MiniProjet1

October 1, 2023

1 Mini project Week 1

The mini-project for week 1 consists in coding Modified Policy Iteration and looking for the ideal number of critic updates at each iteration to converge as fast as possible on a set of 10x10 mazes. Your global report should explain or depict the set of mazes you have used, the best number of critic updates you found, and the number of iterations required on average to converge with these parameters. Think of having a key figure that makes it easy to figure out what is the best number of critic updates and how it compares to vanilla policy iteration.

1.1 Generalized Policy Iteration

1.1.1 Generalized Policy Iteration with the Q function

[ ]:
```python
def evaluate_q_K_steps(mdp: MazeMDPEnv,
                       policy: np.ndarray,
                       q: np.array, K: int,
                       threshold: float) -> np.ndarray:
    # Outputs the state value function of a policy
    nb_evaluations = 0
    for i in range(K):
        qold = q.copy()
        q = evaluate_one_step_q(mdp, qold, policy)
        nb_evaluations += 1

        # Test if convergence has been reached
        if (np.linalg.norm(q - qold)) < threshold:
            break

    return q, nb_evaluations
```

[ ]:
```python
def generalized_policy_iteration_q(mdp: MazeMDPEnv,
                                   threshold: float = 0.01,
                                   render: bool = True,
                                   K: int = 1) -> Tuple[np.ndarray,␣
 ↪List[float]]:
    """policy iteration over the q function."""

    nb_evaluations = 0
```

```
    # initial action values are set to 0
    q = np.zeros((mdp.nb_states, mdp.action_space.n))
    q_list = []
    policy = random_policy(mdp)

    stop = False

    video_recorder = VideoRecorder(mdp,
                                   "videos/GeneralizedPolicyIterationQ.mp4",
                                   enabled=render)
    mdp.init_draw("Generalized Policy iteration Q", recorder=video_recorder)

    while not stop:
        qold = q.copy()
        mdp.draw_v(q, recorder=video_recorder)

        # Step 1 : Policy evaluation
        q, evaluations = evaluate_q_K_steps(mdp, policy,
                                    qold, K, threshold)
        nb_evaluations += evaluations

        # Step 2 : Policy improvement
        policy = get_policy_from_q(q)
        nb_evaluations += env.action_space.n

        # Check convergence
        if (np.linalg.norm(q - qold)) <= threshold:
            stop = True
        q_list.append(np.linalg.norm(q))

    mdp.draw_v_pi(q, get_policy_from_q(q), recorder=video_recorder)
    video_recorder.close()

    return q, q_list, nb_evaluations
```

1.2 Generalized Policy Iteration with the V function

[ ]:
```
def evaluate_v_count(mdp: MazeMDPEnv,
                     policy: np.ndarray,
                     v: np.array, K: int,
                     threshold: float) -> Tuple[np.ndarray, int]:
    # Outputs the state value function of a policy
    # on supprime l'argument norms: np.array
    # To be completed...
    # v = np.zeros(mdp.nb_states)  # initial state values are set to 0
```

```
    nb_evaluations = 0
    for i in range(K):
        v_old = v.copy()
        v = evaluate_one_step_v(mdp, v_old, policy)
        nb_evaluations += 1

        # Test if convergence has been reached
        if (np.linalg.norm(v - v_old)) < threshold:
            break
    return v, nb_evaluations
```

