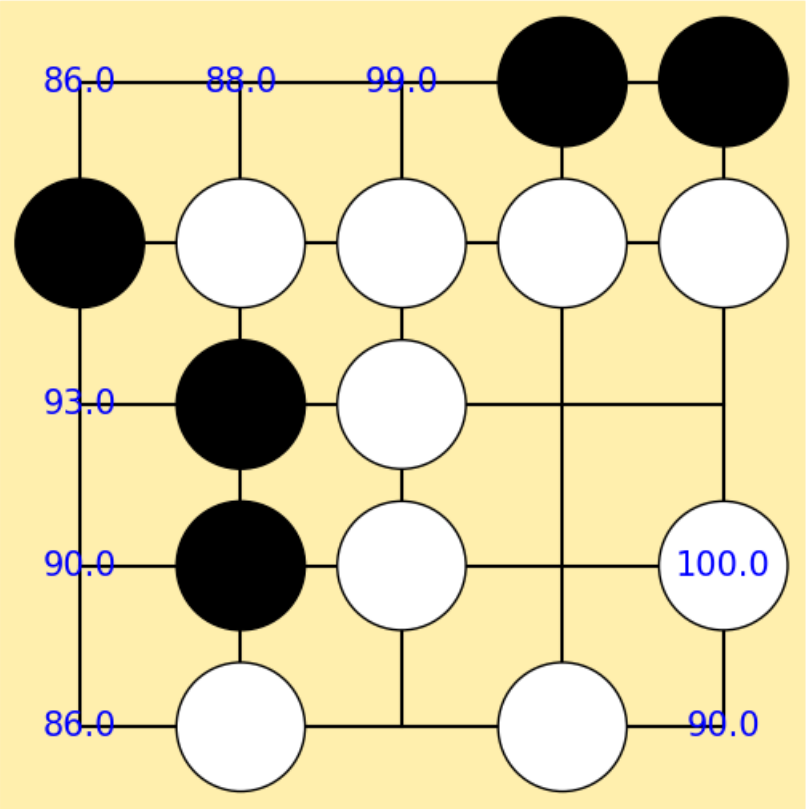
KAIST

Fall Semester
December 2019

Project #2
Deep Reinforcement Learning and AlphaGO Zero



Student : David Albert
Professor : Sae-Young Chung

Contents

Part I

Task 1: Q-learning for solving a single maze

1 Calculating the optimal action value function given any maze

1.1 Method

The formulation of the problem of the maze is as follow :

- Reward at each step is -1
- Episode terminates when the agent reaches the terminal state
- The result of taking an action in each state is deterministic such that from (y, x):
  a) Going 'up' results to move to the upper state (y-1, x) if there is no wall in (y-1, x) and stay in (y, x) otherwise
  b) Going 'down' as the same results considering cell (y+1, x)
  c) Going 'left' as the same results considering cell (y, x-1)
  d) Going 'right' as the same results considering cell (y, x+1)

In this case it is obvious that the optimal state value function $v_*(s)$ is nothing but the negation of the shortest path from s to the terminal state s_{term}. In other words, we can write : $\forall s,\ v_*(s) = -d(s, s_{term})$ where $d(s, s')$ is the length of the shortest path in graph from node s to s'.

Thus, the method that I used in order to analytically calculate the optimal action value function given any maze is as follow:

1) Build the graph corresponding to the maze (I used an adjacency list data structure because the graph is sparse and so an adjacency matrix is not convenient)
   → see Figure I.1 and code AdjacencyList.build_from_maze
2) Find the shortest path from node (y_{term}, x_{term}) to every nodes of the graph, i.e. $\forall (y, x),\ d\left((y, x), (y_{term}, x_{term})\right)$
   → see code AdjacencyList.bfs
3) Build the optimal state value function , i.e. $\forall (y, x),\ v_*(y, x) = -d\left((y, x), (y_{term}, x_{term})\right)$
   → see code optimal_state_value
4) Calculate the optimal action value function from , i.e. $q_*(s, a) = \sum_{s',r} p\left(s', r|s, a\right)\left[r + \gamma v_*(s')\right]$
   → see code optimal_action_value

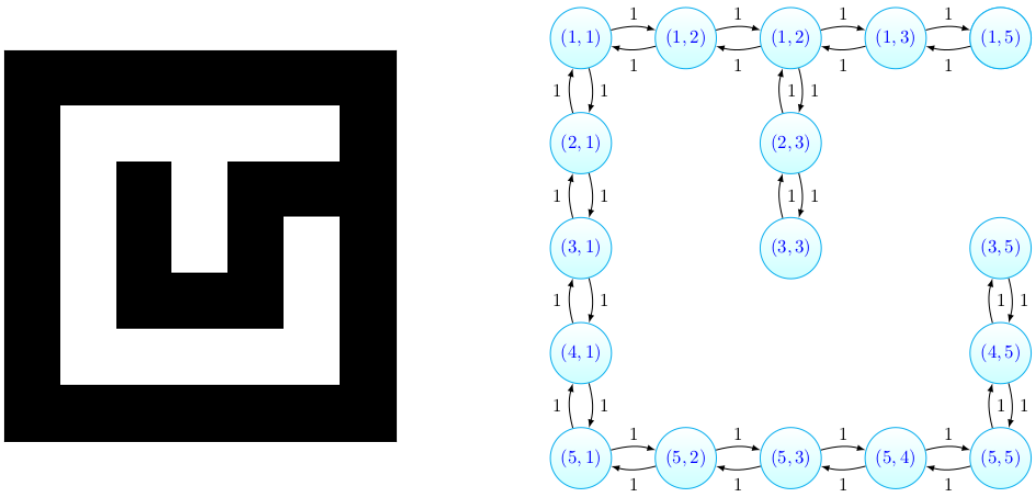


Figure I.1 – 7x7 Maze and its graph representation

1.2 Code

```
# Return the next state given current state and action.
def next_state(maze, curr_state, action):
    if maze[curr_state]:
        raise ValueError
    if action==0: # Up
        next_state = (curr_state[0]-1, curr_state[1])
    if action==1: # Down
        next_state = (curr_state[0]+1, curr_state[1])
    if action==2: # Left
        next_state = (curr_state[0], curr_state[1]-1)
    if action==3: # Right
        next_state = (curr_state[0], curr_state[1]+1)
    return curr_state if maze[next_state] else next_state
```

```
# Return the state value function given any maze and a terminal state.
def optimal_state_value(maze, init_state):
    # Create graph from maze
    adj_list = AdjacencyList()
    adj_list.build_from_maze(maze)
    # Calculate shortest path from init_state
    shortest_path = adj_list.bfs(init_state)
    # Calculate state value from list of shortest pathes
    state_value = np.zeros_like(maze)
    for coord, sh_path in shortest_path.items():
        state_value[coord] = - sh_path
    return state_value
```

```
# Return the action value function given any maze and a terminal state.
def optimal_action_value(maze, init_state):
    # Calculate state value function
    opt_state_value = optimal_state_value(maze, init_state)
    # Get coordinates of cell without wall
    tmp = np.where(maze==0)
    vertex = list(zip(tmp[0],tmp[1]))
    # Calculate Q-value from state-value
    q = np.zeros((maze.shape[0], maze.shape[1], 4))
    for state in vertex:
        if state == init_state:
            continue
```

```
        for action in range(4):
            q[state][action]=-1+opt_state_value[next_state(maze, state,action)]
    return q
```

```
    # Class that represents a graph by an adjacency list
    class AdjacencyList():
    def __init__(self):
    self.nb_vertex = None
    self.adj = {}

    # Build a graph representation of the maze
    # -> each vertex is a state where the agent can go
    def build_from_maze(self, maze):
        self.nb_vertex = sum(sum(maze))
        tmp = np.where(maze==0)
            available_v = list(zip(tmp[0],tmp[1]))
            for _, v in enumerate(available_v):
                v_next = [  (v[0]-1,v[1]), # vertex above
                            (v[0]+1,v[1]), # vertex below
                            (v[0],v[1]-1), # vertex at left
                            (v[0],v[1]+1)] # vertex at right
                self.adj[v] = [node for node in v_next if not maze[node]]

    # Breadth-first search (bfs) algorithm that allows
    # to calculate the distance from a vertex v_init to others
    def bfs(self, v_init):
        queue = deque()
        dist = {}
        for key, _ in self.adj.items():
            dist[key] = np.inf
        dist[v_init] = 0
        visited = {}
        for key, _ in self.adj.items():
            visited[key] = False
        visited[v_init] = True
        queue.append(v_init)
        while (len(queue)>0):
            vertex = queue[0]
            queue.popleft()
            for v_next in self.adj[vertex]:
                if visited[v_next]: continue
                visited[v_next] = True
                dist[v_next] = dist[vertex] +1
                queue.append(v_next)
        return dis
```

