Introduction to Generative Models of a Demonstrator's Actions

Practical Work Session #3

Michele Ferrari **06/12/2021**

Table of Contents

1. MDP Development: Gridworld Environment

The class *Grid* is defined below. It implements a *gridworld* environment, upon which a *Markov Decision Process* (MDP) can be formulated.
The 8 different scenarios presented in this work are shown below: each one consists of a 5×6 grid, where a *goal state* is marked in yellow and *dangerous states* are marked in red. Their respective rewards are shown in the legend, along with the indication of the value of *safe* (white) tiles.

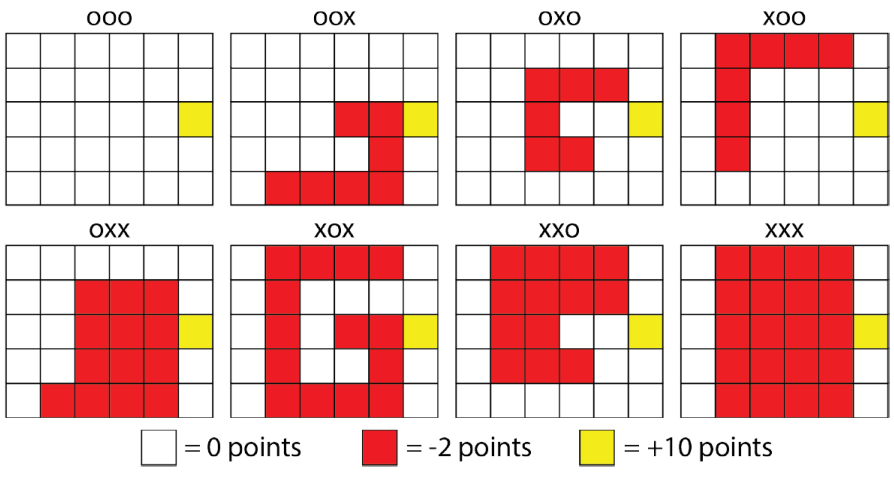

FIGURE 1 – Gridworld environments defined in [1]

As a reminder, an MDP can be modeled as a tuple $< S, P, R, \gamma >$, where:

- S is a set of states,
- P is the output of a transition function T that maps states and actions to distributions over next states: $T : S \times A \rightarrow P(S)$, given A, a set of actions,
- R is a reward function that maps state/action/next-state transitions to scalar rewards,
- γ is a discount factor $(\gamma \in [0, 1))$ that captures a preference for earlier rewards.

```
AGENT_N = 1     # Agent key in dict
GOAL_N = 2      # Goal key in dict
DANGER_N = 3    # Danger key in dict
SAFE_N = 4      # Safe key in dict
START_N = 5     # Start key in dict

class Grid:

    ## Initialization of the grid environment and the q_learning objects and parameters ##
    def __init__(self, s, val_end, val_danger, val_safe, start_pos, end_pos, temp, disc=0.99, learn_rate=0.1,
grid_name="ooo"):
        self.size = s
        self.value_end = val_end
        self.value_danger = val_danger
        self.value_safe = val_safe
        self.tab = self.value_safe*np.ones((self.size[0],self.size[1]))
        self.start = start_pos
        self.end = end_pos
        self.danger = []
        self.state = self.start
        self.q_table = np.zeros((4,self.size[0]*self.size[1]))
        self.discount = disc
        self.lr = learn_rate
        self.grid_name = grid_name

        self.temperature = temp

        # Visualization
        self.colors = {1: (255, 175, 0),  # blueish color: agent
                       2: (0, 255, 0),    # green color: goal
                       3: (0, 0, 255),    # red: danger
                       4: (0, 0, 0),      # white: safe
                       5: (0, 255, 255)}  # yellow: start

        fourcc = cv2.VideoWriter_fourcc(*'mpeg')
        fwidth = self.size[1]*100
        fheight = self.size[0]*100
        self.out = cv2.VideoWriter('videos\simulation_learning_agent_' \
                                   +grid_name+'_'+str(self.temperature)+'.mp4', \
                                   fourcc, 10.0, (fwidth,fheight))

    ## Creation of the maze ##
    def add_end(self,s):
        if not(s == self.start):
            self.tab[s[0],s[1]] = self.value_end
            self.end = s

        else:
            print("This position corresponds to the starting point!")


    def add_safe(self,s):
        if not(s == self.start):
            self.tab[s[0],s[1]] = self.value_safe
```

```
        else:
            print("This position corresponds to the starting point!")

    def add_danger(self,s):
        if not(s == self.start):
            self.tab[s[0],s[1]] = self.value_danger
            self.danger.append(s)
        else:
            print("This position corresponds to the starting point!")


    ## State modification and display ##
    def step(self,o):
        new_s = [self.state[0],self.state[1]]

        if self.start == self.end:
            return self.end, self.value_end, True

        if o == 0 and new_s[0]-1 >= 0: #North
            new_s[0] -= 1
        elif o == 1 and new_s[1]-1 >= 0: #West
            new_s[1] -= 1
        elif o == 2 and new_s[0]+1 < self.size[0]: #South
            new_s[0] += 1
        elif o == 3 and new_s[1]+1 < self.size[1]: #East
            new_s[1] += 1

        if new_s == self.state:
            return self.state, 0, False
        else:
            return new_s, self.tab[new_s[0],new_s[1]], new_s == self.end


    # Random reset of the environment, e.g, for the q-learning
    def rand_reset(self):
        self.start = sub2ind(self.size[1],np.random.randint(0,self.size[0]*self.size[1]))
        self.state = [self.start[0],self.start[1]]


    # Reset at a specific position, e.g, for comparaison of noisy-rational demonstration
    def reset(self,start,goal):
        self.start = start
        self.end = goal
        self.state = start
```