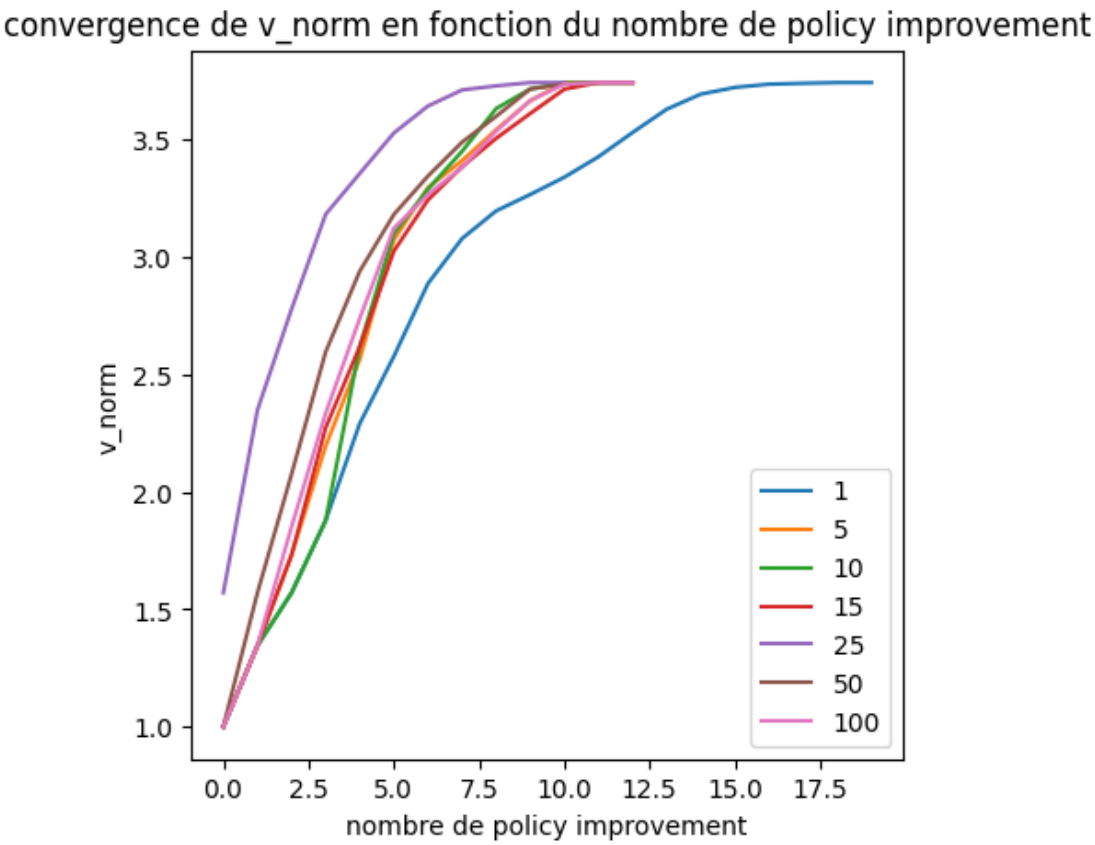
[ ]:
```
def generalized_policy_iteration_v(mdp: MazeMDPEnv, K: int = 1,
                                   threshold: float = 0.01,
                                   render: bool = True) -> Tuple[np.ndarray,␣
↪List[float], int]:
    """policy iteration over the V function."""

    nb_evaluations = 0

    v = np.zeros(mdp.nb_states)  # initial state values are set to 0
    v_list = []
    policy = random_policy(mdp)

    stop = False

    video_recorder = VideoRecorder(mdp, "videos/GeneralizedPolicyIterationV.
↪mp4", enabled=render)
    mdp.init_draw("Policy Iteration (V)", recorder=video_recorder)

    while not stop:
        vold = v.copy()
        mdp.draw_v(v, recorder=video_recorder)

        # Step 1 : Policy Evaluation
        v, evaluations = evaluate_v_count(mdp, policy, vold, K , threshold)
        nb_evaluations += evaluations

        # Step 2 : Policy Improvement
        policy = improve_policy_from_v(mdp, v, policy)
        nb_evaluations += mdp.action_space.n

        # Check convergence
        if (np.linalg.norm(v - vold)) < threshold:
            stop = True
        v_list.append(np.linalg.norm(v))

    mdp.draw_v_pi(v, get_policy_from_v(mdp, v), recorder=video_recorder)
```

```
    video_recorder.close()

    return v, v_list, nb_evaluations
```

Visualisation des résultats:

[ ]:
```
threshold=0.00001
K = [1, 5, 10, 15, 25, 50, 100]
list_nb_evaluations=[]

fig, ax = plt.subplots(figsize = (5,5))
for k in K:
  v, norm, nb_evaluations = generalized_policy_iteration_v(env, k, threshold,␣
 ↪render=False)
  list_nb_evaluations.append(nb_evaluations)
  print(k, ":", len(norm))
  ax.plot(range(len(norm)), norm, label=k)
  ax.set(title = "convergence de v_norm en fonction du nombre de policy␣
 ↪improvement",
         xlabel = "nombre de policy improvement",
         ylabel = "v_norm")
  ax.legend(loc='best')
plt.show()
```

```
1 : 20
5 : 12
10 : 13
15 : 13
25 : 11
50 : 13
100 : 13
```



Commentaire: On remarque qu'avec K=5, on obtient la même convegence qu'avec des K plus grands mais avec un nombre d'évaluation de policy optimal (12). On peut donc étudier l'impact de K sur la vitesse d'éxecution de l'algorithme `generalized_policy_iteration_v` afin de voir s'il existe un K optimal qui permet de converger le plus rapidement possible.

1.3 Etude de K sur plusieurs environnements

Etudions sur l'effet de K sur divers environnements. Cette fois, on se donne 15 labyrinthes de taille 10 par 10.

L'algorithme `generalized_policy_iteration_v` renvoit la variable `nb_evaluations` qui compte le nombre d'appel de la fonction `evaluate_one_step_v` et qui prend également en compte la complexité de la fonction `improve_policy_from_v`.