2 First Case : $\epsilon = 1.$ constant

In this first case, we fix $\epsilon = 1$. In this setting, the agent only take random actions. This is a naive approach and it is necessary to increase the parameter max_steps because for random action, if this parameter is too low, the agent may never (or a few) reach the terminal state while training and thus the Q-value q associated to this states will converge very slowly to the q_*. We need to design max_steps such that it allows the agent to reach the terminal state most of the time. The number of episodes is less important because since the agent is able to reach every states at each episodes, we need to design $n\_episodes$ such that it is enough large to allows the agent to find the optimal policy and that the action-value function become close to the optimal.

A good choice is thus:

- $max\_steps \approx 6000$ and $n\_episodes \approx 80$

Maze informations:

- length of shortest path: 53
- starting point: (9, 3) → green cell on fig I.2
- goal: (8, 11) → red cell on fig I.2

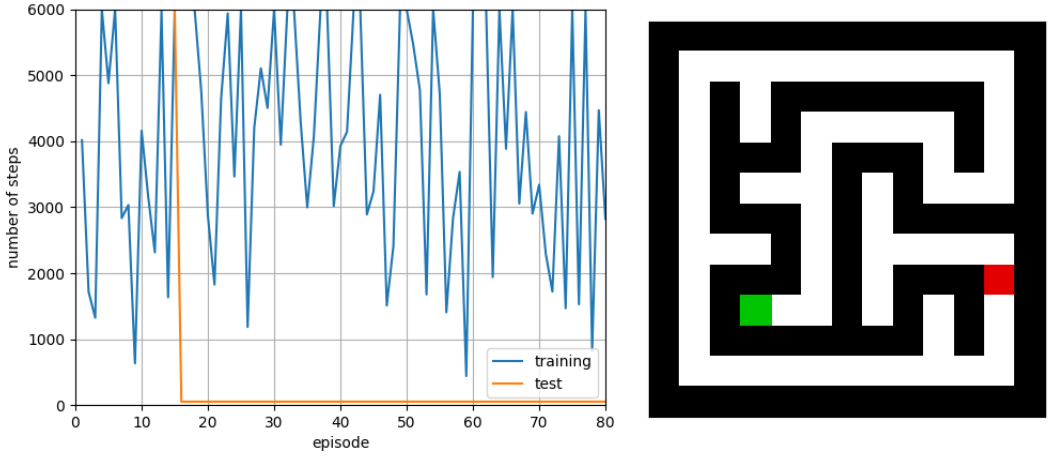


Figure I.2 – Number of steps per episode with $\epsilon - greedy$ policy and $greedy$ policy based on current q-value

We can see that as soon as the 18^{th} episode the greedy policy is optimal for this maze. We can thus conclude that choosing $n\_episodes \geq 30$ is enough to infer the optimal greedy policy but taking $n\_episodes \approx 80$ allows to have a very good estimation of the optimal action value function as shown below.

Results Q-table

Q-value for action 'up'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−24.000	−25.000	−26.000	−27.000	−28.000	−29.000	−29.999	−30.999	−31.999	−32.999	−33.999	0.000
0.000	−24.000	0.000	−26.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−33.999	0.000
0.000	−23.000	0.000	−27.000	0.000	−45.999	−44.999	−43.999	−42.999	−41.999	0.000	−34.999	0.000
0.000	−22.000	0.000	0.000	0.000	−45.999	0.000	0.000	0.000	−41.999	0.000	−35.999	0.000
0.000	−21.000	0.000	−49.998	−48.998	−46.999	0.000	0.000	0.000	−40.999	−38.999	−36.999	0.000
0.000	−20.000	0.000	0.000	0.000	−47.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−19.000	−19.000	−20.000	0.000	−48.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−18.000	0.000	0.000	0.000	−49.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−17.000	0.000	−53.998	−52.998	−50.998	0.000	0.000	0.000	−8.000	0.000	−1.000	0.000
0.000	−16.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−8.000	0.000	−2.000	0.000
0.000	−15.000	−13.000	−12.000	−11.000	−10.000	−9.000	−8.000	−7.000	−7.000	−5.000	−3.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Q-value for action 'down'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−23.000	−25.000	−27.000	−27.000	−28.000	−28.999	−29.999	−30.999	−31.999	−32.999	−34.999	0.000
0.000	−22.000	0.000	−27.999	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−35.999	0.000
0.000	−21.000	0.000	−28.000	0.000	−46.999	−44.999	−43.999	−42.999	−40.999	0.000	−36.999	0.000
0.000	−20.000	0.000	0.000	0.000	−47.999	0.000	0.000	0.000	−39.999	0.000	−37.999	0.000
0.000	−19.000	0.000	−49.998	−48.998	−48.998	0.000	0.000	0.000	−39.999	−38.999	−37.999	0.000
0.000	−18.000	0.000	0.000	0.000	−49.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−17.000	−19.000	−20.000	0.000	−50.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−16.000	0.000	0.000	0.000	−51.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−15.000	0.000	−53.998	−52.998	−51.998	0.000	0.000	0.000	−7.000	0.000	−3.000	0.000
0.000	−14.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−6.000	0.000	−4.000	0.000
0.000	−14.000	−13.000	−12.000	−11.000	−10.000	−9.000	−8.000	−7.000	−6.000	−5.000	−4.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Q-value for action 'left'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−24.000	−24.000	−25.000	−26.000	−27.000	−28.000	−28.999	−29.999	−30.999	−31.999	−32.999	0.000
0.000	−23.000	0.000	−26.999	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−34.999	0.000
0.000	−22.000	0.000	−27.999	0.000	−45.999	−45.999	−44.999	−43.999	−42.999	0.000	−35.999	0.000
0.000	−21.000	0.000	0.000	0.000	−46.999	0.000	0.000	0.000	−40.999	0.000	−36.999	0.000
0.000	−20.000	0.000	−49.998	−49.998	−48.998	0.000	0.000	0.000	−39.999	−39.999	−38.999	0.000
0.000	−19.000	0.000	0.000	0.000	−48.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−18.000	−18.000	−19.000	0.000	−49.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−17.000	0.000	0.000	0.000	−50.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−16.000	0.000	−53.998	−53.998	−52.998	0.000	0.000	0.000	−8.000	0.000	−2.000	0.000
0.000	−15.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−7.000	0.000	−3.000	0.000
0.000	−14.000	−14.000	−13.000	−12.000	−11.000	−10.000	−9.000	−8.000	−7.000	−6.000	−5.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Q-value for action 'right'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−25.000	−26.000	−27.000	−28.000	−29.000	−29.999	−30.999	−31.999	−32.999	−33.999	−33.999	0.000
0.000	−23.000	0.000	−26.999	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−34.999	0.000
0.000	−22.000	0.000	−28.000	0.000	−44.999	−43.999	−42.999	−41.999	−41.999	0.000	−35.999	0.000
0.000	−21.000	0.000	0.000	0.000	−46.999	0.000	0.000	0.000	−40.999	0.000	−36.999	0.000
0.000	−20.000	0.000	−48.998	−47.998	−47.998	0.000	0.000	0.000	−38.999	−37.999	−37.999	0.000
0.000	−19.000	0.000	0.000	0.000	−48.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−19.000	−20.000	−20.000	0.000	−49.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−17.000	0.000	0.000	0.000	−50.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−16.000	0.000	−52.998	−51.998	−51.998	0.000	0.000	0.000	−8.000	0.000	−2.000	0.000
0.000	−15.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−7.000	0.000	−3.000	0.000
0.000	−13.000	−12.000	−11.000	−10.000	−9.000	−8.000	−7.000	−6.000	−5.000	−4.000	−4.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-table