2. Boltzmann Model Implementation

The Boltzmann Model allows to robustly simulate human decision processes, where optimality is not attained due to the non-ideality of the choosing agent. This approach quantifies the probability that an agent selects a given option according to the formula:

$$P(o) = \frac{e^{R(o)}}{\sum_{i \in O} e^{R(i)}}$$

EQUATION 1: Boltzmann Probability Distribution

which can also be computed as

$$\pi(a_t|s_t) = \frac{e^{Q^*(s_t,a_t)/\tau}}{\sum_{a' \in A(s_t)} e^{Q^*(s_t,a')/\tau}}$$

EQUATION 2: Boltzmann Probability Distribution with the temperature parameter

once an *optimal Q-function value* $(Q^*(s_t, a_t))$ has been determined for each state by a RL algorithm (such as *Q-learning*). τ is a *temperature* parameter, that allows to control the generation of this probability distribution, as it will be shown in the following.

The goal of this approach is perturbing the ideal decisional process that can be carried out by an optimal agent, thanks to the injection of some random noise: a sub-optimal policy is obtained and this can be exploited as a simulated human demonstration for interactive learning. This last step, however, is not covered in this work.

The function `softmax_distribution` allows to get a probability distribution over the 4 possible actions according to the Boltzmann Model. This is useful to evaluate the impact of different values of the *temperature* parameter τ on the generation of such sets of probabilities.
The function `softmax_policy`, instead, already chooses an action according to such probability distribution.

N.B.: The same cost is applied for each direction, hence no particular weighting of the *Q-table* values is introduced.

```
def softmax_distribution(self,s,tau):
      # Returns a soft-max probability distribution over actions
      # Inputs:
      # - Q: a Q-function
      # - s: the state for which we want the soft-max distribution
      # - tau: temperature parameter of the soft-max distribution
      # Output:
      # - action: action selected based on the Boltzmann probability distribution

      p = np.zeros(len(self.q_table))
      sum_p = 0
      for i in range(len(p)):
          p[i] = np.exp((self.q_table[i,ind2sub(self.size[1],s)] / tau))
          sum_p += p[i]

      p = p / sum_p

      return p


  def softmax_policy(self,s,tau):
      # Returns a soft-max probability distribution over actions
      # Inputs:
      # - Q: a Q-function
```

```
        # - s: the state for which we want the soft-max distribution
        # - tau: temperature parameter of the soft-max distribution
        # Output:
        # - action: action selected based on the Boltzmann probability distribution

        p = np.zeros(len(self.q_table))
        sum_p = 0
        for i in range(len(p)):
            p[i] = np.exp((self.q_table[i,ind2sub(self.size[1],s)] / tau))
            sum_p += p[i]

        p = p / sum_p

        # Draw a random action from the distribution
        p_cumsum = np.cumsum(p)
        p_cumsum = np.insert(p_cumsum, 0, 0)

        choice = np.random.uniform(0,1)

        action = int(np.where(choice > p_cumsum)[0][-1])

        # Alternatively:
        # action = int(np.random.choice([0,1,2,3], p=p))

        return action
```