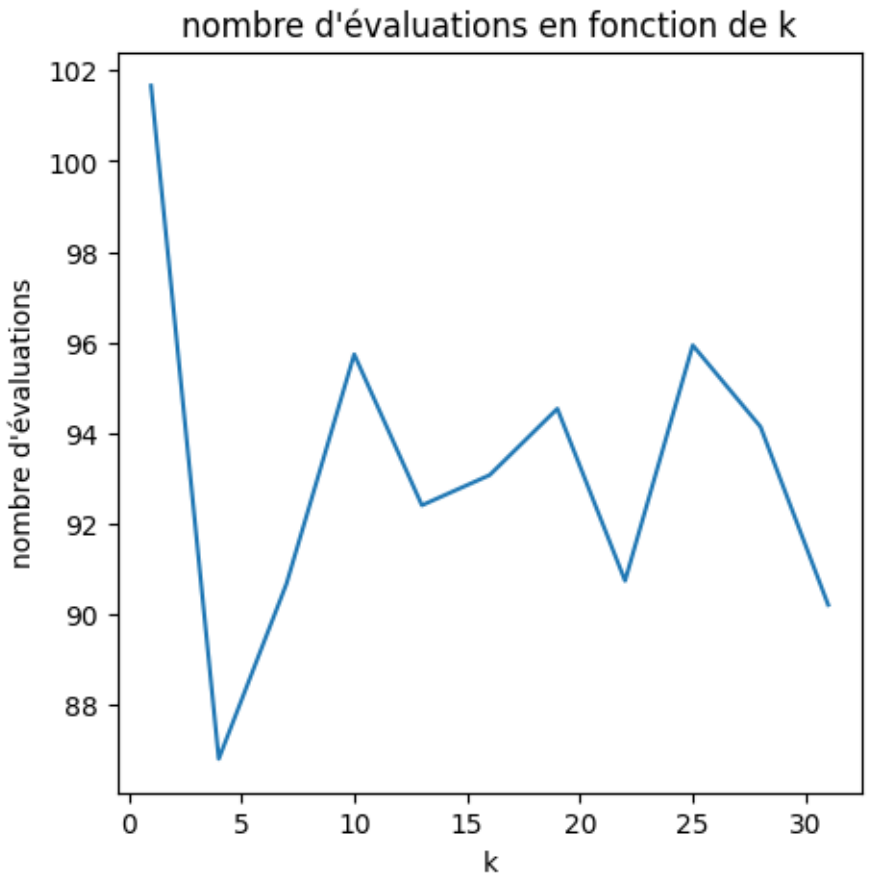
On considère que `improve_policy_from_v` est `mdp.action_space.n` fois plus grand en complexité que `evaluate_one_step_v` car il réalise les mêmes calculs en utilisant une boucle `for` sur `mdp.action_space.n`.

La variable `nb_evaluations` permet donc d'avoir une idée plus précise de la complexité totale de `generalized_policy_iteration_v`.

[ ]:
```
envs = []
nombre_environnements = 15

for _ in range(nombre_environnements):
  env = gym.make("MazeMDP-v0", kwargs={"width": 10, "height": 10, "ratio": 0.
 ↪2}, render_mode="rgb_array")
  env.metadata['render_fps'] = 1
  env.reset()
  # in dynamic programming, there is no agent moving in the environment
  env.set_no_agent()
  envs.append(env)
```

[ ]:
```
import matplotlib.pyplot as plt
import time
threshold=0.00001
K = range(1, 32, 3)

list_timers = [0]*len(K)
list_nb_evaluations = [0]*len(K)

for j, k in enumerate(K):
  print(f"\r etape = {j+1}/{len(K)}", end="")
  start_time = time.time()
  for i, env in enumerate(envs) :
    v, norm, nb_evaluations = generalized_policy_iteration_v(env, k, threshold,␣
 ↪render=False)
    list_nb_evaluations[j] += nb_evaluations
  list_timers[j] = (time.time() - start_time)/len(envs)
  list_nb_evaluations[j] /= len(envs)
```

[ ]:
```
fig, ax = plt.subplots(figsize = (5,5))

ax.plot(K, list_nb_evaluations)
ax.set(title = "nombre d'évaluations en fonction de k",
       xlabel = "k",
       ylabel = "nombre d'évaluations")
plt.show()
```