Optimal Q-value for action 'up'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−24.000	−25.000	−26.000	−27.000	−28.000	−29.000	−30.000	−31.000	−32.000	−33.000	−34.000	0.000
0.000	−24.000	0.000	−26.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−34.000	0.000
0.000	−23.000	0.000	−27.000	0.000	−46.000	−45.000	−44.000	−43.000	−42.000	0.000	−35.000	0.000
0.000	−22.000	0.000	0.000	0.000	−46.000	0.000	0.000	0.000	−42.000	0.000	−36.000	0.000
0.000	−21.000	0.000	−50.000	−49.000	−47.000	0.000	−8.000	0.000	−41.000	−39.000	−37.000	0.000
0.000	−20.000	0.000	0.000	0.000	−48.000	0.000	−8.000	0.000	0.000	0.000	0.000	0.000
0.000	−19.000	−19.000	−20.000	0.000	−49.000	0.000	−7.000	−5.000	−4.000	−3.000	−2.000	0.000
0.000	−18.000	0.000	0.000	0.000	−50.000	0.000	−6.000	0.000	0.000	0.000	0.000	0.000
0.000	−17.000	0.000	−54.000	−53.000	−51.000	0.000	−7.000	0.000	−8.000	0.000	−1.000	0.000
0.000	−16.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−8.000	0.000	−2.000	0.000
0.000	−15.000	−13.000	−12.000	−11.000	−10.000	−9.000	−8.000	−7.000	−7.000	−5.000	−3.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'down'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−23.000	−25.000	−27.000	−27.000	−28.000	−29.000	−30.000	−31.000	−32.000	−33.000	−35.000	0.000
0.000	−22.000	0.000	−28.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−36.000	0.000
0.000	−21.000	0.000	−28.000	0.000	−47.000	−45.000	−44.000	−43.000	−41.000	0.000	−37.000	0.000
0.000	−20.000	0.000	0.000	0.000	−48.000	0.000	0.000	0.000	−40.000	0.000	−38.000	0.000
0.000	−19.000	0.000	−50.000	−49.000	−49.000	0.000	−7.000	0.000	−40.000	−39.000	−38.000	0.000
0.000	−18.000	0.000	0.000	0.000	−50.000	0.000	−6.000	0.000	0.000	0.000	0.000	0.000
0.000	−17.000	−19.000	−20.000	0.000	−51.000	0.000	−7.000	−5.000	−4.000	−3.000	−1.000	0.000
0.000	−16.000	0.000	0.000	0.000	−52.000	0.000	−8.000	0.000	0.000	0.000	0.000	0.000
0.000	−15.000	0.000	−54.000	−53.000	−52.000	0.000	−8.000	0.000	−7.000	0.000	−3.000	0.000
0.000	−14.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−6.000	0.000	−4.000	0.000
0.000	−14.000	−13.000	−12.000	−11.000	−10.000	−9.000	−8.000	−7.000	−6.000	−5.000	−4.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'left'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−24.000	−24.000	−25.000	−26.000	−27.000	−28.000	−29.000	−30.000	−31.000	−32.000	−33.000	0.000
0.000	−23.000	0.000	−27.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−35.000	0.000
0.000	−22.000	0.000	−28.000	0.000	−46.000	−46.000	−45.000	−44.000	−43.000	0.000	−36.000	0.000
0.000	−21.000	0.000	0.000	0.000	−47.000	0.000	0.000	0.000	−41.000	0.000	−37.000	0.000
0.000	−20.000	0.000	−50.000	−50.000	−49.000	0.000	−8.000	0.000	−40.000	−40.000	−39.000	0.000
0.000	−19.000	0.000	0.000	0.000	−49.000	0.000	−7.000	0.000	0.000	0.000	0.000	0.000
0.000	−18.000	−18.000	−19.000	0.000	−50.000	0.000	−6.000	−6.000	−5.000	−4.000	−3.000	0.000
0.000	−17.000	0.000	0.000	0.000	−51.000	0.000	−7.000	0.000	0.000	0.000	0.000	0.000
0.000	−16.000	0.000	−54.000	−54.000	−53.000	0.000	−8.000	0.000	−8.000	0.000	−2.000	0.000
0.000	−15.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−7.000	0.000	−3.000	0.000
0.000	−14.000	−14.000	−13.000	−12.000	−11.000	−10.000	−9.000	−8.000	−7.000	−6.000	−5.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'right'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−25.000	−26.000	−27.000	−28.000	−29.000	−30.000	−31.000	−32.000	−33.000	−34.000	−34.000	0.000
0.000	−23.000	0.000	−27.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−35.000	0.000
0.000	−22.000	0.000	−28.000	0.000	−45.000	−44.000	−43.000	−42.000	−42.000	0.000	−36.000	0.000
0.000	−21.000	0.000	0.000	0.000	−47.000	0.000	0.000	0.000	−41.000	0.000	−37.000	0.000
0.000	−20.000	0.000	−49.000	−48.000	−48.000	0.000	−8.000	0.000	−39.000	−38.000	−38.000	0.000
0.000	−19.000	0.000	0.000	0.000	−49.000	0.000	−7.000	0.000	0.000	0.000	0.000	0.000
0.000	−19.000	−20.000	−20.000	0.000	−50.000	0.000	−5.000	−4.000	−3.000	−2.000	−2.000	0.000
0.000	−17.000	0.000	0.000	0.000	−51.000	0.000	−7.000	0.000	0.000	0.000	0.000	0.000
0.000	−16.000	0.000	−53.000	−52.000	−52.000	0.000	−8.000	0.000	−8.000	0.000	−2.000	0.000
0.000	−15.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−7.000	0.000	−3.000	0.000
0.000	−13.000	−12.000	−11.000	−10.000	−9.000	−8.000	−7.000	−6.000	−5.000	−4.000	−4.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

We can see that the learnt q value is really close to optimal q_*. There is few exceptions which are the cells where the the agents never went (in the area at the left to terminal state, colored in blue in the q_*).