3. Reinforcement Learning Algorithm

The function that runs *Q-learning* on the MDP is defined below, along with some other functions that allow to reconstruct the **optimal** and the **human-generated** policies. Both a **generalised policy** (selected action for each state) and a **specific one** (the starting point is fixed and a unique path is derived) can be computed.

The purpose of the algorithm is updating a table, that tells which action is preferable when the agent is in a specific state. The update is performed thanks to the *Bellman Equation* and *exploration* of the environment is achieved by adding a small probability of choosing random actions instead of the optimal ones. Finally, three parameters can be tuned:

- **discount factor** $\gamma \in [0, 1)$: increasing it, the agent takes future rewards more and more into account,
- **learning rate** $\alpha \in [0, 1]$: an higher learning rate means a faster upgrade of the *Q-table*, due to an increased weight of the *temporal difference,*
- **exploration factor** $\epsilon \in [0, 1]$: this value is a probability threshold below which the agent takes a random action instead of the optimal one. This allows to explore the environment and avoids a premature convergence of the algorithm.

```
def update_q(self,a,s,s1,r,done):
      s = ind2sub(self.size[1],s)
      s1 = ind2sub(self.size[1],s1)

      if done:
          td = r - self.q_table[a,s]
      else:
```

```
            td = r + self.discount*np.max(self.q_table[:,s1]) - self.q_table[a,s]

        self.q_table[a,s] = self.q_table[a,s] + self.lr*td

        return td


    def egreedy_policy(self,epsilon):
        e = np.random.uniform(0,1)

        # N.B.: If all elements are equal, choose an action randomly
        if e < epsilon or (self.q_table[:,ind2sub(self.size[1],self.state)] == self.q_table[:,0][0]).all():
            action = np.random.randint(0,4)
        else:
            action = np.argmax(self.q_table[:,ind2sub(self.size[1],self.state)])

        return action


    def reconstruct_policy_from_q(self,start_state,optimality_flag,tau):
        policy = []
        actions = []
        state = copy(start_state)
        policy.append(copy(start_state))

        score = 0
        count = 0
        timeout = 2*self.size[0]*self.size[1]
        while state != self.end:

            if optimality_flag == "optimal":
                action = np.argmax(self.q_table[:,ind2sub(self.size[1],state)])
            elif optimality_flag == "human":
                action = self.softmax_policy(state,tau)

            # Prevent agent from being stuck in the same transition forwards and backwards
            if count >= 2 and state == policy[count-2]:
                while action == actions[count-2]:
                    action = np.random.randint(0,4)

            score += self.q_table[action,ind2sub(self.size[1],state)]

            if action == 0 and state[0]-1 >= 0: #North
                state[0] -= 1
            elif action == 1 and state[1]-1 >= 0: #West
                state[1] -= 1
            elif action == 2 and state[0]+1 < self.size[0]: #South
                state[0] += 1
            elif action == 3 and state[1]+1 < self.size[1]: #East
                state[1] += 1

            policy.append(copy(state))
            actions.append(action)

            # Interrupt path reconstruction if the policy does not converge to the goal
            count += 1
            if count >= timeout:
                break

        actions = action2str(actions)

        return policy, actions, score


    def get_policy_from_all_states(self,optimality_flag,tau):
        actions = []

        for i in range(self.size[0]*self.size[1]):
            if optimality_flag == "optimal":
                action = np.argmax(self.q_table[:,i])

            elif optimality_flag == "human":
```

```
                action = self.softmax_policy(sub2ind(self.size[1],i),tau)

            actions.append(action)

        actions = action2str(actions)
        return actions


    def q_learning(self,limit_step,nb_episode,algorithm="optimal",epsilon=0.1,tau=1):

        self.q_table = np.zeros((4,self.size[0]*self.size[1]))
        n_step = []
        global_temporal_differences = []

        for e in tqdm(range(nb_episode)):
            k = 0
            done = False
            self.rand_reset()

            if e % 100 == 0:
                self.visualize()

            temporal_differences = []
            while k < limit_step and not(done):

                if self.start == self.end:
                    pass

                if algorithm == "egreedy":
                    # Agent chooses next action according to an epsilon-greedy distribution
                    action = self.egreedy_policy(epsilon)
                elif algorithm == "softmax":
                    # Agent chooses next action according to a soft-max distribution
                    action = self.softmax_policy(self.state,tau)

                [new_state, reward, done] = self.step(action)

                err = self.update_q(action,self.state,new_state,reward,done)

                self.state = new_state
                k += 1
                temporal_differences.append(abs(err))

                if e % 100 == 0:
                    self.visualize()

            n_step.append(k)
            global_temporal_differences.append(temporal_differences)

        self.out.release()
        cv2.destroyAllWindows()

        return n_step, global_temporal_differences
```