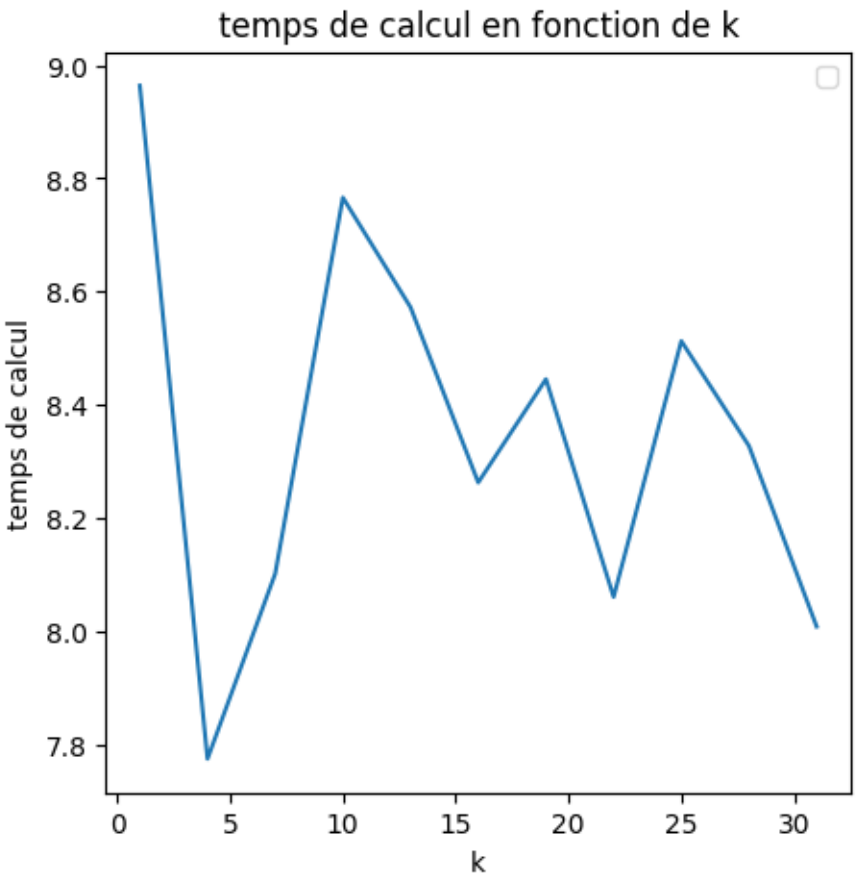


Commentaire : On constate que regarder le nombre d'évaluations en fonction de k rend la conclusion difficile à cause des fortes oscillations. Il faut donc rester prudent dans nos affirmations. On remarque toutefois qu'avec k=4, on obtient le nombre minimal d'évaluations. Prendre k=4 semble donc être le choix le plus judicieux.

On peux également observer le temps de calcul necessaire à la convergence de `generalized_policy_iteration_v` en fonction de k :

[ ]:
```
fig, ax = plt.subplots(figsize = (5,5))

ax.plot(K, list_timers)
ax.set(title = "temps de calcul en fonction de k",
       xlabel = "k",
       ylabel = "temps de calcul")
plt.show()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note
that artists whose label start with an underscore are ignored when legend() is
called with no argument.
```



Commentaire: On remarque d'abord que les deux courbes sont très similaires. Cela confirme qu'analyser le nombre d'épisodes comme on l'a fait est pertinent afin de trouver le K qui permet de converger le plus rapidement.

Encore une fois, k=4 minimise le temps de calcul et semble être le k à choisir dans `generalized_policy_iteration_v`.

Ce résultat reste toutefois à relativiser compte tenu de la faible taille de notre échantillon.

MiniProjet2

October 1, 2023

1 Mini project Week 2:

The mini-project for week 2 consists in coding a naive actor-critic algorithm and looking for the adequate policy and critic learning rates to converge as fast as possible on a set of 10x10 mazes. The global report should contain your actor-critic code, the values of hyper-parameters you used, and a key figure highlighting the learning behavior of your algorithm depending on the hyper-parameters, with a short discussion of your results.

1.1 Algorithme Actor Critic :

On commence par construire notre environnement qui est un labyrinthe de taille 10 x 10

[ ]:
```
import gymnasium as gym
import bbrl_gymnasium

# Environment with a little bit of negative reward (when hitting a wall)
env = gym.make("MazeMDP-v0", kwargs={"width": 10, "height": 10, "ratio": 0.2,␣
 ↪"hit": 0.0}, render_mode="rgb_array")
env.reset()

# in dynamic programming, there is no agent moving in the environment
env.init_draw("The maze")

NB_EPISODES = 1000
```

```
Output()
```

On réalise l'algorithme actor critic :

Ce dernier met à jour V selon l'hyperparamètre α_{critic} et met la policy à jour avec un hyper-paramètre α_{actor}.

Ici, l'agent tire une action au hasard en suivant la policy, en utilisant la fonction `np.random.choice`.

L'algorithme retourne le V optimal trouvé, la policy, ainsi que les listes des V et policy mis à jour à chaque itération. Cela permettra d'étudier la vitesse de convergence de l'approche actor-critic.

[ ]:
```
def actor_critic_v(mdp: MazeMDPEnv,
                   nb_episodes: int,
                   alpha_critic: float,
                   alpha_actor: float,
```

```
                    timeout: int,
                    return_lists: bool = True
                   ) -> float:

    v = np.zeros(mdp.nb_states)  # initial action values are set to 0
    policy = np.ones((mdp.nb_states, mdp.action_space.n)) / mdp.action_space.n␣
↪# action probabilities are uniform

    mdp.set_timeout(timeout)   # episode length

    v_list = [np.array(v)]
    policy_list = [np.argmax(policy, axis=1)]

    for i in range(nb_episodes):
        # Draw the first state of episode i using a uniform distribution over␣
↪all the states
        x, _ = mdp.reset(uniform=True)
        done = False

        while not done:
            u = np.random.choice(range(mdp.action_space.n), p = policy[x])

            # Perform a step of the MDP
            y, r, done, *_ = mdp.step(u)

            # critic
            delta = r + mdp.gamma * v[y] * (1 - done) - v[x]
            v[x] += alpha_critic * delta

            # actor
            policy[x, u] = max(0.02, policy[x, u] + alpha_actor * delta)
            policy[x] /= policy[x].sum()

            # Update the agent position
            x = y
        if return_lists:
            v_list.append(np.array(v))
            policy_list.append(np.argmax(policy, axis = 1))

    return v, policy.argmax(axis = 1), v_list, policy_list
```

On exécute et on visualise la policy obtenue:

[ ]:
```
v, policy, _ , _ = actor_critic_v(env, NB_EPISODES, 0.5, 0.4, timeout=40,␣
 ↪return_lists = False)
env.draw_v_pi(v, policy)
```

