Reinforcement Learning Assignment 2022-2023

Aris Tsilifonis 1115201700170
Department of Informatics and Telecommunications

The purpose of the course was to create a project in the field of reinforcement learning using an environment from the gym API. In Reinforcement learning we usually try to obtain the reward from the environment .However, this is not always the case because for some real life tasks ,such as driving , the reward function cannot always be exactly determined.This happens because that procedure involves many aspects to consider such as the traffic or the weather and road conditions. Inverse reinforcement learning helps as provide a reward function which is difficult to be expressed by the expert agent .Inverse Reinforcement Learning (IRL), as described by Andrew Ng and Stuart Russell in 2000, reverses the problem and instead tries to extract the reward function from an agent's observed behavior.IRL is about determining an unknown reward function for an MDP, given an optimal policy for that MDP.

Regarding my implementation , I chose the gridworld , which is a 4x4 grid where the agent can take four different actions (down,up,right,left) and its goal is to reach top left and bottom right corner.It is a Markov decision process stochastic process which includes S=[1,..,16] finite number of states ,A=[1,..,4] $i \subseteq S_A$, T: probability for transition from state i to state j and R : the reward for taking a specific action a in that state s.The project consists of four different algorithms :value iteration , inverse reinforcement learning using linear programming ,max entropy inverse reinforcement learning and deep max entropy reinforcement learning . All the algorithms were based on papers that I list below.The reader needs to know that there are two main classes of policy-finding algorithms: Model-based and model-free. A model-based algorithm either knows the transition probabilities of the MDP, or forms some explicit approximation of them from observed trajectories. A model-free algorithm does not.In this project, I implemented the value iteration algorithm.

This is a model-based algorithm that relies heavily on knowledge of the transition probabilities. It estimates the value function for each state. The policy in any given state is the single action that is expected to produce the highest reward value.

First of all, I will elaborate on my implementation of value iteration. I used the gridworld that exists on the e-class of the course. In my code , I extracted tuples which contain the information that our algorithm needs .Those were in the form :{0: [(1.0, 0, 0.0, True)], 1: [(1.0, 0, 0.0, True)] etc. This means that for every state we have a tuple for every action containing the transition probability , the number of next state and the reward for that transition as well as if the original state is a terminal state or not .These tuples were used to create a transition matrix which has size (16, 4, 16) .This means that for every state and for each action on that state determine the next state by taking the specific action with one hot encoded vector.The place of 1 (0-15) determined which state out of 16 our agent will transition.With similar procedure I managed to parse the rewards using the tuples that the environment provides.The reward is about each state of the environment , for the one that we are using it is [0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. 0.]

Moving on the next part of my program, I will analyze how my implementation of value iteration operates. My function is defined:
def value_iteration(transitions , reward , states, actions, g=0.9,eps=1e-3):
which get as input the transition probabilities ,the rewards, the states,the actions, the discount factor gamma(g)

input : reward function $r(s)$, transitional model $p(s'|s,a)$,
discounted factor γ, convergence threshold θ
output: optimal policy π^*
initialize $v(s)$ with zeros
converge $\leftarrow$ false
while converge $=$ false **do**
  $\Delta \leftarrow 0$
  for $s \in S$ **do**
    temp $\leftarrow v(s)$
    $v(s) \leftarrow r(s) + \gamma \max_a \sum_{s'} p(s'|s,a)v(s')$
    $\Delta \leftarrow \max(\Delta, |\text{temp} - v(s)|)$
  end
  if $\Delta < \theta$ **then**
    converge $\leftarrow$ true
  end
end
for $s \in S$ **do**
  $\pi^*(s) \leftarrow \text{argmax}_a \sum_{s'} p(s'|s,a)v(s')$
end
return π^*

My implementation of the above algorithm:

```python
def value_iteration(transitions , reward , states, actions, g=0.9,eps=1e-3):
    V1 = {x: 0 for x in range(states)}#initialize V
    while True:
        d = 0
        V = V1.copy()#update new V
        for s in range(states):
            clist = []
            for a in range(actions):
                b = enumerate(transitions[s, a, :])
                c = 0.0
                for i in b:
                    c += i[1] * V[i[0]]
                clist.append(c)
            V1[s] = reward[s] + g * max(clist) #return maximum argument of [list of 4 actions]
            release_list(clist)
            d = max(d, abs(V1[s] - V[s]))
        if d < eps * (1 - g) / g:
            return V
```

$$v(s) = \max_{a_i} \left[r(s) + \gamma \sum_{s'} p(s'|s, a = a_i) v(s') \right]$$
$$= r(s) + \gamma \max_{a} \left[\sum_{s'} p(s'|s, a) v(s') \right]$$

This is called the Bellman equation after Richard Bellman and this is the key to solving MDP. In other words, to solve MDP is to solve the Bellman equation. The above equation tells us that we should choose the action that gives us the highest utility of the state. In other words, among all possible a_i, we should choose the a_i that has the highest $v(s|a=a_i)$. By definition, this utility value is then v(s) because it is the best outcome we can get after we reach the state s.As we see , we need to initialize delta and v to zero after we loop through all the states. In order to check if delta fell down the threshold theta i used the mathematical formula d< epsilon*(1–gamma/gamma). where epsilon is a scale parameter.(remind you that gamma is discount factor so it's definitely below 1).Since now the utility is defined as associated with the optima policy, v(s′) is already the best we can get after we reach that state s′. In other words, if we choose to go to s′, there may be multiple paths in the future following s′, but v(s′) is defined as the outcome of the best path among all these possible paths.Then , I will elaborate how my implementation works in the code snippet above.

As we know from theory , s’ is the value of the previous state of the agent.In the line: b = enumerate(transitions[s, a, :]) we get an enumeration of (s’,transition_probability). We use that pairs to create the sum that is depicted in the image above.In detail, i[1] is transition probability for that specific action and V[i[2]] is utility of its successor state .We create a list of those sums, one per action and we take the maximum value of all items in the list which is then multiplied by the discount factor gamma. d = max(d, abs(V1[s] - V[s])) is to ensure that we check the proper threshold value. The update step is very similar to the update step in the policy iteration algorithm. The only difference is that we take the maximum over all possible actions in the value iteration algorithm.instead of evaluating and then improving, the value iteration algorithm updates the state value function in a single step. This is possible by calculating all possible rewards by looking ahead.The value iteration algorithm is guaranteed to converge to the optimal values.