4. Simulations

The hyperparameters of the algorithm have been set as follows:

- discount factor $\gamma = 0.95$,
- learning rate $\alpha = 0.8$,
- exploration factor $\epsilon = 0.2$,
- max_steps = 100: maximum number of updates of the *Q-table* before the exploration of the maze is stopped and another episode begins,

- n_episodes = 10000: number of episodes, i.e. number of generated scenarios in which an agent explores the maze starting from a random tile.

Several simulations have been performed and only some are reported in the following. In particular, the map of the optimal and human-derived actions is shown for the ***ooo*, *oxo*, *xoo* scenarios**, both for a **low temperature** value $(\tau = 0.1)$ and for an **higher** one $(\tau = 10)$.

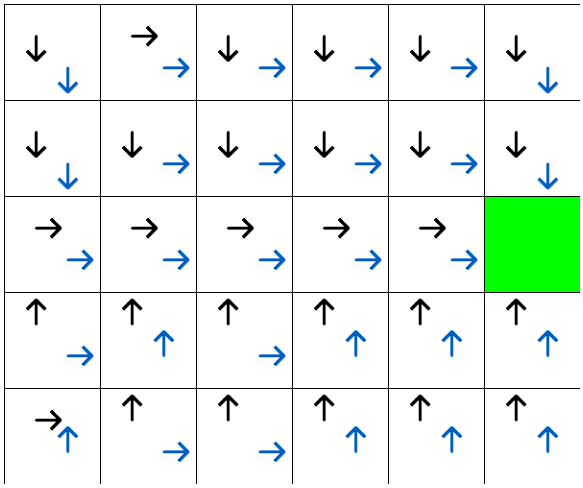