Output()

On souhaite comparer nos résultats avec le V* et la policy* obtenus avec la fonction `policy_iteration_v`

[ ]:
```
def improve_policy_from_v(mdp: MazeMDPEnv, v: np.ndarray,
                          policy: np.ndarray) -> np.ndarray:
    # Improves a policy given the state values
    for x in range(mdp.nb_states):  # for each state x
        # Compute the value of the state x for each action u of the MDP action space
        if x in mdp.terminal_states:
            policy[x] = np.argmax(mdp.r[x, :])
        else:
            v_temp = np.zeros(mdp.action_space.n)
            for u in range(mdp.action_space.n):
                # Process sum of the values of the neighbouring states
                summ = 0
                for y in range(mdp.nb_states):
                    summ = summ + mdp.P[x, u, y] * v[y]
                v_temp[u] = mdp.r[x, u] + mdp.gamma * summ

            for u in range(mdp.action_space.n):
                if v_temp[u] > v_temp[policy[x]]:
                    policy[x] = u
    return policy

def evaluate_one_step_v(mdp: MazeMDPEnv, v: np.ndarray,
                        policy: np.ndarray) -> np.ndarray:
    # Outputs the state value function after one step of policy evaluation
    # Corresponds to one application of the Bellman Operator
    v_new = np.zeros(mdp.nb_states)  # initial state values are set to 0
    for x in range(mdp.nb_states):  # for each state x
        # Compute the value of the state x for each action u of the MDP action space
        if x in mdp.terminal_states:
            v_new[x] = mdp.r[x, policy[x]]
        else:
            # Process sum of the values of the neighbouring states
            summ = 0
            for y in range(mdp.nb_states):
                summ = summ + mdp.P[x, policy[x], y] * v[y]
            v_new[x] = mdp.r[x, policy[x]] + mdp.gamma * summ
    return v_new

def evaluate_v(mdp: MazeMDPEnv, policy: np.ndarray) -> np.ndarray:
    # Outputs the state value function of a policy
    v = np.zeros(mdp.nb_states)  # initial state values are set to 0
    stop = False
```

```
    while not stop:
        vold = v.copy()
        v = evaluate_one_step_v(mdp, vold, policy)

        # Test if convergence has been reached
        if (np.linalg.norm(v - vold)) < 0.01:
            stop = True
    return v

def policy_iteration_v(mdp: MazeMDPEnv) -> Tuple[np.ndarray, List[float]]:
    # policy iteration over the v function
    v = np.zeros(mdp.nb_states)  # initial state values are set to 0
    policy = random_policy(mdp)

    stop = False

    while not stop:
        vold = v.copy()

        # Step 1 : Policy Evaluation
        v = evaluate_v(mdp, policy)

        # Step 2 : Policy Improvement
        policy = improve_policy_from_v(mdp, v, policy)

        # Check convergence
        if (np.linalg.norm(v - vold)) < 0.01:
            stop = True

    return v, policy
```

[ ]:
```
v_star, policy_star = policy_iteration_v(env)
env.draw_v_pi(v_star, policy_star)
```

Output()

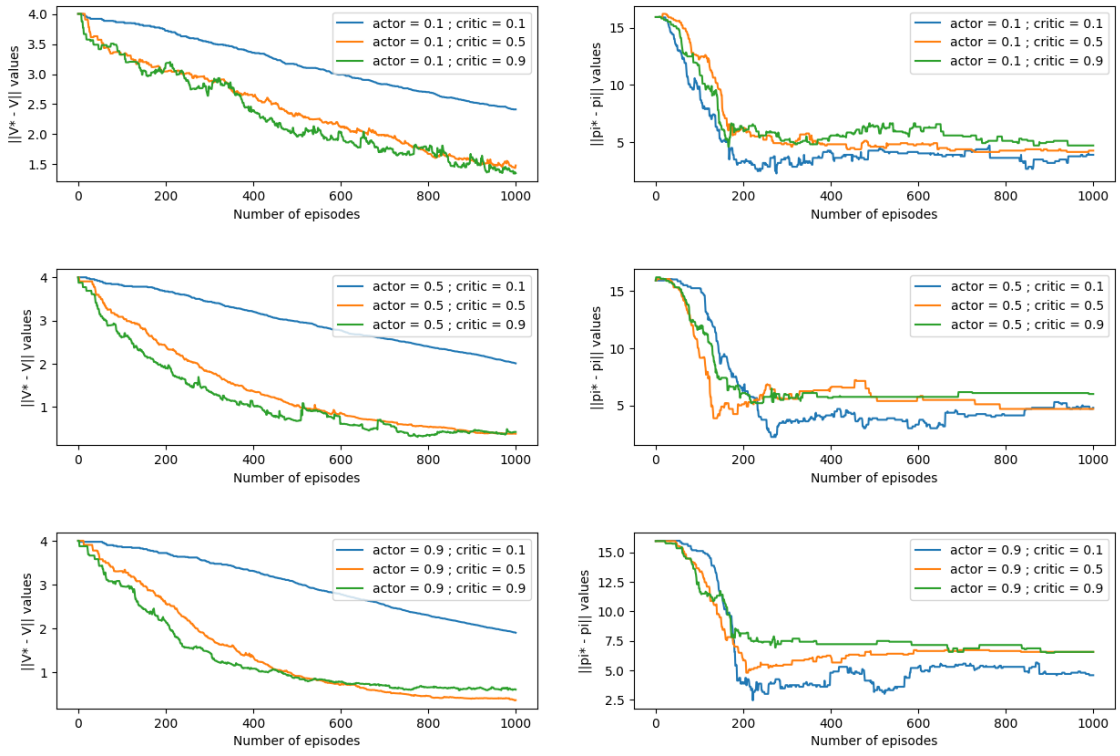
1.1.1 **Comparaison avec V* et policy*:**

