unique()):
    for j, temperature in enumerate(df_['temp'].unique()):
        line_values = []
        for i, alpha in enumerate(df_['alpha'].unique()):
            line_values.append(
                list(df_.loc[df_['alpha'] == alpha][df_['temp'] ==␣
 ↪temperature][df_['type'] == model]['reward'])[0])

        ax[m].plot(df_['alpha'].unique(),line_values, label=f'Temperature =␣
 ↪{str(temperature)}')
        ax[m].legend()
```

```
        ax[m].set_xlabel("alpha")
        ax[m].set_title(f"{model} sarsa")

# plt.subplot(1,1,1)
ax[0].set_ylabel("AVG reward")
plt.show()
```

```
C:\Users\rafael\AppData\Local\Temp\ipykernel_9176\1455600413.py:8: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
  list(df_.loc[df_['alpha'] == alpha][df_['temp'] == temperature][df_['type'] ==
model]['reward'])[0])
```



For the choice of parameters, one of the temperatures was chosen as zero because

1.6.2 Second plot

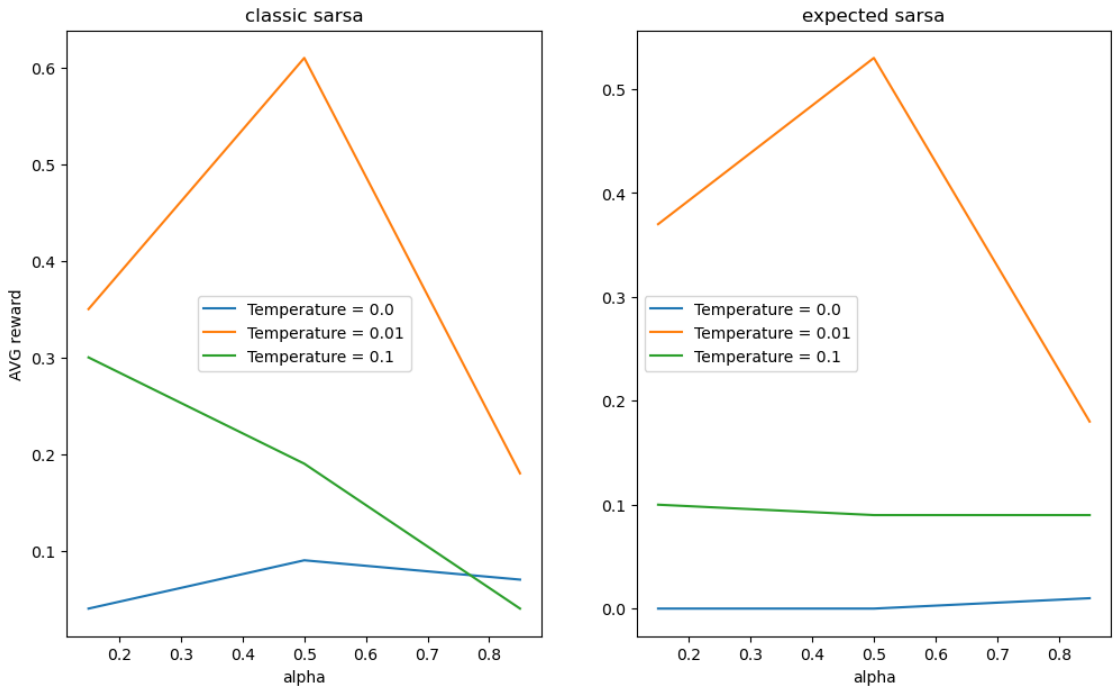
One u-shaped graph that shows the effect of the parameters on the final testing performance, expressed as the return of the agent (during the final testing episode, averaged over the 10 runs)

[12]:
```
df_testing = df[df['mode'] == 1]  # getting training

reward_runs = {'type': [], 'alpha': [], 'temp': [], 'reward': []}
for model in df_testing['type'].unique():
    df_2 = df_testing[df_testing['type'] == model].copy()
    df_2 = df_2[['alpha', 'temp', 'reward', 'run']]
```