FIGURE 2a. Gridworld ooo, tau = 0.1

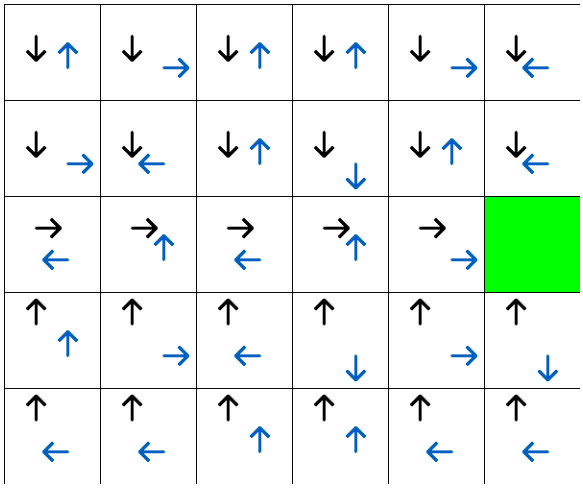

FIGURE 2b. Gridworld ooo, tau = 10

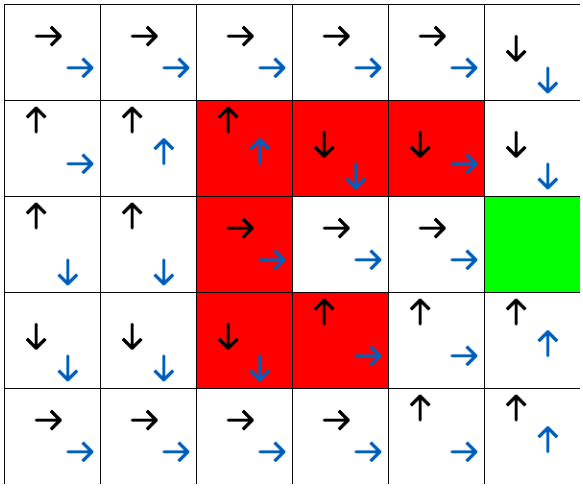

FIGURE 2c. Gridworld oxo, tau = 0.1

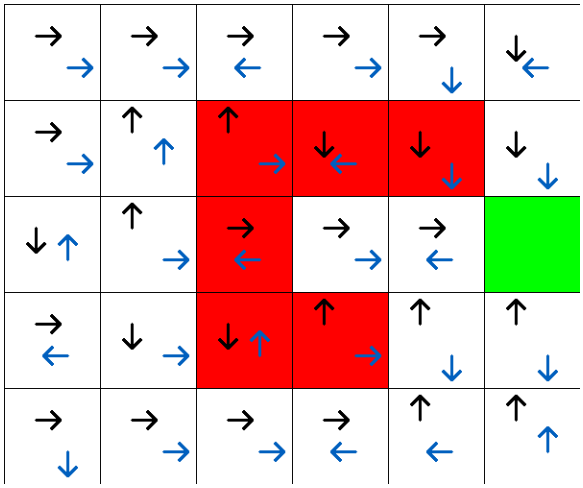

FIGURE 2d. Gridworld oxo, tau = 10

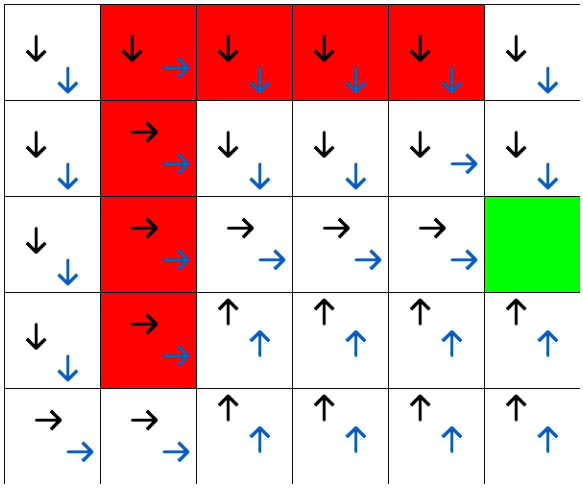

FIGURE 2e. Gridworld xoo, tau = 0.1

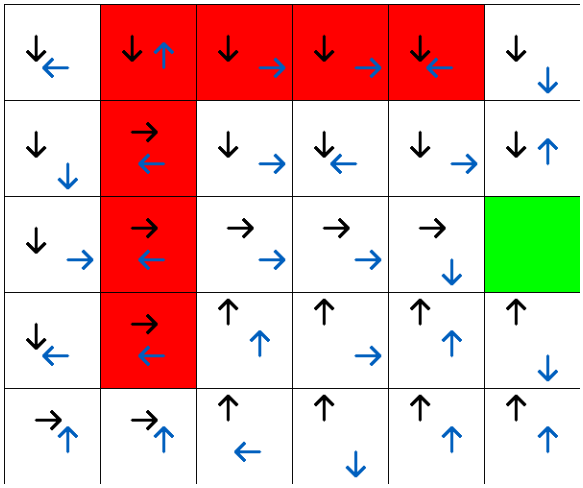

FIGURE 2f. Gridworld xoo, tau = 10

FIGURE 2 - Human policy (blue) vs Optimal policy (black). The goal state is the green tile, whereas the dangerous states are the red ones.

5. Analysis of the Temperature (τ) Parameter

For each one of the 8 mazes, the policy extraction according to the Boltzmann Model is carried out using the following values of the *temperature* parameter: $\tau \in [0.1, 0.25, 0.5, 1, 10, 100]$.