3 Second Case : linearly decrease ϵ from $1.$ to 0.1

In this second case, ϵ is linearly decreased over episodes. So, the agent take less random actions than previous one. In this case, it is less important that max_steps be large. The most important parameter to increase is is the numberof episode $n\_episodes$. If this parameter is too low, the agent won't do that much exploration and the policy may never converge to the optimal one. The results is that even by greatly increasing max_steps we will be stuck by only exploiting a sub-optimal policy.

A good choice is thus:

- $max\_steps \approx 500$
- $n\_episodes \approx 400$

Maze informations:

- length of shortest path: 39
- starting point: (3, 9) → green cell on fig I.3
- goal: (2,5) → red cell on fig I.3

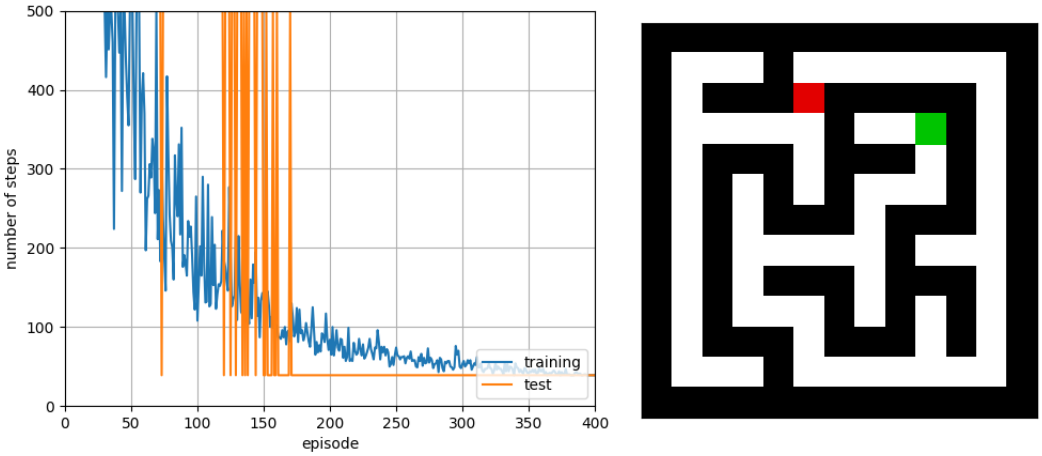


Figure I.3 – Number of steps per episode with $\epsilon - greedy$ policy and $greedy$ policy based on current q-value

We can see that as soon as the 175^{th} episode the greedy policy is optimal for this maze. We can thus conclude that choosing $n\_episodes \geq 175$ is enough to infer the optimal greedy policy but taking $n\_episodes \approx 400$ allows to have a very good estimation of the optimal action value function as shown below.

3.1 Result Q-table

Q-value for action 'up'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	−2.000	−3.000	−4.000	−5.000	−6.000	−7.000	−8.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−8.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	−40.128	−40.321	−39.993	0.000	−9.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−39.994	0.000	−10.000	0.000
0.000	0.000	0.000	−30.970	0.000	0.000	0.000	−35.999	−36.999	−38.997	0.000	−11.000	0.000
0.000	0.000	0.000	−31.702	0.000	0.000	0.000	−35.995	0.000	0.000	0.000	−12.000	0.000
0.000	0.000	0.000	−31.000	−31.000	−31.999	−33.000	−34.998	0.000	−15.677	−14.952	−13.000	0.000
0.000	0.000	0.000	−30.000	0.000	0.000	0.000	−33.998	0.000	0.000	0.000	−14.000	0.000
0.000	0.000	0.000	−29.000	−27.000	−26.000	0.000	−34.768	0.000	−21.247	0.000	−15.000	0.000
0.000	0.000	0.000	0.000	0.000	−26.000	0.000	0.000	0.000	−21.858	0.000	−16.000	0.000
0.000	0.000	0.000	0.000	0.000	−25.000	−23.000	−22.000	−21.000	−21.000	−19.000	−17.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Q-value for action 'down'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	−1.000	−3.000	−4.000	−5.000	−6.000	−7.000	−9.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−10.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	−40.128	−40.112	−39.000	0.000	−11.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−38.000	0.000	−12.000	0.000
0.000	0.000	0.000	−30.938	0.000	0.000	0.000	−35.000	−36.992	−37.992	0.000	−13.000	0.000
0.000	0.000	0.000	−30.000	0.000	0.000	0.000	−34.000	0.000	0.000	0.000	−14.000	0.000
0.000	0.000	0.000	−29.000	−30.999	−31.999	−32.999	−34.989	0.000	−15.702	−14.989	−15.000	0.000
0.000	0.000	0.000	−28.000	0.000	0.000	0.000	−35.297	0.000	0.000	0.000	−16.000	0.000
0.000	0.000	0.000	−28.000	−27.000	−25.000	0.000	−35.122	0.000	−20.970	0.000	−17.000	0.000
0.000	0.000	0.000	0.000	0.000	−24.000	0.000	0.000	0.000	−20.000	0.000	−18.000	0.000
0.000	0.000	0.000	0.000	0.000	−24.000	−23.000	−22.000	−21.000	−20.000	−19.000	−18.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Q-value for action 'left'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	−2.000	−2.000	−3.000	−4.000	−5.000	−6.000	−7.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−9.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	−40.126	−40.109	−40.919	0.000	−10.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−38.998	0.000	−11.000	0.000
0.000	0.000	0.000	−30.997	0.000	0.000	0.000	−35.999	−36.000	−37.000	0.000	−12.000	0.000
0.000	0.000	0.000	−30.916	0.000	0.000	0.000	−34.999	0.000	0.000	0.000	−13.000	0.000
0.000	0.000	0.000	−30.000	−30.000	−31.000	−32.000	−33.000	0.000	−15.479	−15.965	−15.000	0.000
0.000	0.000	0.000	−29.000	0.000	0.000	0.000	−34.788	0.000	0.000	0.000	−15.000	0.000
0.000	0.000	0.000	−28.000	−28.000	−27.000	0.000	−34.847	0.000	−21.404	0.000	−16.000	0.000
0.000	0.000	0.000	0.000	0.000	−25.000	0.000	0.000	0.000	−20.932	0.000	−17.000	0.000
0.000	0.000	0.000	0.000	0.000	−24.000	−24.000	−23.000	−22.000	−21.000	−20.000	−19.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Q-value for action 'right'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	−3.000	−4.000	−5.000	−6.000	−7.000	−8.000	−8.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−9.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	−40.209	−39.985	−39.992	0.000	−10.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−38.997	0.000	−11.000	0.000
0.000	0.000	0.000	−31.220	0.000	0.000	0.000	−36.995	−37.997	−37.987	0.000	−12.000	0.000
0.000	0.000	0.000	−30.935	0.000	0.000	0.000	−34.996	0.000	0.000	0.000	−13.000	0.000
0.000	0.000	0.000	−31.000	−32.000	−32.998	−33.998	−33.997	0.000	−14.995	−14.000	−14.000	0.000
0.000	0.000	0.000	−29.000	0.000	0.000	0.000	−34.321	0.000	0.000	0.000	−15.000	0.000
0.000	0.000	0.000	−27.000	−26.000	−26.000	0.000	−34.711	0.000	−21.551	0.000	−16.000	0.000
0.000	0.000	0.000	0.000	0.000	−25.000	0.000	0.000	0.000	−20.910	0.000	−17.000	0.000
0.000	0.000	0.000	0.000	0.000	−23.000	−22.000	−21.000	−20.000	−19.000	−18.000	−18.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