```
    for i, group in df_2.groupby(['alpha', 'temp', 'run']):  # each config␣
 ↪result
        reward = group['reward'].tail(10).mean()
        a = list(group['alpha'])[0]
        t = list(group['temp'])[0]

        reward_runs['type'].append(model)
        reward_runs['alpha'].append(a)
        reward_runs['temp'].append(t)
        reward_runs['reward'].append(reward)

df_2 = pd.DataFrame.from_dict(reward_runs)
df_2 = df_2.groupby(['type', 'alpha', 'temp']).mean().reset_index()
```

[14]:
```
fig, ax = plt.subplots(1, 2, figsize=(12, 7))

for m, model in enumerate(df_2['type'].unique()):
    for j, temperature in enumerate(df_2['temp'].unique()):
        line_values = []
        for i, alpha in enumerate(df_2['alpha'].unique()):

            line_values.append(
                list(df_2.loc[df_2['alpha'] == alpha][df_2['temp'] ==␣
 ↪temperature][df_2['type'] == model]['reward'])[0])

        ax[m].plot(df_2['alpha'].unique(),line_values, label=f'Temperature =␣
 ↪{str(temperature)}')
        ax[m].legend()
        ax[m].set_xlabel("alpha")
        ax[m].set_title(f"{model} sarsa")

# plt.subplot(1,1,1)
ax[0].set_ylabel("AVG reward")
plt.show()
```

```
C:\Users\rafael\AppData\Local\Temp\ipykernel_9176\3728207743.py:9: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
  list(df_2.loc[df_2['alpha'] == alpha][df_2['temp'] ==
temperature][df_2['type'] == model]['reward'])[0])
```



1.6.3 Third plot

Learning curves (mean and standard deviation computed based on the 10 runs) for what you pick as the best parameter setting for each algorithm

[15]:
```
def moving_average(a, n=3) :
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n
```

[16]:
```
fig, ax = plt.subplots(len(learning_rates), len(temperatures), sharey='row',␣
↪sharex='col', figsize=(12, 7))

ax[0][1].set_title("Reward over training steps", y=1.2, fontsize=20)

for i, alpha in enumerate(learning_rates):

    for j, temperature in enumerate(temperatures):
        if i == 0:
            text = ax[i][j].text(2150,1.1,f"Temp: {temperature}", size=12)
        if j == 0 and i == 1:
            ax[i][j].set_ylabel(f'Avg Reward over steps and all {n_runs} runs',␣
↪fontsize=12)
        if j == 2:
```

```
            text = ax[i][j].text(6000,.25,f"Alpha: {alpha}", size=12,␣
↪rotation=270)
        if j == 1 and i == 2:
            ax[i][j].set_xlabel(f'Training step', fontsize=12)


        for model in model_name:
            runs_values = []
            mini_df = df[(df['alpha'] == alpha) & (df['temp'] == temperature) &␣
↪(df['type'] == model)][
                ['reward', 'run']]
            for k, group in mini_df.groupby('run'):
                runs_values.append(list(group['reward']))
            runs_values_avg = moving_average(np.mean(np.array(runs_values),␣
↪axis=0),25)
            runs_values_std = moving_average(np.std(np.array(runs_values),␣
↪axis=0),25)
            ax[i][j].plot(list(range(len(runs_values_avg))), runs_values_avg,␣
↪label=model,linewidth=0.5)
            ax[i][j].fill_between(list(range(len(runs_values_avg))),␣
↪runs_values_avg-runs_values_std, runs_values_avg+runs_values_std,alpha=0.2,
   linewidth=4, linestyle='dashdot', antialiased=True)
            ax[i][j].set_ylim([0,1])
            ax[i][j].legend()
            ax[i][j].grid(visible=True)
```

1.7 Q1 discussion:

The ambient used in the problem is a 4x4 scenario with multiple holes between the initial state and the target state. On top of that the "slippery" feature being a .33 chance of moving to a random direction makes it very hard for the agent to perform perfect paths consistently. However, it was possible to achieve an above-average with both the classic and expected SARSA.

The classic and expected SARSA both used softmax for the policy / choosing the next action.

The models with a temperature of 0.01 performed the best, following the trend of the smaller the temperature the better (however a temperature of zero will result in very little exploration and lead to a worse agent) as this environment is very "harsh", and greediness is needed to achieve the final state with some small room for exploration and future improvement.

The second graph showed in more detail that the expected sarsa outperformed the classical on their best configurations in general, therefore, in a real-world deployment where you might want to squeeze every drop of performance, it might be a better option to use the expected SARSA.

From the models with a temperature of 0.01, we can see that an alpha of 0.5 is better than more extreme values like .85 and .15 . That being said, since this experiment stopped after 5000 segments of training, maybe with more training these values could change, and it might be best to lower this learning rate and see if it will converge into a more stable and maybe even higher average then .5.

1.8 Building Cart Pole (Q-learning / ACTOR CRITIC)

[17]:
```
env = gym.make('CartPole-v1')
```

1.9 Building Q-learning class

[18]:
```
class qlearning:
    def __init__(self, env, alpha=.85, gamma=.95, epsilon=.1, bins=10):
        self.a = alpha
        self.g = gamma
        self.q = self.gen_table(env, bins)
        self.e = epsilon
        self.n_bins = bins