In each scenario and given a certain value of the temperature, the standard deviation among the 4 probability values is computed in each state and these results are averaged out to obtain a single metric, that represents the *dispersion* of the probability values among eachother. Repeating the computation for each value of the temperature, it can be seen that **the mean standard deviation decreases dramatically as the temperature increases, indicating a uniformity of the probability distribution.**

This is true for all mazes (trends are very similar) and in particular a significative drop can be noticed as τ goes from 1 to 10 (clearly due to the nonlinearity of the scale along the x-axis).

This behaviour can be mathematically explained by observing EQUATION 2: as τ increases, the probabilities tend to resemble more and more a uniform distribution, whereas for values of τ close to 0.1, the distribution is more skewed towards the optimal action. In particular, for very small values, the human policy tends to be nearly identical to the one derived by the optimal agent, as it can be seen in FIGURE 2 (mazes on the left column).

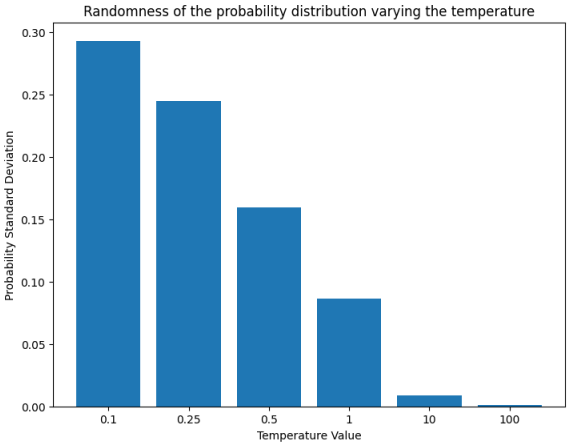

FIGURE 3a. Gridworld ooo

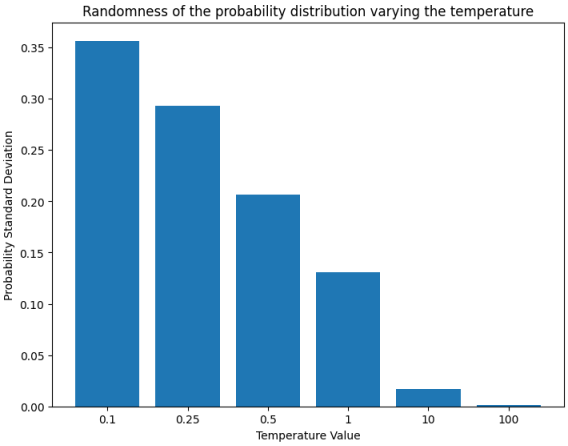

FIGURE 3b. Gridworld oox

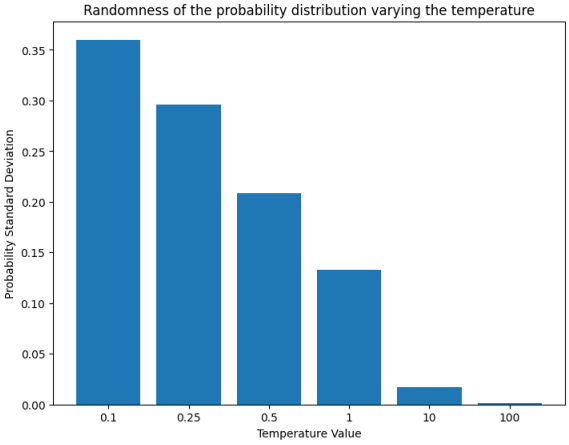

FIGURE 3c. Gridworld oxo

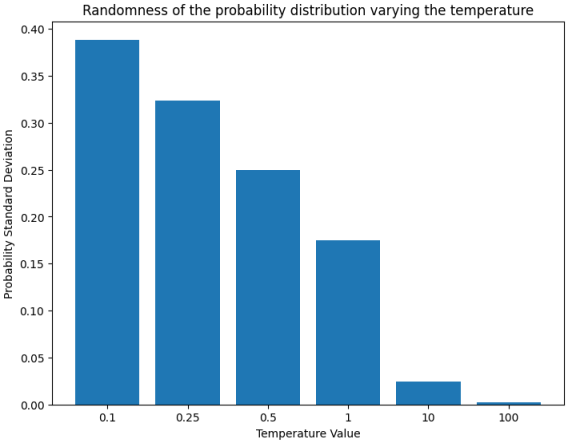

FIGURE 3d. Gridworld oxx

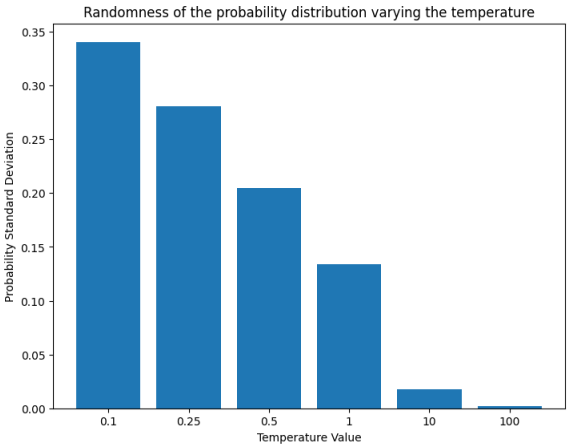

FIGURE 3e. Gridworld xoo

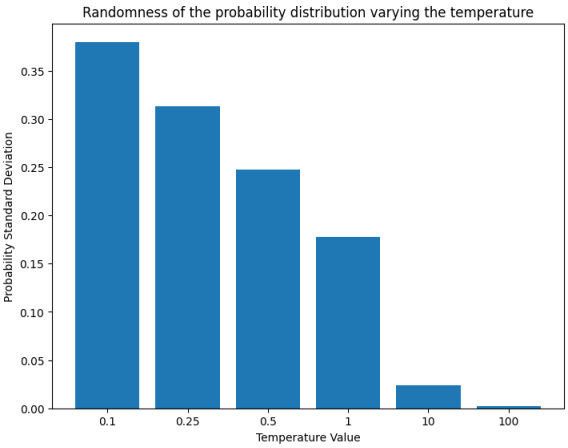

FIGURE 3f. Gridworld xox

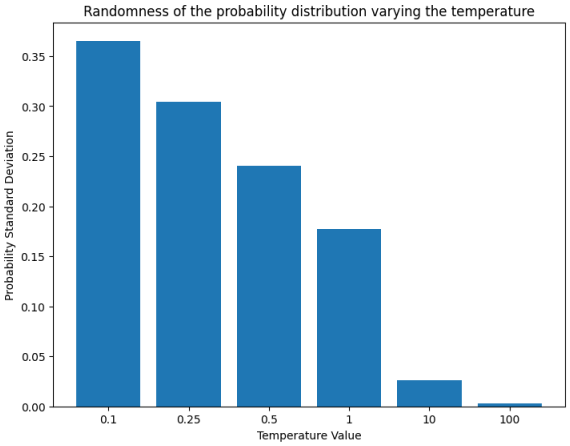

FIGURE 3g. Gridworld xxo

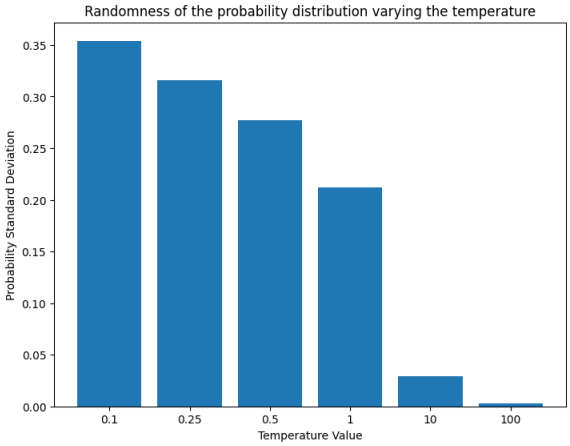

FIGURE 3h. Gridworld xxx

FIGURE 3: Mean standard deviation of the probability distribution for each state for the eight different scenarios, varying the temperature parameter τ.