3.2 Optimal Q-table

Optimal Q-value for action 'up'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−8.000	−9.000	−10.000	0.000	−2.000	−3.000	−4.000	−5.000	−6.000	−7.000	−8.000	0.000
0.000	−8.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−8.000	0.000
0.000	−7.000	−5.000	−4.000	−3.000	−1.000	0.000	−42.000	−41.000	−40.000	0.000	−9.000	0.000
0.000	−6.000	0.000	0.000	0.000	−2.000	0.000	0.000	0.000	−40.000	0.000	−10.000	0.000
0.000	−7.000	0.000	−32.000	0.000	−3.000	0.000	−36.000	−37.000	−39.000	0.000	−11.000	0.000
0.000	−8.000	0.000	−32.000	0.000	0.000	0.000	−36.000	0.000	0.000	0.000	−12.000	0.000
0.000	−9.000	0.000	−31.000	−31.000	−32.000	−33.000	−35.000	0.000	−16.000	−15.000	−13.000	0.000
0.000	−10.000	0.000	−30.000	0.000	0.000	0.000	−34.000	0.000	0.000	0.000	−14.000	0.000
0.000	−11.000	0.000	−29.000	−27.000	−26.000	0.000	−35.000	0.000	−22.000	0.000	−15.000	0.000
0.000	−12.000	0.000	0.000	0.000	−26.000	0.000	0.000	0.000	−22.000	0.000	−16.000	0.000
0.000	−13.000	−15.000	−16.000	0.000	−25.000	−23.000	−22.000	−21.000	−21.000	−19.000	−17.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'down'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−7.000	−9.000	−10.000	0.000	−1.000	−3.000	−4.000	−5.000	−6.000	−7.000	−9.000	0.000
0.000	−6.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−10.000	0.000
0.000	−7.000	−5.000	−4.000	−3.000	−3.000	0.000	−42.000	−41.000	−39.000	0.000	−11.000	0.000
0.000	−8.000	0.000	0.000	0.000	−4.000	0.000	0.000	0.000	−38.000	0.000	−12.000	0.000
0.000	−9.000	0.000	−31.000	0.000	−4.000	0.000	−35.000	−37.000	−38.000	0.000	−13.000	0.000
0.000	−10.000	0.000	−30.000	0.000	0.000	0.000	−34.000	0.000	0.000	0.000	−14.000	0.000
0.000	−11.000	0.000	−29.000	−31.000	−32.000	−33.000	−35.000	0.000	−16.000	−15.000	−15.000	0.000
0.000	−12.000	0.000	−28.000	0.000	0.000	0.000	−36.000	0.000	0.000	0.000	−16.000	0.000
0.000	−13.000	0.000	−28.000	−27.000	−25.000	0.000	−36.000	0.000	−21.000	0.000	−17.000	0.000
0.000	−14.000	0.000	0.000	0.000	−24.000	0.000	0.000	0.000	−20.000	0.000	−18.000	0.000
0.000	−14.000	−15.000	−16.000	0.000	−24.000	−23.000	−22.000	−21.000	−20.000	−19.000	−18.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'left'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−8.000	−8.000	−9.000	0.000	−2.000	−2.000	−3.000	−4.000	−5.000	−6.000	−7.000	0.000
0.000	−7.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−9.000	0.000
0.000	−6.000	−6.000	−5.000	−4.000	−3.000	0.000	−42.000	−42.000	−41.000	0.000	−10.000	0.000
0.000	−7.000	0.000	0.000	0.000	−3.000	0.000	0.000	0.000	−39.000	0.000	−11.000	0.000
0.000	−8.000	0.000	−32.000	0.000	−4.000	0.000	−36.000	−36.000	−37.000	0.000	−12.000	0.000
0.000	−9.000	0.000	−31.000	0.000	0.000	0.000	−35.000	0.000	0.000	0.000	−13.000	0.000
0.000	−10.000	0.000	−30.000	−30.000	−31.000	−32.000	−33.000	0.000	−16.000	−16.000	−15.000	0.000
0.000	−11.000	0.000	−29.000	0.000	0.000	0.000	−35.000	0.000	0.000	0.000	−15.000	0.000
0.000	−12.000	0.000	−28.000	−28.000	−27.000	0.000	−36.000	0.000	−22.000	0.000	−16.000	0.000
0.000	−13.000	0.000	0.000	0.000	−25.000	0.000	0.000	0.000	−21.000	0.000	−17.000	0.000
0.000	−14.000	−14.000	−15.000	0.000	−24.000	−24.000	−23.000	−22.000	−21.000	−20.000	−19.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'right'

0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−9.000	−10.000	−10.000	0.000	−3.000	−4.000	−5.000	−6.000	−7.000	−8.000	−8.000	0.000
0.000	−7.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	−9.000	0.000
0.000	−5.000	−4.000	−3.000	−2.000	−2.000	0.000	−41.000	−40.000	−40.000	0.000	−10.000	0.000
0.000	−7.000	0.000	0.000	0.000	−3.000	0.000	0.000	0.000	−39.000	0.000	−11.000	0.000
0.000	−8.000	0.000	−32.000	0.000	−4.000	0.000	−37.000	−38.000	−38.000	0.000	−12.000	0.000
0.000	−9.000	0.000	−31.000	0.000	0.000	0.000	−35.000	0.000	0.000	0.000	−13.000	0.000
0.000	−10.000	0.000	−31.000	−32.000	−33.000	−34.000	−34.000	0.000	−15.000	−14.000	−14.000	0.000
0.000	−11.000	0.000	−29.000	0.000	0.000	0.000	−35.000	0.000	0.000	0.000	−15.000	0.000
0.000	−12.000	0.000	−27.000	−26.000	−26.000	0.000	−36.000	0.000	−22.000	0.000	−16.000	0.000
0.000	−13.000	0.000	0.000	0.000	−25.000	0.000	0.000	0.000	−21.000	0.000	−17.000	0.000
0.000	−15.000	−16.000	−16.000	0.000	−23.000	−22.000	−21.000	−20.000	−19.000	−18.000	−18.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

We can see that the learnt q value is really close to optimal q_*. Same thing as the first case append for states which were never visited (colored in blue in the q_*). There is one state (in green in q_* table) whose value is not really accurate. This is probably due to the fact that the agent stopped exploring early.

Part II

Task 2: DQN for learning any maze

1 Training method and test performances

In this task, we are using DQN algorithm to learn to solve any maze of dimensions 7×9 with a single neural network. I mainly used the original code for this task because I found that DQN algorithm with good parameters and some few tricks can results to a good model. What I mean by "good model" is a model performing in **more than 90%** of the case the shortest path during testing time.

In order to improve performances I trained the model several times using a small trick. The trick consists in increasing progressively the minimal length of the shortest path produced while generating the maze. The key idea behind this trick is that, at the beginning of training, short pathes allow to speed up training but after the agent is sufficiently trained to avoid making bad actions when the terminal state is close, it makes sense to increase progressively the length of pathes in order that the agent improve his skills.