        # changing bounds into more compact values to speed up training (fewer␣
 ↪bins needed for this accuracy):
        self.env_space = [[3, -3],
                          [6, -6],
                          [0.300, -0.300],
                          [5, -5]]

        return
```

```
    def gen_table(self, env, bins):
        action_dim = env.action_space.n

        table = np.random.uniform(low=-0.001, high=0.001, size=(bins, bins,␣
↪bins, bins, action_dim))

        self.q = table
        return self.q

    def update(self, reward, state, action, next_state):
        a, b, c, d, e = self.get_s(state, action)
        a_, b_, c_, d_ = self.get_s(next_state)

        self.q[a][b][c][d][e] = self.q[a][b][c][d][e] + self.a * (
                reward + self.g * np.max(self.q[a_][b_][c_][d_]) - self.
↪q[a][b][c][d][e])

        return None

    def choose(self, env, state):

        if rnd.random() < self.e:
            # random sampling
            chosen = rnd.choice(list(range(env.action_space.n)))
        else:
            # greedy choice
            table = self.q
            for miniState in self.get_s(state):
                table = table[miniState]

            chosen = np.argmax(table)
        return chosen

    def get_s(self, state, action=None):
        indexes = []
        for i, feature in enumerate(state):
            max_value = self.env_space[i][0]
            min_value = self.env_space[i][1]

            if (feature > max_value) or (feature < min_value):
                raise ValueError(
                    f"Feature out of bounds for feature{str(i)} on bins :␣
↪{str(feature)}  |min : {str(min_value)} - "
                    f"max :{str(max_value)}|")
            window_size = (max_value - min_value) / self.n_bins
            bin_loc = (feature - min_value) // window_size
            indexes.append(int(bin_loc))
```

```
        if action is None:
            return indexes
        else:
            return indexes + [action]
```

1.10 Building the training process

[19]:
```
# defining one episode
def episode(model, env, render=False, penalty=250):
    state = env.reset()[0]
    if render:
        env.render()
    ended = False
    ep_reward = 0

    while not ended:

        action = model.choose(env, state)

        # take A from S and get S'
        new_state, reward, ended, time_limit, prob = env.step(action)

        if ended:
            reward -= penalty

        model.update(reward, state, action, new_state)

        # S <- S'
        state = new_state
        ep_reward += reward
        if time_limit:
            break

    if render:
        env.close()
    return ep_reward
```

[20]:
```
# defining process for each of the runs
def run(model, env, episode_n=1000, verbose=True, penalty=250):
    run_results = []
    for i, mode in enumerate(range(episode_n)):
        if verbose and (len(run_results) > 1):
            print(f"\n{i + 1}th Segment: {np.mean(run_results)} avg reward",␣
 ↪end='')
        reward = episode(model, env, penalty=penalty)
        run_results.append(reward)
```

```
    return run_results
```

1.11 Running the models

[36]:
```
# configurations
n_bins = 10

epsilons = [.075, .15, 0.2]
learning_rates = [1 / 4, 1 / 8, 1 / 16]

n_runs = 10

#setting one default rng for numpy and 10 seeds for the 10 runs
rnd = np.random.default_rng(112233)

gym_seeds= [11,22,33,44,55,66,77,88,99,1010]

training_size = 10
testing_size = 1
df_qlearn = None
```

[37]:
```
# Runing the training