6. *LESS is More* Approach

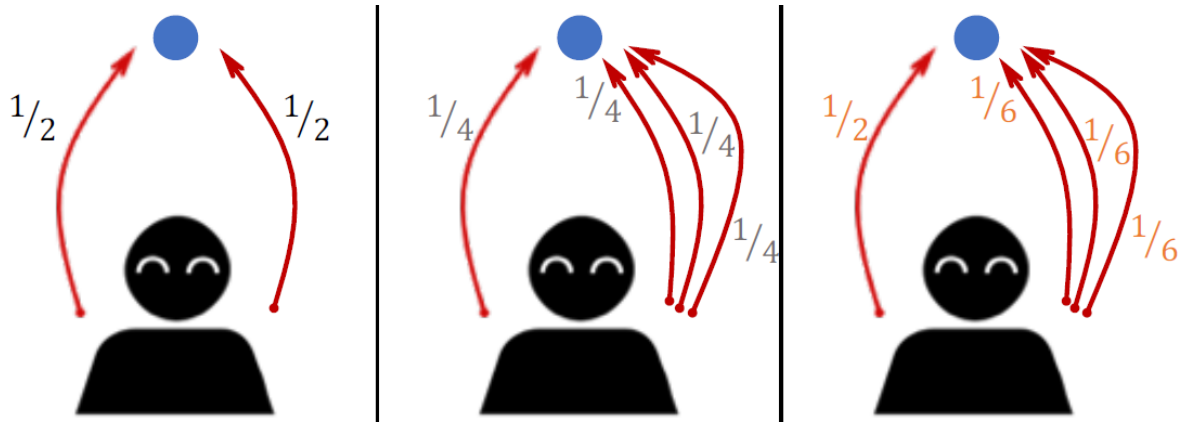

FIGURE 4 - (Left) Original options. (Center) New options and respective probability computed according to the Boltzmann Model. (Right) New options and respective probability computed according to the LESS approach.

In [3], a novel approach is proposed that allows to replace the Boltzmann Model with a more suitable method allowing to explicitly account for distances between trajectories, rather than only for their rewards. A *similarity* metric among trajectories has to be introduced, so that *similar* trajectories now affect the decision together.

In the noisily-rational problem of human decision modeling, actions can be discrete, but yet the Boltzmann model can produce unwanted results, as shown in FIGURE 4: if we add some trajectories, all equally good, to two equally good original paths, the probability will be equally split among all these trajectories, meaning that the left path will become less and less appealing compared to the right path. In order to overcome this issue, we wanted ideally that the overall probability would be the same on the left and on the right, subdividing the right probability among all paths that reach the goal from the right.

Moreover, if we consider that spaces of trajectories are continuous and bounded, we can see that they naturally contain a continuum of alternatives of varying similarity to each other, hence the Boltzmann model is even more inadequate in this case.

The solution proposed in the article is a strategy to influence probability through similarity in trajectories: *LESS*, *Limiting Errors due to Similar Selections*, computes a similarity value from freatures extracted from the different trajectories and, thanks to an *attribute rule*, softens even more the probability distribution - in order to take into account feature similarity - according to the model:

$$P(\xi) = \frac{e^{R(\phi(\xi))}}{\sum_{\bar{\phi} \in \Phi_{\Xi_f}} e^{R(\bar{\phi})}} \cdot \frac{s(\phi(\xi), \xi)}{\sum_{\bar{\xi} \in \Xi_f} s(\phi(\xi), \bar{\xi})} \cdot$$

This approach allows to reduce the weight given to similar trajectories, so that equally valid actions (e.g.: reaching the goal from the left or from the right) will have more or less the same probability, no matter how many very similar trajectories represent one or the other action.

Bibliography

[1] Ho, M., Littman, M., Cushman, F., & Austerweil, J.L. (2018). Effectively Learning from Pedagogical Demonstrations. Cognitive Science.

[2] Milli, S., & Dragan, A.D. (2019). Literal or Pedagogic Human ? Analyzing Human Model Misspecification in Objective Learning. UAI.

[3] Andreea Bobu, Dexter R.R. Scobee, Jaime F. Fisac, S. Shankar Sastry, and Anca D. Dragan. 2020. LESS is More: Rethinking Probabilistic Models of Human Behavior. In Proceedings of the 2020 ACM/IEEE International Conference on Human-Robot Interaction (HRI ’20), March 23–26, 2020, Cambridge, United Kingdom.