The first time training was made with the parameters mentionned below and the following times using the same parameters except for ϵ which is set to be constant (typically $\epsilon = 0.1$).

1.1 Model and parameters for the 1st training period

I show here the parts of the code that I changed from the original file *maze_DQN.py*.

I changed the last fully connected layer to allows more linear combinations of the outputs of the convolutional layers. Given that my student ID is 20196425, the output of the last hidden layer is chosen to be 355.

```
env = maze_environment(7, 9, 0, mode = 'nn')
last_digit_of_your_student_id = 5
c1 = 30
c2 = 30
f1 = (350 // 10) * 10 + last_digit_of_your_student_id
```

I didn't add layers to the model, I kept 2 convolutional layers followed by a fully connected layer. However, I changed the dimensions of the number of filters for the first convolutional layer (4×4 instead of 3×3). I also change the learning rate from 0.001 to 0.00025 which proved to achieve better performances.

```
sess = tf.InteractiveSession()
x_input = tf.placeholder(tf.float32, shape=[None, env.ny, env.nx, env.nf])
...
# First convolutional layer
W_conv1 = tf.Variable(tf.truncated_normal([4, 4, env.nf, c1], stddev=0.1))
...
# Second convolutional layer
W_conv2 = tf.Variable(tf.truncated_normal([3, 3, c1, c2], stddev=0.1))
...
# Fully-connected Layer
W_fc1 = tf.Variable(tf.truncated_normal([env.ny * env.nx * c2, f1], stddev=0.1))
...
# Cost function and optimizer
mse = tf.reduce_mean(tf.square(tf.reduce_sum(
```

```
    y_pred * a_selected, 1, keepdims=True)- y_true))
train = tf.train.AdamOptimizer(0.00025).minimize(mse)
# First convolutional layer for target network
W_conv1_t = tf.Variable(tf.constant(0., shape=[4, 4, env.nf, c1]))
...
# Second convolutional layer for target network
W_conv2_t = tf.Variable(tf.constant(0., shape=[3, 3, c1, c2]))
...
```

The other parameters used are as follow for the first training period.

```
replay_memory_size = 1000
n_episodes = 15000
init_epsilon = 1.0
final_epsilon = 0.1
final_exploration_episode = 11000
target_update_frequency = 100
max_steps = 60
gamma = 1.
minibatch_size = 32
```

Each time, in addition to the final model I saved the model with the highest test ratio.

```
best_score = 0
...
...
n_test_period = 100
if episode % n_test_period == n_test_period - 1:   # for testing
    test_score, test_extra_steps = run_tests(env, 100, max_steps)
    print('episode: %4d, frames: %6d, train score: %.1f, mean steps ...')
    score = np.sum(test_extra_steps==0) / 100
    # Save best model
    if score > best_score:
      best_score = score
      print("Saving new best model ", score)
      saver.save(sess, './best_rereremaze.ckpt')
...
```

1.2 Testing performance

I tested my models using the following lines.

```
# Restore model
name_model = "./best_maze.ckpt"
saver = tf.train.Saver({W_conv1, b_conv1, W_conv2, b_conv2,
                        W_fc1, b_fc1, W_fc2, b_fc2}, max_to_keep = 0)
saver.restore(sess, name_model)
env = maze_environment(7, 9, 0, mode = 'nn')
n_test_runs = 100
test_score, test_extra_steps = run_tests(env, n_test_runs, max_steps)
print('Average over {} test episodes'.format(n_test_runs))
print('Final test score: %.1f' % test_score)
print( 'Final test success ratio:
    %.2f' % (np.sum(test_extra_steps==0) / n_test_runs))
print('Final extra steps: %.2f' % (np.mean(test_extra_steps)))
```

Results for this first period was not bad but far from perfect. I got scores in the following gap $[0.74, 0.79]$.

```
Final test score: -16.3
Final test success ratio: 0.79
```

1.3 Next training periods

I then retrain the previously the trained model three times with each times the following parameters and by progressively increasing the minimal length of pathes in the maze.

```
n_episodes = 15000
init_epsilon = 0.1
final_epsilon = 0.1
replay_memory_size = 1000
target_update_frequency = 100
max_steps = 60
gamma = 1.
minibatch_size = 32
```

2nd training period $\rightarrow$ shortest path ≥ 6

```
env = maze_environment(7, 9, 6, mode = 'nn')
```

Test performance gap : $[0.79, 0.88]$.

```
Average over 100 test episodes
Final test score: -11.5
Final test success ratio: 0.90
Final extra steps: 5.74
```

3rd training period $\rightarrow$ shortest path ≥ 10

```
env = maze_environment(7, 9, 10, mode = 'nn')
```

Test performance gap : $[0.85, 0.95]$.

```
Average over 100 test episodes
Final test score: -8.8
Final test success ratio: 0.95
Final extra steps: 4.42
```

4th training period $\rightarrow$ shortest path ≥ 15

```
env = maze_environment(7, 9, 15, mode = 'nn')
```

Test performance gap : $[0.80, 0.91]$.

```
Average over 100 test episodes
Final test score: -10.5
Final test success ratio: 0.91
Final extra steps: 5.26
```

Conclusion of tests

We get better tests performances after the 3rd training period (see more tests performance in Appendix). We can interpret this by saying that the 4th training period may have cause to reduce the performances when the target is close to the agent.

1.4 Comparison to optimal Q values

To analyse the trained network, we can compare the estimated Q values by our neural network with the analytically calculated Q values. The real Q values are computed using the same method as in the first task (see 1). As for the estimated Q values by our trained neural network, I used the following code.

```
# Calculate estimated Q values by our trained NN for a given maze
def estimated_q(env):
    q = np.zeros((env.ny, env.nx, 4))
    loc = env.init_state
    for i in range(env.ny):
        for j in range(env.nx):
            new_loc = (i,j)
            s[loc[0], loc[1], 1] = 0
            s[new_loc[0], new_loc[1], 1] = 1
            if not env.maze[i,j]:
              q[i, j, :] = sess.run(y_pred, {x_input: np.reshape(np.copy(s),
                                                [1, env.ny, env.nx, env.nf])})
            loc = new_loc
    return q
```

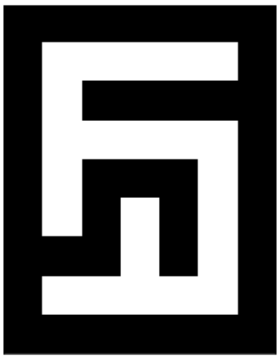


Figure II.1 – Maze used to compare q values

- Target position (7,4)

Estimated Q-value for action 'up'

0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−12.804	−11.763	−12.566	−4.739	−4.805	0.000
0.000	−11.450	0.000	0.000	0.000	0.000	0.000
0.000	−10.282	−5.116	−4.058	−2.500	−2.849	0.000
0.000	−9.131	0.000	0.000	0.000	−1.287	0.000
0.000	−9.372	0.000	−6.230	0.000	−1.668	0.000
0.000	0.000	0.000	−6.620	0.000	−2.513	0.000
0.000	−8.403	−7.380	−4.903	−4.087	−2.329	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'up'