# generating one initial table state for every run
common_table = np.random.uniform(low=-0.001, high=0.001, size=(n_bins, n_bins,␣
 ↪n_bins, n_bins, 2))

for alpha in learning_rates:
    for epsilon in epsilons:
        print(f'Training on |Epsilon: {str(epsilon)}\t| Alpha: {str(alpha)}')

        episode_results = []

        for i in range(n_runs):
            env = gym.make('CartPole-v1')
            env.action_space.seed(gym_seeds[i])
            result_df = pd.DataFrame()
            # creating model copies for each run

            n_model = qlearning(env, alpha=alpha, epsilon=epsilon, bins=n_bins)
            n_model.q = common_table.copy()
            result_df['ep_reward'] = run(n_model, env, verbose=False)
            result_df['alpha'] = alpha
            result_df['epsilon'] = epsilon
            result_df['run'] = i
            if df_qlearn is None:
```

```
                df_qlearn = result_df.copy()
            else:
                df_qlearn = pd.concat([df_qlearn, result_df])
```

```
Training on |Epsilon: 0.075     | Alpha: 0.25
Training on |Epsilon: 0.15      | Alpha: 0.25
Training on |Epsilon: 0.2       | Alpha: 0.25
Training on |Epsilon: 0.075     | Alpha: 0.125
Training on |Epsilon: 0.15      | Alpha: 0.125
Training on |Epsilon: 0.2       | Alpha: 0.125
Training on |Epsilon: 0.075     | Alpha: 0.0625
Training on |Epsilon: 0.15      | Alpha: 0.0625
Training on |Epsilon: 0.2       | Alpha: 0.0625
```

[38]:
```
df_qlearn
```

[38]:

	ep_reward	alpha	epsilon	run
0	-228.0	0.2500	0.075	0
1	-182.0	0.2500	0.075	0
2	-210.0	0.2500	0.075	0
3	-150.0	0.2500	0.075	0
4	-225.0	0.2500	0.075	0
..	…	…	…	…
995	40.0	0.0625	0.200	9
996	-25.0	0.0625	0.200	9
997	-124.0	0.0625	0.200	9
998	-30.0	0.0625	0.200	9
999	-148.0	0.0625	0.200	9

[90000 rows x 4 columns]

[39]:
```
# saving dataset if desired
df_qlearn.to_csv('Qlearning.csv', index=False, sep=';', encoding='utf-8')

# loading dataset if already ran
# df_qlearn = pd.read_csv('Qlearning.csv', sep=';', encoding='utf-8')
```

1.12 Actor critic

[40]:
```
env = gym.make('CartPole-v1')
```

[41]:
```
num_states = 10 ** len(env.reset()[0])
num_actions = env.action_space.n

df_AClearn = None
```

[42]:
```
# tabular state value function
def get_s(s, n_bins=10):
```

```
    env_space = [[3, -3],
                 [6, -6],
                 [0.300, -0.300],
                 [5, -5]]
    indexes = 0
    for i, feature in enumerate(s):
        max_value = env_space[i][0]
        min_value = env_space[i][1]

        if (feature > max_value) or (feature < min_value):
            raise ValueError(
                f"Feature out of bounds for feature{str(i)} on bins :␣
 ↪{str(feature)}  |min : {str(min_value)} - "
                f"max :{str(max_value)}|")
        window_size = (max_value - min_value) / n_bins
        bin_loc = (feature - min_value) // window_size
        indexes += int(bin_loc) * (10 ** i)

    return indexes
```

[43]:
```
# Run the main loop

def run_ac(env,ep_number,e,alpha,num_states, num_actions,g=.99):
    policy = np.ones((num_states, num_actions)) / num_actions # since it has a␣
 ↪softmax everything as one will avoid division by zero
    V = np.zeros(num_states) # value function
    # Set alpha and gamma
    alpha_actor = alpha
    alpha_critic = alpha_actor/10


    rewards_history = [0, 0, 0, 0, 0, 0, 0, 0, 0]
    for episode in range(ep_number):
        state = get_s(env.reset()[0])
        done = False

        # eligibility traces
        z_critic = np.zeros(num_states)
        z_actor = np.zeros((num_states, num_actions))

        history = []

        while not done:

            action_probs = np.exp(policy[state]) / np.sum(np.exp(policy[state]))