Another important function of the program is to find the optimal policy based on the transition probabilities of the environment and the value iteration result which is the transition probability V(s') s':(previous state).We build exactly the same sum as in value iteration algorithm.Inside the function expected utility which is listed below we compute the agents best policy .Mathematically , this is defined in the solution as

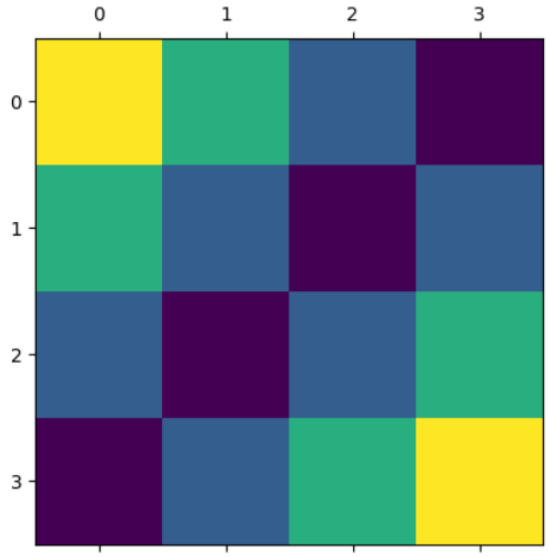
$$\pi(s) \leftarrow \operatorname{argmax}_a \left[\sum_{s'} p(s'|s,a) v(s') \right]$$

This equation is converted into code in this way:

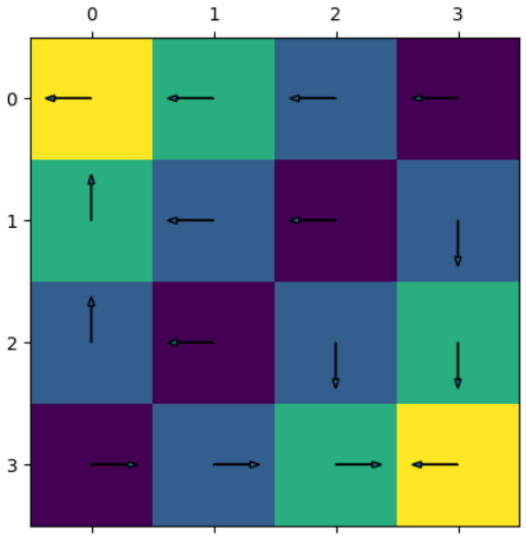
```python
def utility(transitions,state,a, V):
    b = enumerate(transitions[state, a, :]) # [0. 0. 0. ... 0. 1. 0.] multiply 0*0.0 + .. + 15*1.0+16*0.0
    c = 0.0
    for i in b:
        c += i[1] * V[i[0]] #probability * V(s') s':(previous state)
    return c

def optimal_policy(transitions, V):
    states, actions, _ = transitions.shape
    policy = {}
    #for every state find optimal action
    for s in range(states):
        dlist = []
        policy[s] = max(range(actions), key=lambda a: utility(transitions,s,a, V))#compute utility for every action, return action with
    return policy
```

Utility function does the same operation as in value iteration.Regarding the optimal policy function , its purpose is to compute utility for every action, return action with max value .That is happening in the line : policy[s] = max(range(actions), key=lambda a: utility(transitions,s,a, V) were we compute 4 different values of utilities and the action that has the maximum utility is returned as an argument to form the optimal policy.The result of optimal policy for own example is: {0: 0, 1: 3, 2: 3, 3: 2, 4: 0, 5: 0, 6: 0, 7: 2, 8: 0, 9: 0, 10: 1, 11: 2, 12: 0, 13: 1, 14: 1, 15: 0} which means the best action we can take for each state.

Finally there are some plots that help us improve our understanding of the solution.In particular, I designed a function that plots a heatmap of the grid based on the results of value iteration .



The brighter colors (yellow,green) mean higher probability to reach goal state and the darker colors purple lower accordingly.



I tried to picture the transitions that the agent should opt to do based on the probabilities of value iteration and the best policy that the function above returns.As it can be clearly observed the expert agent is trying to move closer to the yellow corner in accordance to the best policy that we computed. That is logical based on the Markov Decision Problem that we chose, so it is a way to validate my results because the arrows look towards the goal states.I will move forward to the implementation of the next algorithm based on inverse RL.

Linear Programming solution for inverse Reinforcement Learning

source : https://ai.stanford.edu/~ang/papers/icml00-irl.pdf

In my Inverse reinforcement learning implementation , the number of states S = [0-15] and optimal policy a1 are known from previously presented code.The paper above, proposed this solution :

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} \min_{a \in \{a_2, \ldots, a_k\}} \{(\boldsymbol{P}_{a_1}(i) - \boldsymbol{P}_a(i)) \\ & (\boldsymbol{I} - \gamma \boldsymbol{P}_{a_1})^{-1} \boldsymbol{R}\} - \lambda ||\boldsymbol{R}||_1 \\ \text{s.t.} \quad & (\boldsymbol{P}_{a_1} - \boldsymbol{P}_a)(\boldsymbol{I} - \gamma \boldsymbol{P}_{a_1})^{-1} \boldsymbol{R} \succeq 0 \\ & \forall a \in A \setminus a_1 \\ & |\boldsymbol{R}_i| \leq R_{\max}, \quad i = 1, \ldots, N \end{aligned}$$