0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−12.000	−13.000	−14.000	−15.000	−16.000	0.000
0.000	−12.000	0.000	0.000	0.000	0.000	0.000
0.000	−11.000	−9.000	−8.000	−7.000	−6.000	0.000
0.000	−10.000	0.000	0.000	0.000	−6.000	0.000
0.000	−11.000	0.000	−4.000	0.000	−5.000	0.000
0.000	0.000	0.000	−4.000	0.000	−4.000	0.000
0.000	−4.000	−3.000	−3.000	0.000	−3.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000

Estimated Q-value for action 'down'

0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−10.480	−11.134	−12.034	−4.371	−1.924	0.000
0.000	−9.511	0.000	0.000	0.000	0.000	0.000
0.000	−9.107	−5.649	−3.990	−2.348	−1.203	0.000
0.000	−8.683	0.000	0.000	0.000	−2.892	0.000
0.000	−8.367	0.000	−6.711	0.000	−2.291	0.000
0.000	0.000	0.000	−6.271	0.000	−2.376	0.000
0.000	−8.284	−7.164	−5.082	−3.669	−2.912	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'down'

0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−11.000	−13.000	−14.000	−15.000	−16.000	0.000
0.000	−10.000	0.000	0.000	0.000	0.000	0.000
0.000	−11.000	−9.000	−8.000	−7.000	−5.000	0.000
0.000	−12.000	0.000	0.000	0.000	−4.000	0.000
0.000	−12.000	0.000	−3.000	0.000	−3.000	0.000
0.000	0.000	0.000	−2.000	0.000	−2.000	0.000
0.000	−4.000	−3.000	−2.000	0.000	−2.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000

Estimated Q-value for action 'left'

0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−11.639	−10.325	−11.759	−6.424	−4.231	0.000
0.000	−11.123	0.000	0.000	0.000	0.000	0.000
0.000	−9.914	−5.720	−4.384	−2.697	−2.189	0.000
0.000	−9.208	0.000	0.000	0.000	−2.238	0.000
0.000	−9.189	0.000	−6.255	0.000	−2.431	0.000
0.000	0.000	0.000	−6.427	0.000	−2.717	0.000
0.000	−8.701	−7.476	−4.817	−4.530	−3.785	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'left'

0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−12.000	−12.000	−13.000	−14.000	−15.000	0.000
0.000	−11.000	0.000	0.000	0.000	0.000	0.000
0.000	−10.000	−10.000	−9.000	−8.000	−7.000	0.000
0.000	−11.000	0.000	0.000	0.000	−5.000	0.000
0.000	−12.000	0.000	−4.000	0.000	−4.000	0.000
0.000	0.000	0.000	−3.000	0.000	−3.000	0.000
0.000	−4.000	−4.000	−3.000	0.000	−1.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000

Estimated Q-value for action 'right'

0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−12.134	−12.114	−12.725	−3.778	−3.935	0.000
0.000	−11.316	0.000	0.000	0.000	0.000	0.000
0.000	−10.517	−4.733	−3.389	−1.750	−1.861	0.000
0.000	−9.340	0.000	0.000	0.000	−1.848	0.000
0.000	−9.226	0.000	−6.158	0.000	−2.230	0.000
0.000	0.000	0.000	−5.928	0.000	−2.871	0.000
0.000	−7.592	−6.086	−4.237	−3.074	−2.745	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000

Optimal Q-value for action 'right'

0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	−13.000	−14.000	−15.000	−16.000	−16.000	0.000
0.000	−11.000	0.000	0.000	0.000	0.000	0.000
0.000	−9.000	−8.000	−7.000	−6.000	−6.000	0.000
0.000	−11.000	0.000	0.000	0.000	−5.000	0.000
0.000	−12.000	0.000	−4.000	0.000	−4.000	0.000
0.000	0.000	0.000	−3.000	0.000	−3.000	0.000
0.000	−3.000	−2.000	−1.000	0.000	−2.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000

Although that most of the values are different from more than 1 with the optimal q value, the agent usually have an optimal policy when the path between him and the target is not too long.

Part III

Task 3: Mini AlphaGo Zero

In this last task, the goal is to train a policy network to predict the next position to play using the current state of the game. We first intended to reuse exactly the same trained method as in AlphaGO Zero. We finally didn't do this because of the deadline and that it requires to reimplements more function than expected in the environment. Nevertheless, the method that we used share several features with the real AlphaGO Zero method. We wil describe our method more in details in the first section and explain parametrisation and results in the second section.

1 Method

1.1 Neural network architecture

We mainly used the same neural network architecture than in AlphaGO Zero. That is to say way build a model where policy network and value network share a common part composed of N residual blocks. Because we needed to build an AI able to play to a game of Go of dimensions 6×6, it was obvious that we didn't need 39 residual blocks with 256 filters for each convolutional layer. Instead of this we only used $N = 1$ residual block (composed of two 3×3-convolutional layers with 50 filters) for the common part. For the part specific to the policy, we followed the paper which advised to use one 1×1 convolutional layer and then reshape the output in a tensor of order 1. In the part specific to the value network, we also use one 1×1 convolutional layer followed by two fully connected layers, the first with 155 outputs and the second with only 3 outputs (in order to predict the probability of winning, loosing and tie).

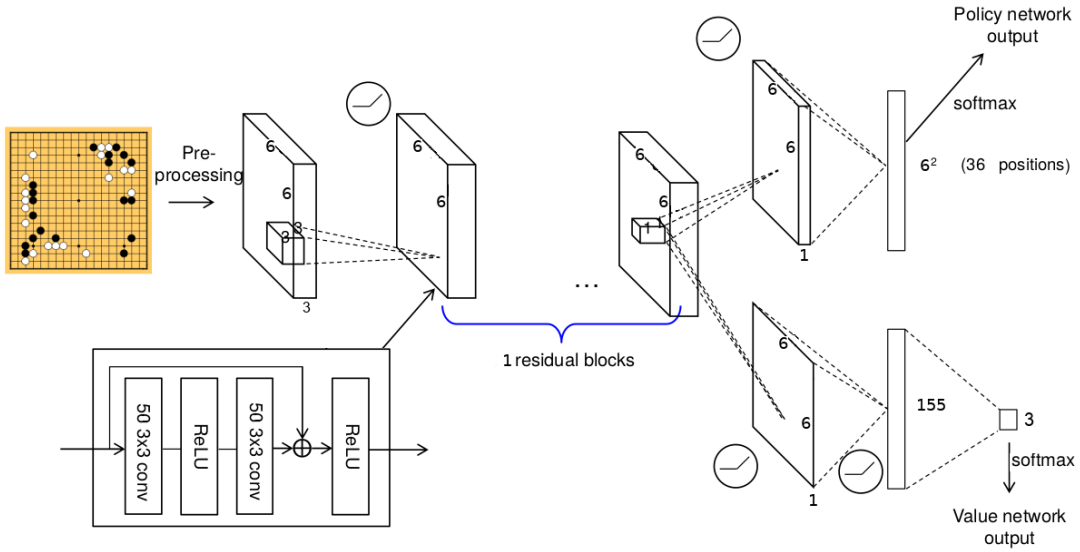


Figure III.1 – Neural network architecture

1.2 Loss function

One thing that we modified from the original code is the way of generating data. In fact, we now need a target to the policy network p and so we cannot only use the result of each game z for the training process as done in $go\_train.py$. We used this z to compute the loss related to the value network outputs but we needed other data to compute the loss related to the policy outputs. The method used in AlphaGO Zero requires, during self-play step, to store the outputs of the policy network. To this end we needed to fundamentally change the original environment. Because of the time, we decided to not make this changes and use the probability of winning estimated by the value network v to compute the loss of the policy. The target for the policy π is the one hot vector where the one represent the position to play in order to have the maximal probability of winning. This method is less computationnally efficient because it requires to evaluate the output for every possible next state.

$$\mathcal{L} = -z^T log(v) - \pi^T log(p) + c\| \theta\|^2 \tag{III.1}$$

2 Test results

2.1 Training procedure

For training the model explainde above we used Adam optimizer with a learning rate of 0.001. We repeated self-play and neural network training for three generations. For each generation we trained the neural network during 20 epochs over all the data generated during self-play period. For each generation, the number of game played was as follow :

- First generation : 4000 games
- Second generation : 4000 games
- Third generation : 5000 games

2.2 Results

We tested our trained model against a random policy for 50000 games. The code for testing is given in Appendix 2.2.

