3.1 Monte Carlo methods

Monte Carlo methods is a model-free technique to estimate Q-values based on episodes.

1. collect episodes with random actions
2. q-values are estimated for each state and action
3. optimal policy is derived by choosing actions with the higher Q value

In RL, we distinguish between first-visit and every-visit MC methods

In two episodes:

- **unique** tasks' returns can be filled into the Q-value table
- a task **appearing in both episodes** will have their return averaged
- a task appearing twice in one (1,1) and (1,2), and once in the other episode (2,1) will have two options (episode, task):
    - in **first-visit**, (1,1) and (2,1) returns are averaged
    - in **every-visit**, (1,1), (1,2), and (2,1) returns are averaged

```
def generate_episode():
    episode = []
    state, info = env.reset() # reset the environment
    terminated = False
    while not terminated:
        action = env.action_space.sample() # select a random action
        next_state, reward, terminated, truncated, info = env.step(action)
        episode.append((state,action,reward)) # update episode data
        state = next_state
    return episode

def first_visit_mc(num_episodes):
    Q = np.zeros((num_states, num_actions))
    returns_sum = np.zeros((num_states, num_actions))
    returns_count =np.zeros((num_states, num_actions))

    for i in range(num_episodes):
        episode = generate_episode()
        visited_states_actions = set() # remove for every_visit

        for j, (state, action, reward) in enumerate(episode):
            if (state, action) not in visited_states:  # remove for
every_visit, this defines first visit condition
                returns_sum[state,action] += sum([x[2] for x in episode[j:]])
                # Update the returns, their counts and the visited states
                returns_count[state,action] += 1
                visited_states.add((state,action))  # remove for every_visit

    nonzero_counts = returns_count != 0
    Q[nonzero_counts] = returns_sum[nonzero_counts] /
returns_count[nonzero_counts]
    return Q

def get_policy():
    policy = {state: np.argmax(Q[state]) for state in range(num_states)}
    return policy
```

3.2 Temporal difference learning

or TD learning, is model-free.
The critical difference between MC is in the update of estimates

TD learning can update value estimates at each step within an episode, based on the most recent experience

SARSA is a specific TD algorithm (state, action, reward, next_state, next_action)
On-policy method: adjusts strategy based on taken actions

Rule: $Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma Q(s_1, a_1)$

New Q value = (1-alpha)(old Q value of current pair) + alpha(reward + gamma * Q(next_state))

```
# make env
# get num of states and actions
# initialize Q array with zeros
# alpha = 0.1, gamma = 1, num_episodes = 1000

for episode in range(num_episodes):
    state, info = env.reset()
    action = env.action_space.sample()
    terminated=False
    while not terminated:
        next_state, reward, terminated, truncated, info = env.step(action)
        next_action = env.action_space.sample()
        update_q_table(state, action, reward, next_state, next_action)
        state, action = next_state, next_action

def update_q_table(state, action, reward, next_state, next_action) # SARSA
    old_value = Q[state, action]
    next_state = Q[next_state, next_action]
    Q[state, action] = (1-alpha) * old_value + alpha * (reward + gamma *
next_value)
```

3.3 Q-Learning

Quality learning helps an agent learn the optimal Q table by interacting with an environment

Q-learning rule:

$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma(maxa_1)Q(s_1,a_1)$

This updates independent of taken actions

Q-learning is an off-policy learner

```
# create env, specify params
# initalize Q table with zeros
# iniate an empty list for reward per random episode

for episode in range(num_episodes):
    state, info = env.reset()
    terminated=False
    episode_reward = 0
    while not terminated:
        action = env.action_space.sample()
        new_state, reward, termianted, truncated, info = env.step(action)
        update_q_table(state, action, new_state)
        episode_reward += reward
        state = new_state

def update_q_table(state, action, reward, new_state): # Q learning rule
    old_value = Q[state, action]
    next_max = max(Q[new_state])
    Q[state, action] = (1-alpha) * old_value + alpha * (reward + gamma *
next_max)

# use same loop of performing an action, but now based on its learned Q-table
policy
# we then store each value to the empty list
# then average reward per learned episode using np.mean
```