4.1 Expected SARSA

A twist on SARSA that enhances agent decision-making

It is a TD method, model-free, but updates Q-table differently

Expected SARSA rule:
$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma E\{Q(s',A)\}$

This includes the expected Q value for next state based on all actions, making this method more robust to changes and uncertainties.

$E\{Q(s',A)\} = \Sigma(Prob(a) * Q(s',a) \text{ for a in A})$

Here, each Q-value is weighted by the probability of its corresponding action being selected under the current policy

If actions are randomly selected, they have equal probabilities and simplifies:
$E\{Q(s',A)\} = Mean(Q(s',a) \text{ for a in A})$

```
# set up environment
# initialize array of zeros for states and actions
# define parameters
def update_q_table(state, action, next_state, reward):
    expected_q = np.mean(Q[next_state])
    Q[state,action] = (1-alpha)*Q[state,action] + alpha*
(reward+gamma*expected_q)

# standard training loop, but with new update_q_table
```

4.2 Double Q-learning

Q-learning has a tendency of overestimating Q values, which could lead to suboptimal policy learning

Double Q-learning maintains two Q-tables, and each table is updated based on the other to reduce risk of overestimation

Ex: if Q_0 is picked, it gets best next action, but updates its value based on the reward observed from Q_1

This process alternates between both tables

```
# initialize environments, and two Q-tables with same dimensions and zeros
# initialize parameters

# for each action taken in the environment, decide randomly to update one of the tables
def update_q_tables(state, action, reward, next_state):
    i = np.random.randint(2)
    best_next_action = np.argmax(Q[i][next_state])
    Q[i][state, action] = (1-alpha) * Q[i][state,action] + alpha*(reward+gamma*Q[1-i][next_state, best_next_action])

# standard training loop

final_Q = (Q[0] + Q[1])/2
# or
final_Q = Q[0] + Q[1]
```

4.3 Balancing exploration and exploitation

We can balance exploring new actions to gain new info and exploiting current knowledge to maximize rewards

We can use epsilon-greedy strategy, which involves exploring a random action with probability epsilon, and exploiting a best known action with probability 1 - epsilon

The value of epsilon decreases over time, to start with majority exploring, and to end with majority exploitation

```
# initialize env, params (epsilon, epsilon_decay, and min_epsilon), and arrays
def epislon_greedy(state):
    if np.random.rand() < epsilon:
        action = env.action_space.sample() # explore
    else:
        action = np.argmax(Q[state, :]) # exploit
    return action

epsilon = 0.9 # exploration rate
# standard training loop, but use epsilon_greedy to select action
```

4.4 Multi-armed bandits

```
n_bandits=4
turn_bandit_probs = np.random.rand(n_bandits)
# initialize params (decayed epsilon-greedy params)
# initialize zero arrays for counts, values, rewards, and selected_arms

for i in range(n_iterations):
    arm = epsilon_greedy()
```

```
    reward = np.random.rand()
    rewards[0] = reward
    selected_arms[i] = arm
    counts[arm] += 1
    values[arm] += (reward - values[arm]) / counts[arm]
    epsilon = max(min_epsilon, epsilon*epsilon_decay)
```

Analyzing selections

```
selections_percentage = np.zeros((n_iterations, n_bandits))
for i in range(n_iterations):
    selections_percentage[i, selected_arms[i]] = 1
selections_percentage = np.cumsum(selections_percentage, axis=0) /
                        np.arange(1, n_iterations + 1).reshape(-1, 1)
```