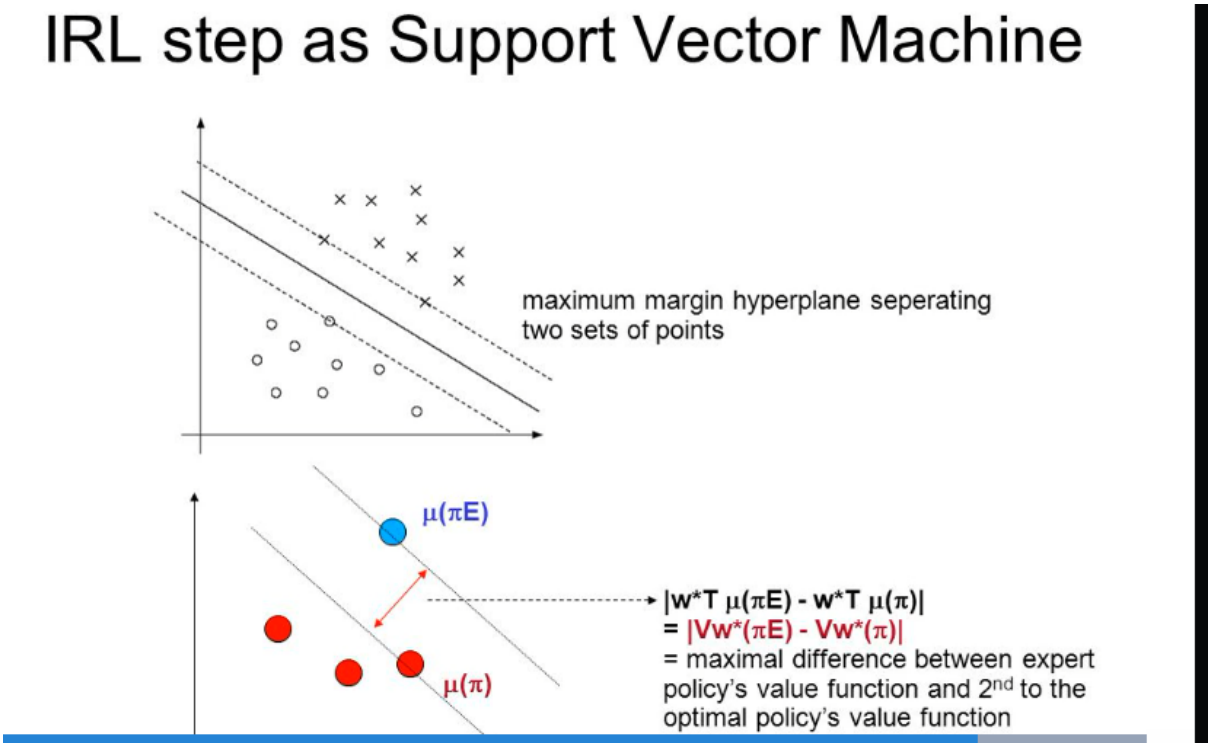
R is a vector containing the reward for each state.(I-γ*Pa1) is always invertible.To see this , first note that Pa1, being a transition matrix has all eigenvalues in the unit circle in the complex plane .Since γ<1 this implies that the matrix γ*Pa1 has all the eigenvalues in the interior of the

unit circle(and in particular that 1 is not an eigenvalue).This means (I-γ*Pa1) has no zero eigenvalues,and is thus not singular.(paper notes pg3). This objective function means that we are trying to find a reward vector that maximizes the difference in expected reward from the action with the smallest difference in expected reward from the second best action(sub-optimal policy).Since the above function cannot be interpreted as a linear programming solution, it can be transformed as :

$$\text{maximize} : \sum_{i \in S} \{t_i - lu_i\}$$

$$\begin{aligned} subject : \left(P_{a_1}(i) - P_a(i)\right)\left(I - cP_{a_1}(i)\right)^{-1} R \geq t_i \\ \left(P_{a_1}(i) - P_a(i)\right)\left(I - cP_{a_1}(i)\right)^{-1} R \geq 0 \\ -u \leq R \leq u \\ |R_i| \leq R_{max} \end{aligned}$$

Regarding the first constraint , which is non existent in the aforementioned paper , I got the idea from another paper of Pieter Abbeel and Andrew Ng. The paper elaborates on the topic of Apprenticeship Learning via Inverse Reinforcement Learning.



https://ai.stanford.edu/~ang/papers/icml04-apprentice.pdf

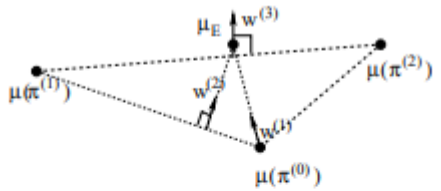


Figure 1. Three iterations for max-margin algorithm.

the reward function being optimized by the expert. The maximization in that step is equivalently written

$$\max_{t,w} \quad t \tag{10}$$

$$\text{s.t.} \quad w^T \mu_E \geq w^T \mu^{(j)} + t, \; j = 0, \ldots, i-1 \tag{11}$$

$$||w||_2 \leq 1 \tag{12}$$

As you can see , I limit the maximal difference between expert policy value function and second to the optimal policy value function by a margin t_i.(μ is feature expectation)

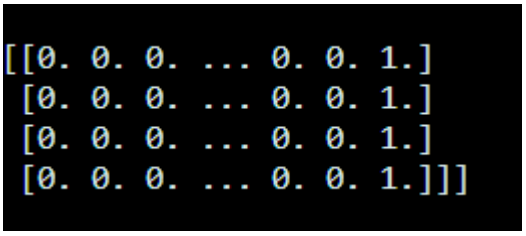
In order to better understand the solution , I will present to you some very important requirements in linear programming programmes.

Elements of a basic LPP

In python linear programming uses matrices c,A_ub,b
where A_ub@x <= b where the objective function is to minimize a function for example c@x :
c1-D array
The coefficients of the linear objective function to be minimized.
A_ub 2-D array
b_ub 1-D array
The inequality constraint vector. Each element represents an upper bound on the corresponding value of A_ub @ x.
All the info where obtained by python documentation:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html
Then I will proceed explaining the technical details of my program.I have the same functions V = value_iteration(transitions, reward,states,actions) , op = optimal_policy(transitions, V) so as to obtain necessary information for my algorithm.

lp_irl(transitions, op,states , actions) function implements the aforementioned functionality when we pass transitions, optimal policy ,states and actions for our MDP.