            # e-greedy
```

```
            if rnd.random() < e:
                action = rnd.choice(list(range(num_actions)))  # random action␣
↪to increase exploration
            else:
                action = np.random.choice(num_actions, p=action_probs)  #␣
↪softmax action

            # Take action A observe S', R
            next_state, reward, done, _, _ = env.step(action)
            next_state = get_s(next_state)

            # if S' is terminal then v S' = 0
            target = reward + g * V[next_state] if not done else reward
            TD_error = target - V[state]
            V[state] += alpha_critic * TD_error

            # Update the eligibility traces for the critic and actor
            z_critic[state] += 1
            z_actor[state, action] += 1

            # Update
            grad_log_policy = z_actor / (np.sum(z_actor, axis=1, keepdims=True)␣
↪+ 1e-8)
            policy[state] += alpha_actor * TD_error * grad_log_policy[state]

            # over-time decay of the traces
            z_critic *= g
            z_actor *= g

            # S <- S'
            state = next_state
            history.append(reward)

        episode_reward = sum(history)
        rewards_history.append(episode_reward)
        if episode % 100 == 0:
            print(f"Episode {episode}: reward={episode_reward} | rolling avg of␣
↪last 10 epsodes: {np.mean(rewards_history[-10:])}")
    return rewards_history
```

[ ]:
```
for alpha in learning_rates:
    for epsilon in epsilons:
        print(f'Training on |Epsilon: {str(epsilon)}\t| Alpha: {str(alpha)}')

        episode_results = []
        for i in range(n_runs):
            env = gym.make('CartPole-v1')
```

```
            env.action_space.seed(gym_seeds[i])

            result_df = pd.DataFrame()
            # creating model copies for each run

            result_df['ep_reward'] =␣
 ↪run_ac(env,1000,epsilon,alpha,num_states,num_actions)
            result_df['alpha'] = alpha
            result_df['epsilon'] = epsilon
            result_df['run'] = i
            if df_AClearn is None:
                df_AClearn = result_df.copy()
            else:
                df_AClearn = pd.concat([df_AClearn, result_df])
```

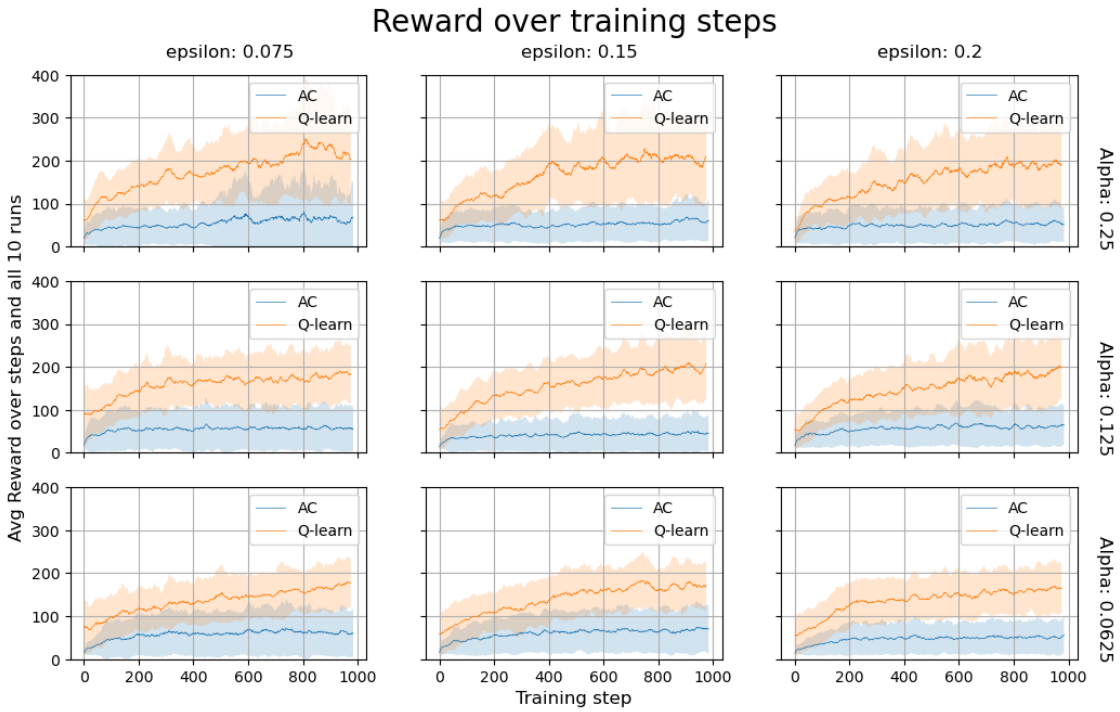
[45]:
```
# saving dataset if desired
df_AClearn.to_csv('AClearning.csv', index=False, sep=';', encoding='utf-8')