On va tracer les normes des différences entre V* et V ainsi qu'entre policy* et policy en fonction du nombre d'épisode afin d'analyser la vitesse de convergence de l'algorithme `actor_critic_v`. On essaye divers paramètres α_{critic} et α_{actor}

[ ]:
```
def plot_compare_actor_critic(env, nb_episodes, list_alpha_critic,␣
 ↪list_alpha_actor, timeout):
        figure, axes = plt.subplots(len(list_alpha_actor), 2, figsize = (15,10))

        for alpha_critic in list_alpha_critic :
                for i, alpha_actor in enumerate(list_alpha_actor):
```

```
                        v, _, v_list, policy_list = actor_critic_v(env,␣
↪nb_episodes, alpha_critic, alpha_actor, timeout)

                        axes[i, 0].plot(range(len(v_list)), [np.linalg.
↪norm(v_star - v) for v in v_list],
                                        label = f"actor = {alpha_actor} ;␣
↪critic = {alpha_critic}")
                        axes[i, 0].set(xlabel = 'Number of episodes', ylabel =␣
↪'||V* - V|| values')
                        axes[i, 0].legend(loc='best')

                        axes[i, 1].plot(range(len(v_list)), [np.linalg.
↪norm(policy_star - policy) for policy in policy_list],
                                        label = f"actor = {alpha_actor} ;␣
↪critic = {alpha_critic}")
                        axes[i, 1].set(xlabel = 'Number of episodes', ylabel =␣
↪'||pi* - pi|| values')
                        axes[i, 1].legend(loc='best')

        plt.subplots_adjust(hspace=0.5)
        plt.show()
```

[ ]:
```
list_alpha_critic = [0.1, 0.5, 0.9]
list_alpha_actor = [0.1, 0.5, 0.9]
plot_compare_actor_critic(env, 1000, list_alpha_critic, list_alpha_actor, 40)
```