We will have a closer look to the exact code:
def T(tp, policy,eyes_2s,g ,a, s):
return np.dot(tp[policy[s], s]-tp[a, s],np.linalg.inv(eyes_2s-g* tp[policy[s]]))
 #because we have identity matrix , we need tp[policy[s]]
tp is transitions probabilities is a matrix transpose [actions,states,states] from the original matrix [states,actions,states] where the 3rd dimension contain the one hot encoded vectors that were computed in the value_iteration program.Transposing the matrix is essential for the T function to align the values in order to compute the inverse .Last 2 dimensions of the array must be square.



T function returns an array with size 1*states (each value depends on the difference of transition probabilities.
c must be a 1-D array and must not have more than one non-singleton dimension
c = -np.r_[zero_1s, one_1s, -l1 * one 1s]
All variables are 1d vectors because the objective function is now linear .I put the coefficients of the variables that can be reviewed in the objective function I proposed. For instance , t has one_1s as coefficient where the number of ones is determined by the number of states of the MDP. Because R is not present in the objective function we have zero values as coefficients. Regarding the minus , it is because the objective function goal is to minimize the difference of the variables accordingly. Then the linear programming python utility will have to maximize the solution for us.

Next is, T_stackmat

```
alist = []
for s in range(states) :
    for a in action_set - {policy[s]}:
        alist.append(np.eye(1,states,s))
I_stackmat = np.vstack(alist)

blist = []
for s in range(states) :
    for a in action_set - {policy[s]}:
        blist.append(-T(tp, policy,eyes_2s,g, a, s))
T_stackmat = np.vstack(blist) #(n_states*n-1_actions)*nstates
```

(states*n-1_actions)*nstates loop runs(stacked arrays)
This is the core of the inverse RL programming method because our expert agent will try to test the other suboptimal actions from the set .This is due to the fact that there is a possibility of improvement in optimal policy.

```
A_ub = np.bmat([[T_stack, zero_stack, zero_stack], # -TR <= 0
                [T_stack, I_stack, zero_stack],  # TR>=t => TR-t>=0 => -TR +t<=0
                [-eyes_2s, zero_2s, -eyes_2s],   # -R <= u
                [eyes_2s, zero_2s, -eyes_2s],  # R <= u
                [eyes_2s, zero_2s, zero_2s], # R <= Rmax
                [-eyes_2s, zero_2s, zero_2s],  # -R <= Rmax
                ])
```

In the represented code, the constraints are formed based on the theoretical setting .The reader must take into account that reward function is a vector of n_states .Moreover, its columns in A_ub represent a coefficient of the constraints : first column for R coefficient, second column for t coefficient and and third for u (they are aligned with the elements of c matrix).
In the context of linear programming, the reward matrix (represented as R in literature) is a matrix that depicts the rewards associated with transitioning from one state to another. The matrix has dimensions of states because each element in the matrix shows the reward associated in each state.

This is different from the reward that linprog returns, which is the optimal value of the objective function. In the case of Markov Decision Processes, the objective function is often defined as the expected cumulative reward after a certain time step, and linprog is used to find the policy that maximizes this expected reward. Let’s review the

inequalities one by one to showcase the way it is transformed from theory to code.
R <= Rmax:
The coefficient on this occasion is the identity matrix (it produces the same matrix) for the reward variable.Rmax is the bound of the variable so it should not be in the A_ub matrix.
#-R <= Rmax: I defined the constraint in a similar way.
R <= u => R-u<=0: Coefficient of R value is the same as before. Linear variable u needs to be multiplied by identity matrix too in order to be present in the constraint matrix
#-R <= u : Similarly
-TR <= 0:
Regarding the T it is the equation that we defined previously on pg.8
The matrices can be multiplied since (x*n_states)@(n_states*1)
TR>=t => TR-t>=0 => -TR +t<=0 (it must be <= for the algorithm)
-T is already computed
T and t need to have the same number of rows which can happen when t is multiplied by the matrix I_stackmat where:
I_stackmat has (states*n-1_actions)*nstates loop runs(stacked arrays)
I_stackmat contains stacks of arrays np.eye(1,states,s) which are in the form
[[0. 0. 0. ... 0. 0. 1.] . The 1 value is important in order to get the correct t element for the corresponding state, action pair and subtract it from T@R.

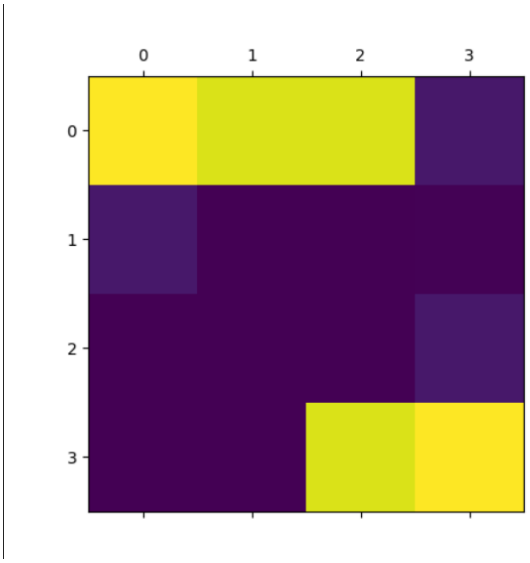
The inequality constraint vector b_ub. Each element represents an upper bound on the corresponding value of A_ub @ x.
The values need to be vertically stacked in order for the algorithm to work.
bmat works in a similar way too.

```
b_ub = np.vstack([np.zeros((states * (actions-1) * 2 + 2 * states, 1)),
                  Rmax * np.ones((2 * states, 1))])
```

the number of rows in A_ub must equal the number of values in b_ub.