# loading dataset if already ran
# df_AClearn = pd.read_csv('AClearning.csv', sep=';', encoding='utf-8')
```

[48]:
```
df_AClearn['model'] = 'AC'
df_qlearn['model'] = 'Q-learn'
df_qlearn['ep_reward'] = df_qlearn['ep_reward'] + 250
```

[49]:
```
df = pd.concat([df_AClearn,df_qlearn])
```

[50]:
```
fig, ax = plt.subplots(len(learning_rates), len(epsilons), sharey='row',␣
 ↪sharex='col', figsize=(12, 7))

ax[0][1].set_title("Reward over training steps", y=1.2, fontsize=20)

for i, alpha in enumerate(learning_rates):

    for j, epsilon in enumerate(epsilons):
        if i == 0:
            text = ax[i][j].text(300,440,f"epsilon: {epsilon}", size=12)
        if j == 0 and i == 1:
            ax[i][j].set_ylabel(f'Avg Reward over steps and all {n_runs} runs',␣
 ↪fontsize=12)
        if j == 2:
            text = ax[i][j].text(1100,.3,f"Alpha: {alpha}", size=12,␣
 ↪rotation=270)
        if j == 1 and i == 2:
            ax[i][j].set_xlabel(f'Training step', fontsize=12)
```

```
        for model in df['model'].unique():
            runs_values = []
            mini_df = df[(df['alpha'] == alpha) & (df['epsilon'] == epsilon) &␣
↪(df['model'] == model)][
                ['ep_reward', 'run']]
            for k, group in mini_df.groupby('run'):
                runs_values.append(list(group['ep_reward']))
            runs_values_avg = moving_average(np.mean(np.array(runs_values),␣
↪axis=0),25)
            runs_values_std = moving_average(np.std(np.array(runs_values),␣
↪axis=0),25)
            ax[i][j].plot(list(range(len(runs_values_avg))), runs_values_avg,␣
↪label=model,linewidth=0.5)
            ax[i][j].fill_between(list(range(len(runs_values_avg))),␣
↪runs_values_avg-runs_values_std, runs_values_avg+runs_values_std,alpha=0.2,
   linewidth=4, linestyle='dashdot', antialiased=True)
            ax[i][j].set_ylim([0,400])
            ax[i][j].legend()
            ax[i][j].grid(visible=True)
```



1.13 Q2 Discussion

This ambient was also an episodic one, but on this, the parameters for the states were continuous, creating new challenges in the implementation. This environment was a cart with a pole trying

to stay with the pole straight up for as long as possible(the longer the more reward). It had 4 features that represented the state instead of just the location of the actor like the Q1 exercise.

Two algorithms were compared, q-learning and the Actor-Critic algorithm with eligibility traces incorporated.

Both models used the table method for encoding the possible states, solving the problem of the continuous values and encoding the state for the value function, and finally, both used an e-greedy method to encourage more exploration (depending on the epsilon value).

For the comparison of the models, it's possible to see in the graphs that the Actor-Critic model perform significantly worse than the Q-learning, however, it did perform better than a random model. This is likely due to problems in the implementation, as the literature showed that the AC model usually performs better or is equal to Q-learning models.

Altough the differences are small this graph showed that the higher epsilon hindered the algorithms to better learn. This could be because this environment is more random and more states will be explored without the need for a very high incentive to the exploration.

On that note, the best Q learning was the one with the lower epsilon and higher alpha. Given the fact that the line kept going up until the very end, there is probably still something left to be learned and the higher alpha managed to help "absorb" it faster. In a scenario where we could train for longer, the model would probably perform even better.

The Epsilon, on the other hand, showed that the lower the value the better, that could be because the volatility of the system will already take care of exploration as explained before, reducing the need for an artificial exploration through the epsilon.

[ ]: