Report

March 7, 2020

1 Machine, Data and Learning

1.1 Assignment 2 Part 2

Vasu Singhal (2018101074)

Tanish Lad (2018114005)

```
[ ]: import numpy as np
     import os
```

The parameters for value iteration algorithm are initialised below.

Team Number is 32, hence Y = 2, and hence for task 1, Penalty = -5.

The value of gamma, delta, and step cost is changed accordingly for task 2.

```
[ ]: num_h = 5
     num_a = 4
     num_s = 3
     gamma = 0.99
     delta = 1e-3
     step_cost = {
         "SHOOT": -5,
         "DODGE": -5,
         "RECHARGE": -5
     }
     non_terminal_reward = 0
     terminal_reward = 10
     inf = 1e17

     actions = ["SHOOT", "DODGE", "RECHARGE"]
```

All the possible states in which the player can exist are defined below and stored as a list of tuples

```
[ ]: states = [(health, arrows, stamina) for health in range(num_h)
               for arrows in range(num_a)
               for stamina in range(num_s)]
```

```
[ ]: to_print = []
     all_tasks = []
```

all_state is a dictionary whose key is a ”state” and value is the probability. It will be used later to store the probability of reaching a particle state. Currently all the probabilities are 0.

```
[ ]: def get_states():
         all_state = {}
         for health, arrows, stamina in states:
             all_state[tuple([health, arrows, stamina])] = 0

         return all_state
```

The transition_prob dictionary stores the probability of transition from current state to the next state via a given action.

In other words, **transition_prob**[s][a][s’] stores **P(s’|s,a)**

```
[ ]: def get_transition_probabilities():
         transition_prob = {}

         for health, arrows, stamina in states:
             transition_prob[tuple([health, arrows, stamina])] = {
                 "SHOOT": get_states(), "DODGE": get_states(), "RECHARGE":␣
      ↪get_states()}

         for health, arrows, stamina in states:
             state = tuple([health, arrows, stamina])

             # RECHARGE
             next_state1 = tuple([health, arrows, stamina])
             next_state2 = tuple([health, arrows, min(num_s - 1, stamina + 1)])

             transition_prob[state]["RECHARGE"][next_state1] += 0.2
             transition_prob[state]["RECHARGE"][next_state2] += 0.8

             # DODGE
             if(stamina == 0):
                 continue

             next_state1 = tuple([health, arrows, max(stamina - 1, 0)])
             next_state2 = tuple([health, arrows, max(stamina - 2, 0)])
             next_state3 = tuple(
                 [health, min(arrows + 1, num_a - 1), max(stamina - 1, 0)])
             next_state4 = tuple(
                 [health, min(arrows + 1, num_a - 1), max(stamina - 2, 0)])

             transition_prob[state]["DODGE"][next_state1] += 0.16
```

```
        transition_prob[state]["DODGE"][next_state2] += 0.04
        transition_prob[state]["DODGE"][next_state3] += 0.64
        transition_prob[state]["DODGE"][next_state4] += 0.16

        # SHOOT
        if(arrows == 0):
            continue

        next_state1 = tuple(
            [max(health - 1, 0), max(arrows - 1, 0), max(stamina - 1, 0)])
        next_state2 = tuple([max(health, 0), max(
            arrows - 1, 0), max(stamina - 1, 0)])

        transition_prob[state]["SHOOT"][next_state1] += 0.5
        transition_prob[state]["SHOOT"][next_state2] += 0.5

    return transition_prob
```

[ ]:
```
transition_prob = get_transition_probabilities()
# transition_prob
```

The **get_utilities** function returns the array new_utilities which stores the utilities of the current iteration.

In other words, given **U_{t}**, the function get_utilities will calculate **U_{t+1}**

[ ]:
```
def get_utilities(utilities):
    new_utilities = np.zeros(shape=(num_h, num_a, num_s))

    for health, arrows, stamina in states:

        cur_state = tuple([health, arrows, stamina])

        if health == 0:
            continue

        cur_max = -inf

        for action in actions:

            if action == "SHOOT":
                if stamina == 0 or arrows == 0:
                    continue

            elif action == "DODGE" and stamina == 0:
                continue

            total_reward = 0
            cur = 0

            for h, a, s in states:

                new_state = tuple([h, a, s])

                if h == 0:
                    total_reward += (step_cost[action] + terminal_reward) * \
                        transition_prob[cur_state][action][new_state]

                else:
                    total_reward += (step_cost[action] + non_terminal_reward) *␣
↪\

                        transition_prob[cur_state][action][new_state]

                cur += gamma * \
                    transition_prob[cur_state][action][new_state] * \
                    utilities[h, a, s]

            cur += total_reward

            if cur_max < cur:
                cur_max = cur

        new_utilities[health, arrows, stamina] = cur_max

    return new_utilities
```

The function **get_action** calculates which action maximizes the utility at that iteration for every state.

[ ]:
```
def get_action(new_utilities):
    for health, arrows, stamina in states:

        cur_state = tuple([health, arrows, stamina])

        if health == 0:
            to_print.append(
                f"({health},{arrows},{stamina}):
↪{-1}=[{round(new_utilities[health, arrows, stamina], 3)}]")
            continue

        cur_max = -inf
        cur_action = ""

        for action in actions:
            if action == "SHOOT":
                if stamina == 0 or arrows == 0:
                    continue

            elif action == "DODGE" and stamina == 0:
                continue

            cur = 0
            total_reward = 0

            for h, a, s in states:
                new_state = tuple([h, a, s])

                if h == 0:
                    total_reward += (step_cost[action] + terminal_reward) * \
                        transition_prob[cur_state][action][new_state]
                else:
                    total_reward += (step_cost[action] + non_terminal_reward) *␣
↪\
                        transition_prob[cur_state][action][new_state]

                cur += gamma * \
                    transition_prob[cur_state][action][new_state] * \
                    new_utilities[h, a, s]
            cur += total_reward
            if cur_max < cur:
                cur_max = cur
                cur_action = action

        to_print.append(
            f"({health},{arrows},{stamina}):
↪{cur_action}=[{round(new_utilities[health, arrows, stamina], 3)}]")
```

The function **hasConverged** checks if our model has converged or not.

In other words, it calculates the max of each of the absolute values of U_{t+1} - U_{t} and checks if it is less than the given max possible epsillon (which in this case is delta = 1e-3)

[ ]:
```
def hasConverged(utilities, new_utilities):
    return np.max(np.abs(new_utilities - utilities)) < delta
```

[ ]:
```
def value_iteration():

    utilities = np.zeros(shape=(num_h, num_a, num_s))

    iterations = 0

    while True:
        to_print.append(f"iteration={iterations}")

        new_utilities = get_utilities(utilities)

        get_action(new_utilities)

        converged = hasConverged(utilities, new_utilities)

        if converged:
            break

        utilities = new_utilities
        iterations += 1

        to_print.append("\n")
```

value_iteration is called below for each task and subparts. The values of parameters are changed according to the task and subparts

[ ]:
```
for i in range(4):
    if i == 1:
        step_cost["SHOOT"] = -0.25
        step_cost["DODGE"] = -2.5
        step_cost["RECHARGE"] = -2.5

    elif i == 2:
        step_cost["SHOOT"] = -2.5
        gamma = 0.1

    elif i == 3:
        delta = 1e-10

    value_iteration()

    all_tasks.append(to_print)
    to_print = []
```

[ ]:
```
# for i in all_tasks:
#     for j in i:
#         print(j)
```

2 Inferences and Observations

2.1 Task 1

The policy that model follows is fairly obvious. Since there is positive reward for the state where health of Mighty dragon is 0, Lero has a tendency to try to achieve such states

Lero has to recharge when his stamina is 0 because that is his only option.

When Lero has no arrows but has stamina, Lero chooses to dodge so as to gain arrows

When Lero has both stamina and arrows, he can either shoot, dodge, or even recharge. If the Mighty Dragon's health is low, Lero shoots. Lero dodges when he feels that he will require more arrows in the future to kill the Dragon. Lero recharges when he feels that he will require more stamina in the future to kill the Mighty Dragon.

2.2 Task 2

2.2.1 Part 1

Here, the step cost of "SHOOT" action is the least negative out of all the three actions, and shooting brings the player closer to terminal states (with high positive reward), and hence there is more incentive for the agent to "SHOOT" whenever possible. This change can be observed from the differences in policies obtained in Task 1 and Task 2 Part 1. For example, in the state tuple of (3, 3, 1) and (3, 2, 1) [here, (h, a, s) represents the tuple of health, arrows, stamina], the agent changes its policy from "RECHARGE" to "SHOOT". Similarly for the state tuple of (4, 2, 1), the agent changes its policy from "DODGE" to "SHOOT". The model tries to be greedy and "SHOOTS" whenever possible.

2.2.2 Part 2

In part 2, the gamma is 0.1, which is a lot lesser than what was in previous tasks (0.99). Smaller gamma means smaller horizon, so shorter term focus, so less preference to future. So whenever he has arrows, he shoots without thinking of the future. This change can be observed from the differences in policies obtained in Task 1 and Task 2 Part 2. For example, in the state tuple of (3, 3, 1) and (3, 2, 1) [here, (h, a, s) represents the tuple of health, arrows, stamina], the agent changes its policy from "RECHARGE" to "SHOOT". Similarly for the state (4, 2, 1), the agent changes its policy from "DODGE" to "SHOOT". The agent doesn't think whether he will have enough stamina or enough arrows in future to kill the dragon, he just tries to get as much closer to any one of the good terminal states as possible. Also, the number of iterations it takes to converge gets very low due to very less gamma.

2.2.3 Part 3

Little difference can be observed in Part 3 compared to the outputs of Part 2, except for the fact that the number of iterations required to converge have increased due to a lot lesser delta.