Finally, return results["x"][:states] allows us to get the reward from every state ,which are in the form

[4.8 4.51875 4.51875 0.3 0.3 0.0. 0.01875 0. 0. 0.
0.3 0. 0. 4.53310089 4.8] (16 values)



I present a heatmap with the reward for every state of the MDP.
The figure above shows the results of linear programming inverse RL implementation. The correct answer has the brightest color and it is only in the upper left and lower right like the map.

—----------

Sources:
https://www.aaai.org/Papers/AAAI/2008/AAAI08-227.pdf?source=post_page----------

https://ai.stackexchange.com/questions/27500/why-is-it-that-the-state-visitation-frequency-equals-the-sum-of-state-visitation

https://jonathan-hui.medium.com/rl-inverse-reinforcement-learning-56c739acfb5a

Eclass paper : A Survey of Inverse Reinforcement Learning: Challenges, Methods and Progress

https://www.ics.uci.edu/~dechter/courses/ics-295/winter-2018/presentations/Dheeru.pdf

Max Entropy solution for inverse Reinforcement Learning

The Principle of Maximum Entropy is based on the premise that when estimating the probability distribution, you should select that distribution

which leaves you the largest remaining uncertainty (i.e., the maximum entropy) consistent with your constraints. That way you have not introduced any additional assumptions or biases into your calculations.

Ziebart et al. build on Abbeel & Ng's approach to removing multiplicity in the possible reward functions. They introduced a method of matching feature expectations between observed paths and optimal paths for the reward functions that were recovered . By generating a policy which is optimal for our reward function, we would expect that on average it generates the same paths as the optimal policy for the true reward function. This approach thus recovers reward functions that can be learned from using standard reinforcement learning methods to recover policies similar to the optimal policy. The observed feature expectations from our N observed trajectories is a simple average over feature counts.

counts, $\mathbf{f}_\zeta = \sum_{s_j \in \zeta} \mathbf{f}_{s_j}$, which are the sum of the state features along the path.

$$\text{reward}(\mathbf{f}_\zeta) = \theta^\top \mathbf{f}_\zeta = \sum_{s_j \in \zeta} \theta^\top \mathbf{f}_{s_j}$$

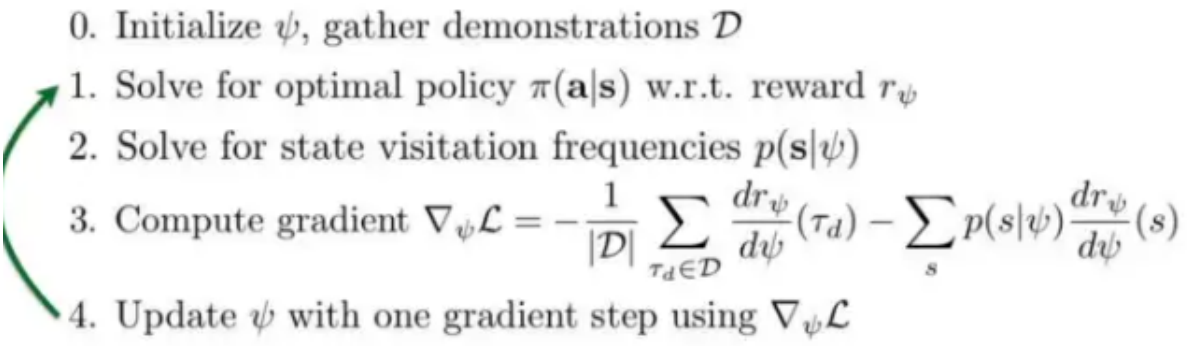
The agent demonstrates single trajectories, $\tilde{\zeta}_i$, and has an expected empirical feature count, $\tilde{\mathbf{f}} = \frac{1}{m} \sum_i \mathbf{f}_{\tilde{\zeta}_i}$, based on many (m) demonstrated trajectories.

Regarding my implementation ,I presented feature counts by incrementing a vector of size (1,states) with one hot encoded vector The place of 1 means the current state of each step in each trajectory. For this purpose I passed as feature matrix the identity matrix which each row represents one state.In my code every step is a tuple of (current_state, policies[current_state], next_state) .If we reach goal state before the end of trajectory , current state equals next_state. In my work , each trajectory consists of 5 steps and the sum of all trajectories equals 150 by default. Trajectories are produced randomly by the gym environment operations.

After I compute feature expectation , which could be described as the average path across all the trajectories, I am ready to run max entropy inverse-RL algorithm.After some mathematical calculation which can be found in the link at the sources, the following gradient of the entropy

We can combine states in trajectories, as below

$$\nabla_\theta L = \frac{1}{M}\sum_s \mathbf{f}_s - \sum_s P(s|\theta)\mathbf{f}_s$$,where $P(s|\theta)$ is the state visitation frequency



Source

Instead of step 3 , I provided a simpler solution in the picture above.
source:
https://docs.google.com/document/d/1yMWXWFEFuLRE_184AkIiHwUVw4rjpLKkb22y55b4FOA/edit

```
for _ in range(epochs):
    reward = features_array.dot(thita)
    V = value_iteration(transitions, reward,states,actions, g)
    op = optimal_policy(transitions, V)
    svisit_fq = state_visitation_frequency(transitions, op , trajectories,states_num)
    gradient = feature_expectation - features_array.dot(svisit_fq)
    thita = thita + learning_rate * gradient

return features_array.dot(thita)#return reward
```

About state visitation frequency , it is best explained on this link.
https://ai.stackexchange.com/questions/27500/why-is-it-that-the-state-visitation-frequency-equals-the-sum-of-state-visitation
It is basically the equation from the eclass paper about inverse RL.
A Survey of Inverse Reinforcement Learning: Challenges, Methods and Progress

$$\phi^{\pi}(s) = \phi^{0}(s) + \sum_{s' \in \mathcal{S}} P(s, \pi(s), s')\phi^{\pi}(s').$$

The state visitation frequency is defined as the discounted sum of probabilities of visiting a given state.
φπ(s') is the state_visitation_frequency of the previous state.P(s,π(s),s') is the probability of transitioning from state s to s' with optimal policy π(s). The specific steps of the state_visitation_frequency are:

$\mu_t(s)$ probability of visiting s at t

$\mu_1(s) = p(s_1 = s)$

for $t = 1$ to T

$$\mu_{t+1}(s') = \sum_{a} \sum_{s} \mu_t(S)\, \pi(a|s)\, p(s'|s, a)$$

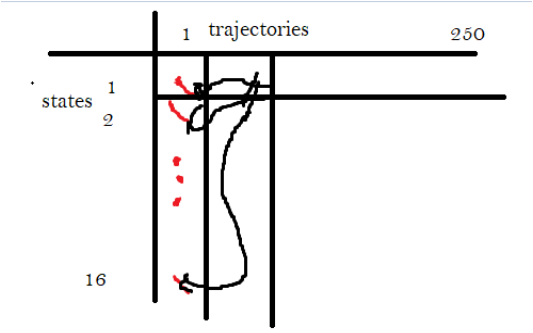
$$p(s|\psi) = \frac{1}{T} \mu_t(s)$$

μt+1 is the equation described above.
I divided the whole sum with the number of trajectories. You should know that the size of svf matrix quals to the numbers of states * trajectory number because for each trajectory we update the value of svf for every state.

```
svf = np.zeros((states_num, trajectories_num))

#get start state for first step from every trajectory
for trjs in trajectories:
    svf[trjs[0][0], 0] = svf[trjs[0][0], 0] + 1
```

.About computing the φ0(s), I am counting how many times a state is visited at the first step of each trajectory.
With dynamic programming , I compute φπ(s)

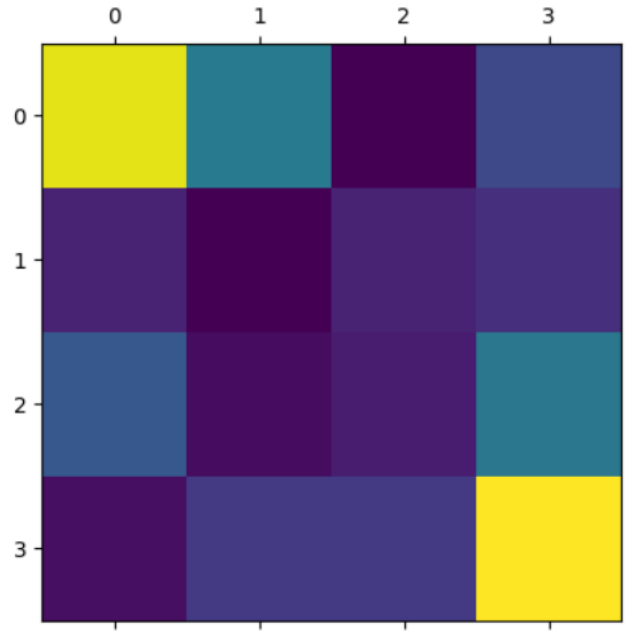
```
#loop for every trajectory updating the value for every state
for trj in range(1, trajectories_num):
    for st in range(states_num):
        svf_var = []
        for prev_st in range(states_num):
            svf_var.append( transitions[prev_st, policies[prev_st], st] * svf[prev_st, trj - 1])
        svf[st, trj] = sum(svf_var)
```



An explanation of svf algorithm

In order to compute φπ(s) for every state I need to traverse all the previous states(s') and compute the respective sum of products. As you can see every new svf term is constructed by the previous terms (trj-1 second dimension).I order to improve my gradient , I perform gradient ascent : thita = thita + learning_rate * gradient (maximization problem)

The resulting reward distributed is the gridworld matrix is



This is also reasonably well estimation of the reward . Even if the actual optimal policy is not known, it can be approximated using only the trajectory sequence from an expert , with a clearly better approximation than linear programming (without its constraints) .

Sources: https://arxiv.org/pdf/1507.04888.pdf

Max Entropy Deep inverse Reinforcement Learning

In the previous implementation the reward function was approximated in a linear way . On this attempt , I replaced that function with one that is produced by a neural network. The code has many similarities with max entropy . Based on the paper that is listed above , I wrote code that is described by this algorithm :

expert's state action frequencies μ_D^a, which are needed for the calculation of the loss are summed over the actions to compute the expert state frequencies $\mu_D = \sum_{a=1}^{A} \mu_D^a$.

Algorithm 1 Maximum Entropy Deep IRL

Input: $\mu_D^a, f, S, A, T, \gamma$

Output: optimal weights θ^*

$\theta^1 = \text{initialise\_weights}()$

Iterative model refinement

for n = 1 : N **do**

  $r^n = \text{nn\_forward}(f, \theta^n)$

Solution of MDP with current reward

  $\pi^n = \text{approx\_value\_iteration}(r^n, S, A, T, \gamma)$

  $\mathbb{E}[\mu^n] = \text{propagate\_policy}(\pi^n, S, A, T)$

Determine Maximum Entropy loss and gradients

  $\mathcal{L}_D^n = \log(\pi^n) \times \mu_D^a$

  $\frac{\partial \mathcal{L}_D^n}{\partial r^n} = \mu_D - \mathbb{E}[\mu^n]$

Compute network gradients

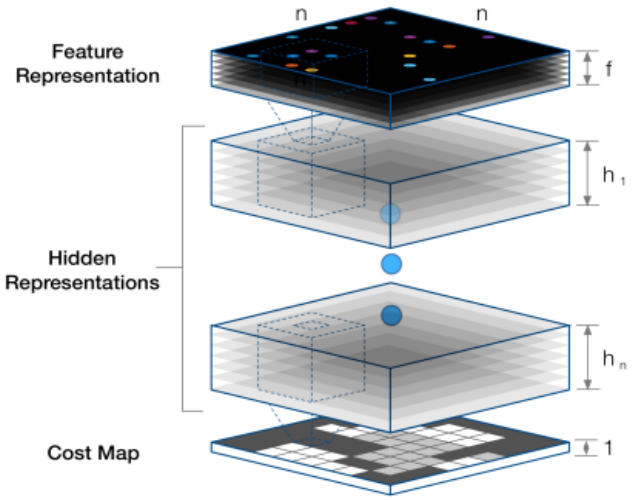
  $\frac{\partial \mathcal{L}_D^n}{\partial \theta_D^n} = \text{nn\_backprop}(f, \theta^n, \frac{\partial \mathcal{L}_D^n}{\partial r^n})$

  $\theta^{n+1} = \text{update\_weights}(\theta^n, \frac{\partial \mathcal{L}_D^n}{\partial \theta_D^n})$

end for

The derivative of the Maximum Entropy objective with respect to the reward equals the difference in state visitation counts between solutions given by the expert demonstrations and the expected visitation counts for the learned systems trajectory distribution in 12.

$$\mathbb{E}[\mu] = \sum_{\varsigma:\{s,a\}\in\varsigma} P(\varsigma|r) \tag{12}$$



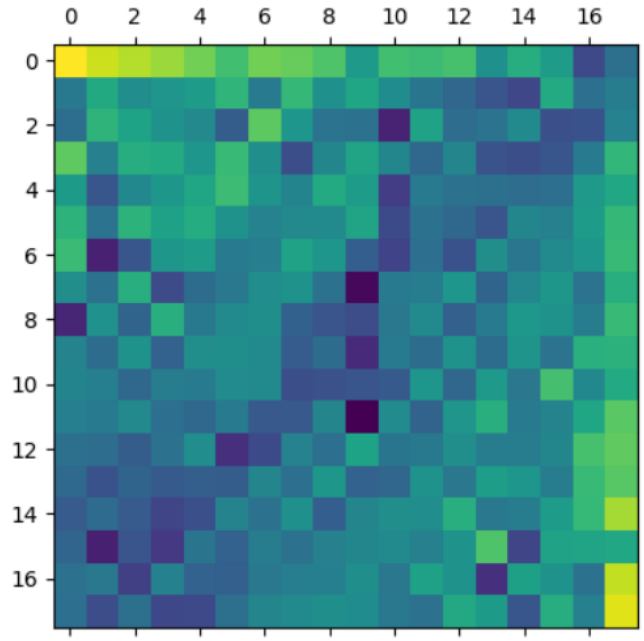
μD is the features expectation and E[μ] state visitation accordingly.

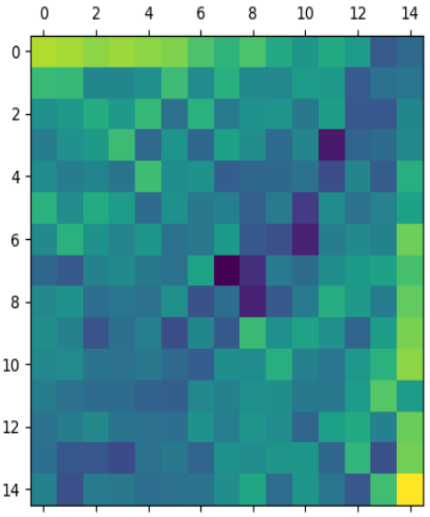
For the purpose of this algorithm , I defined a neural network with chainer framework , which has 4 layers and 64 hidden units.The input of this neural network is the dimensionality of the features(the number of states:16).I used relu and tanh for activation function.I used Adam optimizer for this network with weight decay to balance features.

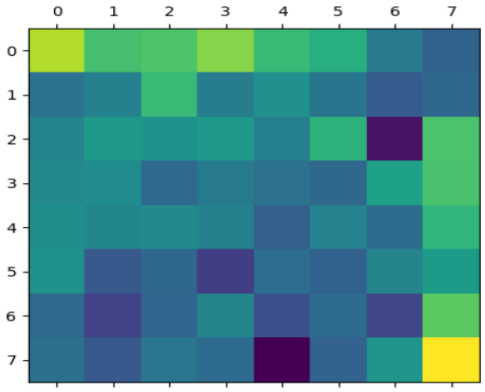
```
class Class_Reward(chainer.Chain):
    def __init__(self, inputi, hiddeni):
        super(Class_Reward, self).__init__(
            ln1=Link.Linear(inputi, hiddeni),
            ln2=Link.Linear(hiddeni, hiddeni),# hidden layer
            ln3=Link.Linear(hiddeni, 1)
        )

    def __call__(self, x):
        h1 = Func.relu(self.ln1(x))#activation function
        h2 = Func.tanh(self.ln2(h1))
        return self.ln3(h2)
```

```
    feature_expectation = np.zeros(states_dim)#(1,states) ->size

    NN_reward = Class_Reward(states_dim,64, 128)
    optimizer = optimizers.Adam()
    optimizer.setup(NN_reward)
    optimizer.add_hook(chainer.optimizer.WeightDecay(1e-3))

    feature_expectation = f_exp(features_array,feature_expectation,trajectories)

    f = chainer.Variable(features_array.astype(np.float32)) # this time there is no thita. I compute reward variable with a
    for _ in range(epochs):
        NN_reward.zerograds()#reset gradients
        reward_var = NN_reward(f)#compute reward
        vi = value_iteration(transitions, reward_var.array,states,actions, g)#same steps as max_ent
        pi = optimal_policy(transitions, vi) #same steps as max_ent
        svisit_fq = state_visitation_frequency(transitions, pi , trajectories,states_num) #same steps as max_ent
        L_th_grad = feature_expectation - svisit_fq #compute gradient
        L_th_grad = L_th_grad.reshape((states_num, 1)).astype(np.float32)
        reward_var.grad = -L_th_grad #balance rewards
        reward_var.backward()
        optimizer.update()

    return NN_reward(f).array#return reward
```

The process that I followed is similar any neural network api because I reset gradients at the start of every epoch and I have a backwards step(The aim of backpropagation (backward pass) is to distribute the total error back to the network so as to update the weights in order to minimize the cost function (loss) and optimizer update at the end of it.
I set a chainer variable f which is basically the features array and I pass it to neural network in order to estimate the rewards (similarly to maximum entropy).Finally , I have to compute the gradients in a way that is explicitly described in the aforementioned paper.The variable L_th_grad needed typecast in order for the program to work(transform from float64 to 32). I observed that rewards were produced with opposite values so I just negated gradient to balance them.
Need to have chainer (pip install chainer or pip3 -//-).
Regarding the results , they seem optimal and much more improved than the maximum entropy algorithm . The program manages to produce reward maps for even more complex reward maps , such as 25x25 , which maximum entropy one was not able to complete that task properly.
The likelihood function (exp_svf) should have a regularization term but weight decay seems to have the same effect.