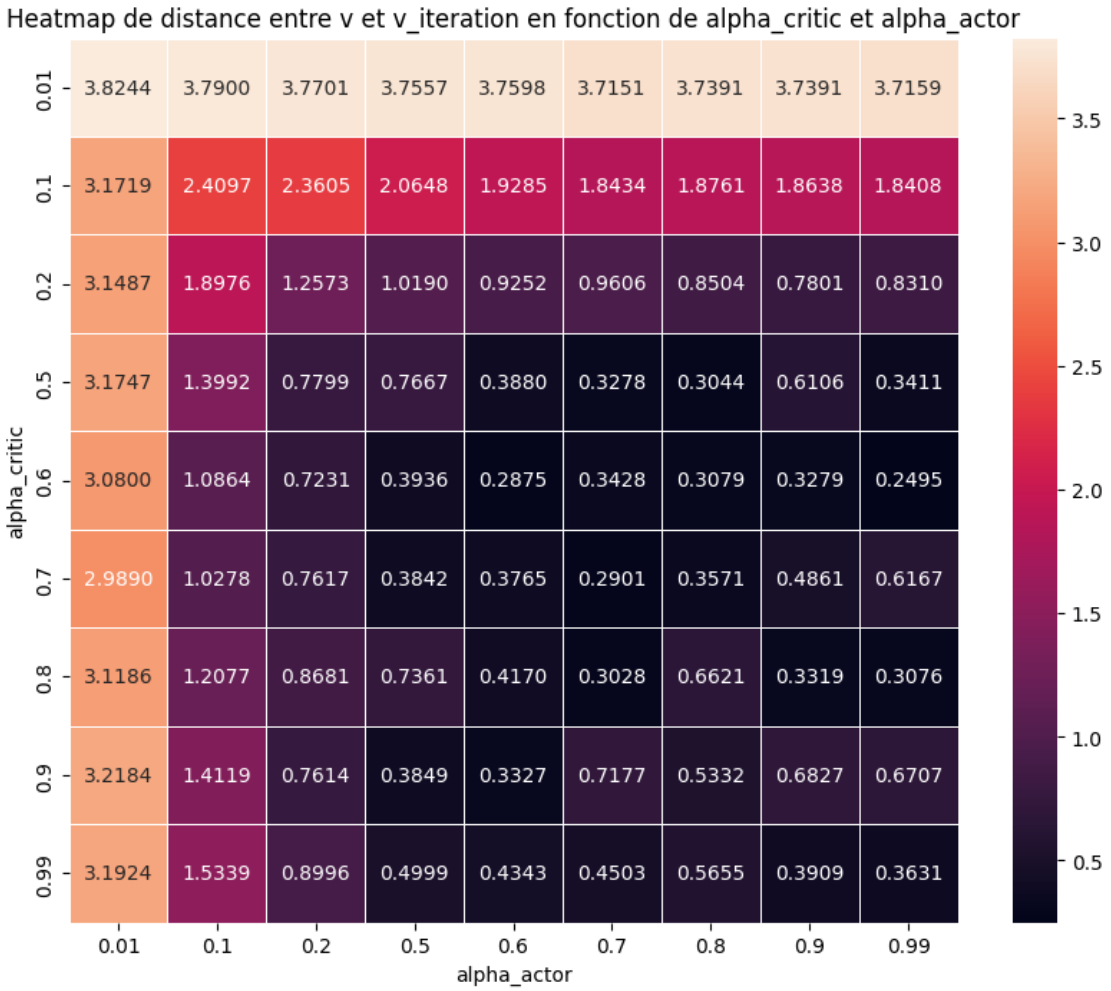


D'une part, α_{critic} règle la vitesse à laquelle l'agent ajuste ses estimations des récompenses et de la valeur des états. Ici, on voit bien qu'un α_{critic} trop faible rend l'amélioration de V très lente mais plus sûre. A l'inverse, un grand α_{critic} rend l'ajustement de V plus rapide mais aussi moins stable, avec beaucoup plus d'oscillations.

D'autre part, α_{actor} contrôle la vitesse selon laquelle l'agent améliore sa policy. On observe donc des convergences plus fortes avec des α_{actor} élevés.

Etudions plus en détail l'influence des hyperparamètres α_{actor} et α_{critic} afin de trouver les meilleurs. On représente la distance entre V et V* en testant plusieurs hyperparamètres à l'aide d'une heatmap.

[ ]:
```
list_alpha_critic = [0.01, 0.1, 0.2, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99]
list_alpha_actor = [0.01, 0.1, 0.2, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99]

liste_normes = []
for alpha_critic in list_alpha_critic :
    normes_critic = []
    for alpha_actor in list_alpha_actor :
        v, _, _, _ = actor_critic_v(env, nb_episodes = 1000,␣
 ↪alpha_critic=alpha_critic, alpha_actor=alpha_actor,
                                    timeout = 40, return_lists = False)
        normes_critic.append(np.linalg.norm(v_star - v))
    liste_normes.append(normes_critic)

# heatmap
critic_names = [str(a) for a in list_alpha_critic]
actor_names = [str(g) for g in list_alpha_actor]

figure, ax = plt.subplots(figsize = (10,8))
sns.heatmap(liste_normes, linewidth=0.5, annot = True, fmt=".4f", ax = ax)
ax.set(title = "Heatmap de distance entre v et v_iteration en fonction de␣
 ↪alpha_critic et alpha_actor",
       xlabel="alpha_actor", ylabel="alpha_critic")
ax.set_xticklabels(actor_names)
ax.set_yticklabels(critic_names)

plt.show()

i, j = np.argmin(liste_normes) // len(list_alpha_critic), np.
 ↪argmin(liste_normes) % len(list_alpha_critic)
print(f"best_alpha_critic = {list_alpha_critic[i]}, best_alpha_actor =␣
 ↪{list_alpha_actor[j]}, "
      f"distance = {liste_normes[i][j]}")
```




```
best_alpha_critic = 0.6, best_alpha_actor = 0.99, distance = 0.24949852448929613
```

Ici, les hyperparamètres qui permettent d'obtenir la meilleure convergence vers V* sont $\alpha_{critic} = 0.6$ et $\alpha_{actor} = 0.99$

1.2 Moyenne sur plusieurs environnements:

On va étudier la meilleure convergence en utilisant plusieurs environnements et en faisant une moyenne pour plus de précision:

[ ]:
```python
import gymnasium as gym
import bbrl_gymnasium

envs = []
nombre_environnements = 10

for _ in range(nombre_environnements):
```