We didn't achieve 100% of winning games but $99,83\%$ when playing black stones and $99,646\%$ when playing white stones. It is difficult to analyse this result since it is only game against random player and so it does not mean that it can defeat good human players. However, I can affirm that it is a better player than me but I am not a good adversary for this game.

```
### TEST MODEL AGAINST RANDOM PLAYER ###
INFO:tensorflow:Restoring parameters from ./project2_task3_20196425.ckpt
INFO:tensorflow:Restoring parameters from ./project2_task3_20196425.ckpt

Win when playing black: 0.9983  49915/50000
Win when playing white: 0.99646 49823/50000
```

Figure III.2 – Results against random player

Appendix

Appendix task 2

```
Average over 100 test episodes
Final test score: -11.5
Final test success ratio: 0.91
Final extra steps: 5.73
Average over 100 test episodes
Final test score: -10.7
Final test success ratio: 0.92
Final extra steps: 5.36
Average over 100 test episodes
Final test score: -8.8
Final test success ratio: 0.95
Final extra steps: 4.42
Average over 100 test episodes
Final test score: -14.5
Final test success ratio: 0.85
Final extra steps: 7.26
Average over 100 test episodes
Final test score: -10.3
Final test success ratio: 0.92
Final extra steps: 5.16
Average over 100 test episodes
Final test score: -14.3
Final test success ratio: 0.86
Final extra steps: 7.17
Average over 100 test episodes
Final test score: -10.9
Final test success ratio: 0.93
Final extra steps: 5.45
Average over 100 test episodes
Final test score: -11.5
Final test success ratio: 0.91
Final extra steps: 5.76
Average over 100 test episodes
Final test score: -13.8
Final test success ratio: 0.87
Final extra steps: 6.88
```

Figure III.3 – 10 tests performance for the best model

Appendix task 3

```
def test_game(player1={'net':None,'scope':"20196425",'ckpt':"filename.ckpt"},
    player2={'net': None, 'scope':"20196425", 'ckpt': "filename.ckpt"},
    n_time=1):
"""
Each player is describe by 3 items which allow to load the graph and restore it.
- player['net'] is the "network" function (if None then random player)
- player['scope'] is the scope to restore the network parameters
- player['filename'] is the path to the checkpoint file to restore
"""
    # Find and execute action for the output of a policy network
    def find_action_and_execute(output, p, board, state):
        sorted_output = np.argsort(-output)    # descending order
        for xy in sorted_output:
          bn, sn, vm = game.next_move_with_action(  b=board, state=state,
                                                    xy=game.xy(xy), p=p)
          if vm > 0:
              return bn, sn, vm[0]
        return board, state, 0

    # Find and execute a random action
    def find_random_action_and_execute(p, board, state):
        sorted_output = np.arange(36)    # descending order
        np.random.shuffle(sorted_output)
        for xy in sorted_output:
          bn, sn, vm = game.next_move_with_action(  b=board, state=state,
                                                    xy=game.xy(xy), p=p)
          if vm > 0:
              return bn, sn, vm[0]
        return board, state, 0


    sess = tf.Session()
    S = tf.placeholder(tf.float32, shape=[None, game.nx, game.ny, 3], name="S")

    ''' construct graph '''
    if player1['net'] is not None:
    with tf.variable_scope(player1['scope']):
      # Estimation for log probability
      _, Y1 = player1['net'](S)

    if player2['net'] is not None:
    with tf.variable_scope(player2['scope']):
      # Estimation for log probability
      _, Y2 = player2['net'](S)

    ''' Restore params '''
    if player1['net'] is not None:
        saver1 = tf.train.Saver(var_list=tf.get_collection(
            tf.GraphKeys.GLOBAL_VARIABLES, scope=player1['scope']))
        saver1.restore(sess, player1['ckpt'])

    if player2['net'] is not None:
        saver2 = tf.train.Saver(var_list=tf.get_collection(
            tf.GraphKeys.GLOBAL_VARIABLES, scope=player2['scope']))
        saver2.restore(sess, player2['ckpt'])
```

```
    # Start testing
    win_list = []
    for i in range(n_time):
        np0 = game.nx * game.ny * 2
        [board, state] = game.game_init(1)
        p = 1
        d = np.zeros((game.nx, game.ny, 3), dtype=np.float32)
        vm0 = 1
        for i in range(np0):
          # update d
          d[:, :, 0] = (board[:, :, 0] == p)
          d[:, :, 1] = (board[:, :, 0] == 3 - p)
          d[:, :, 2] = 2 - p

          if p == 1:
              if player1['net'] is not None:
                  policy = sess.run(Y1, feed_dict={S: [d]})
                  board, state, valid_move = find_action_and_execute(
                      policy[0], p=p, board=board, state=state)
              else:
                  board, state, valid_move = find_random_action_and_execute(
                      p=p, board=board, state=state)
          else:
              if player2['net'] is not None:
                  policy = sess.run(Y2, feed_dict={S: [d]})
                  board, state, valid_move = find_action_and_execute(
                      policy[0], p=p, board=board, state=state)
              else:
                  board, state, valid_move = find_random_action_and_execute(
                      p=p, board=board, state=state)

          if vm0 + valid_move == 0:
              break

          p = 3 - p
          vm0 = valid_move

        r, _, s1, s2 = game.winner(board, state)
        win_list.append(r[0])
        return win_list
```

```
### TEST AGAINST RANDOM POLICY
def test_random(player1={'net':[],'scope':'scope_name',
                         'ckpt': 'filename.ckpt'},
                n_test=50000):

    tf.reset_default_graph()
    win_list = test_game(player1=player1, n_time=n_test)
    win_list = np.array(win_list)
    n_black_win = np.sum(win_list==1)

    tf.reset_default_graph()
    win_list = test_game(player2=player1, n_time=n_test)
    win_list = np.array(win_list)
    n_white_win = np.sum(win_list==2)

    print("Win when playing black: {}\t{}/{}"
            .format(n_black_win/n_test, n_black_win, n_test))
    print("Win when playing white: {}\t{}/{}"
            .format(n_white_win/n_test, n_white_win, n_test))
```

```
### TEST WITH TWO TRAINED POLICIES
def test_two(player1={'net':[],'scope':'scope_name', 'ckpt': 'filename.ckpt'},
   player2={'net':[],'scope':'scope_name', 'ckpt': 'filename.ckpt'}):

    tf.reset_default_graph()
    win = test_game(player1=player1, player2=player2, n_time=1)
    print("Winner: ", win[0])
```