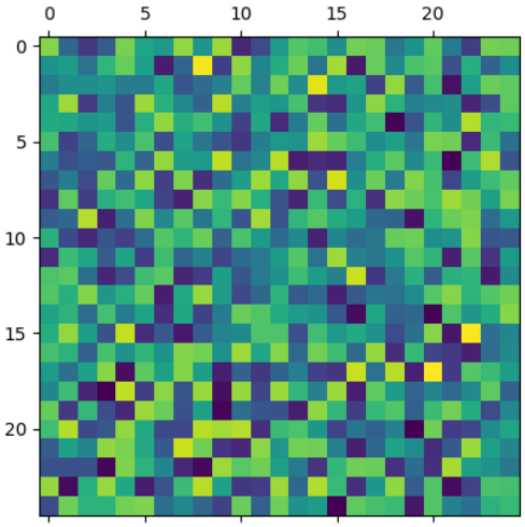
Maximum entropy with Deep Neural Network produces the most optimal results .It is the algorithm that can show the goal states in more complex maps.

Experiments - Results

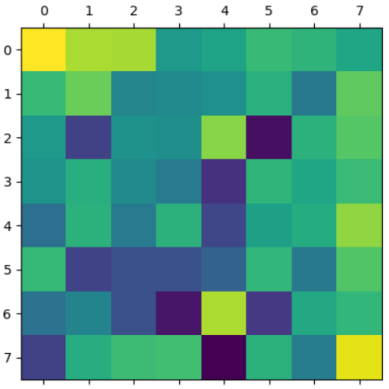
Running time was about 5 minutes for maximum entropy with bigger than 25x25 grid and about 50 min for 25x25 grid.20 min elapsed while the program max entropy deep irl was running on 15x15 grid.

Some examples with max_entropy_irl (25x25). As you see it struggles to work with more complex maps. But , doubling the size is not enough to ruin the results.It is obvious that max entropy with an 8x8 grid is efficient.

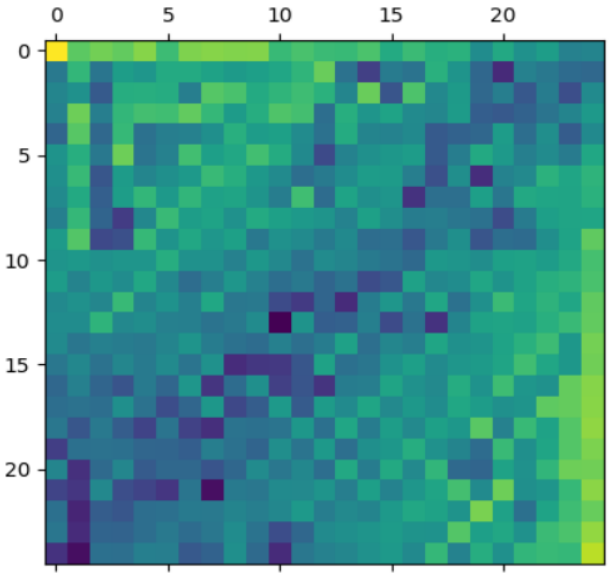
Max entropy IRL (25x25)



Max entropy IRL (8x8)

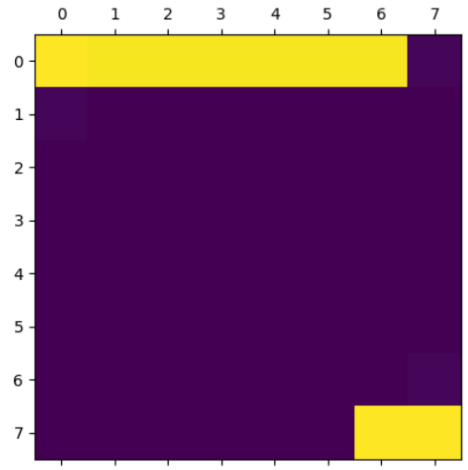


Max entropy Deep IRL (25x25) Com: Goal states correct! Better accuracy than maximum entropy reward map

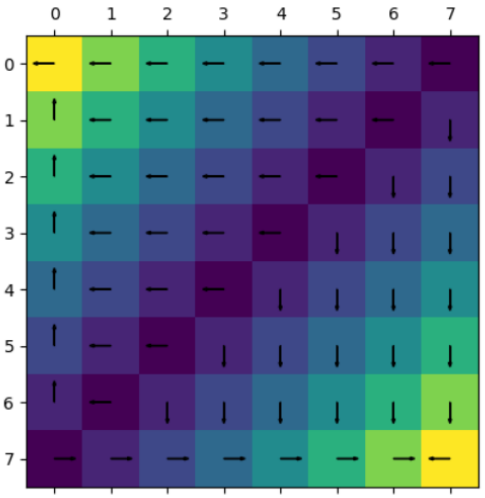


Another examples with similar results as the default dimension (4x4)

Linear programming (8x8)



Value Iteration (8x8)



Overall , my work contributed to create a better understanding about the concepts of inverse approach in reinforcement learning as well as implementing this knowledge in an interactive environment such as gym. By converting the theory from those papers to code, it became more clearer to me how inverse reinforcement learning operates to estimate rewards. Concluding my work , I am satisfied that my programs managed to produce an optimized reward map and I am sure that these algorithms can be implemented to a wide variety of problems.

—---------- Aris Tsilifonis 4/2/2023 —----------