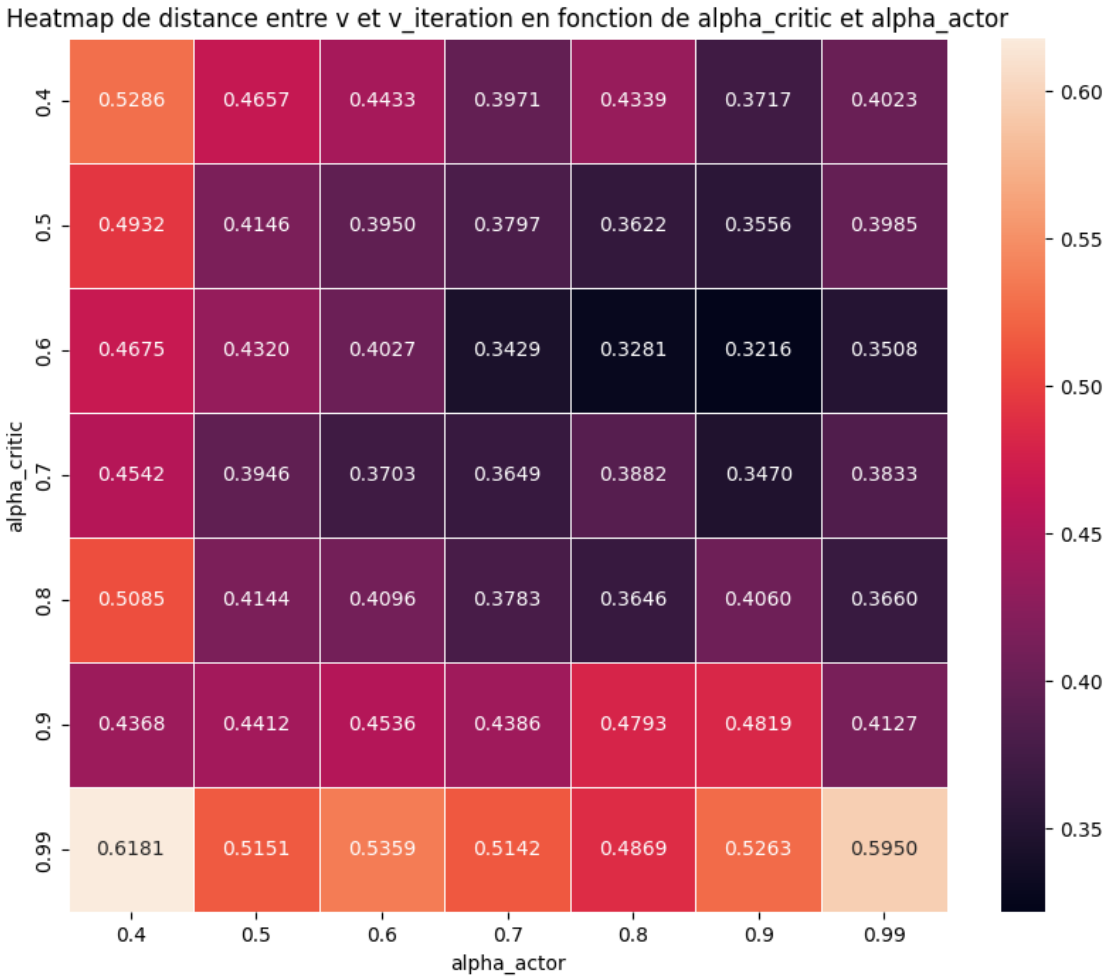
```
  env = gym.make("MazeMDP-v0", kwargs={"width": 10, "height": 10, "ratio": 0.
↪2}, render_mode="rgb_array")
  env.metadata['render_fps'] = 1
  env.reset()
  # in dynamic programming, there is no agent moving in the environment
  env.set_no_agent()
  envs.append(env)
```

[ ]:
```
list_alpha_critic = [0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99]
list_alpha_actor = [0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99]

moyennes_normes = np.zeros([len(list_alpha_critic), len(list_alpha_actor)])

for e, env in enumerate(envs):
    print(f"environnement {e+1}/{len(envs)}")
    v_star, policy_star = policy_iteration_v(env)
    for i, alpha_critic in enumerate(list_alpha_critic) :
        for j, alpha_actor in enumerate(list_alpha_actor) :
            v, _, _, _ = actor_critic_v(env, nb_episodes = 1000,␣
 ↪alpha_critic=alpha_critic, alpha_actor=alpha_actor,
                                        timeout = 40, return_lists = False)
            moyennes_normes[i][j] += np.linalg.norm(v_star - v)

moyennes_normes /= len(envs)
```

```
environnement 1/10
environnement 2/10
environnement 3/10
environnement 4/10
environnement 5/10
environnement 6/10
environnement 7/10
environnement 8/10
environnement 9/10
environnement 10/10
```

[ ]:
```
# heatmap
critic_names = [str(a) for a in list_alpha_critic]
actor_names = [str(g) for g in list_alpha_actor]

figure, ax = plt.subplots(figsize = (10,8))
sns.heatmap(moyennes_normes, linewidth=0.5, annot = True, fmt=".4f", ax = ax)
ax.set(title = "Heatmap de distance entre v et v_iteration en fonction de␣
 ↪alpha_critic et alpha_actor",
       xlabel="alpha_actor", ylabel="alpha_critic")
ax.set_xticklabels(actor_names)
ax.set_yticklabels(critic_names)
```

```
plt.show()

i, j = np.argmin(moyennes_normes) // len(list_alpha_critic), np.
 ↪argmin(moyennes_normes) % len(list_alpha_critic)
print(f"best_alpha_critic = {list_alpha_critic[i]}, best_alpha_actor =␣
 ↪{list_alpha_actor[j]}, "
      f"distance = {moyennes_normes[i][j]}")
```




```
best_alpha_critic = 0.6, best_alpha_actor = 0.9, distance = 0.32164466827878774
```

En moyenne, sur 10 environnements, on a que les meilleurs hyperparamètres sont $\alpha_{critic} = 0.6$ et $\alpha_{actor} = 0.9$. Pour un même nombre d'épisodes, ce sont ces hyperparamètres qui permettent de se rapprocher le plus de V* en norme. On les choisit donc.