EECS 4080: Investigating the Safety of Agents with Machine Learning Components

Part 2: Requirements and Design of test bed

Jonathan Azpur

Supervisor: Dr. Yves Lespérance

1. Introduction

The increasing use of machine learning techniques to build agents has given rise to widespread concerns over whether such systems are safe to deploy. We decided to investigate the research on safety in machine learning and develop a simulation of a machine learning agent with safety strategies. For this part of the project we are looking to test and implement a strategy for safety in machine learning agents. In this project we will build two set of problems in which an agent has to achieve some goal **G**. The implementation of the project will support a machine learning agent in the form of reinforcement learning that will evaluate its environment, and produce optimal strategies to achieve its goal state from any possible initial state in it's environment. In order to test safety strategies for this agent we will introduce danger situations into a non-deterministic environment and our algorithms/agent will have to work around these circumstances to obtain not only an optimal strategy but also a safe strategy. The project has been developed in Python.

2. Domain & Problem definition

The domain of this test bed will involve a 5x4 grid-world and a planetary robot (machine learning agent) capable of exploring each grid. The grid-world will have white grids, which will represent safe spaces that our agent can explore, red grids, which will represent craters (unsafe spaces) in which the agent can fall, a yellow or green grid that will represent the agent's goal state, and a purple grid that will represent the initial state for our agent. The grid-world will be limited by four borders. In both domains our agent will be represented as a blue square and at any time, the agent will be able to move from one grid to another by performing 1 out of 4 possible actions (Up, Down, Left and Right). Once our agent has chosen to execute an action, it will transition to the new expected grid.

We didn't want to limit our q-learning and safety algorithms to be limited to a single problem set, so our algorithm is able to adapt to multiple problems. As long as the problem trying to be solved is performed in an environment with a set of states (including an initial state and a goal state), a set of actions, a set of rewards and a transition function our agent will be able to find an optimal policy. In order to showcase the modularity of our algorithms we will be evaluating our agent in two different problems.

We assume that we will have access to a set of states and actions, and a transition matrix for each domain.

2.1. Navigation Domain

Our planetary robot will be located in a starting point (initial state). The agent's goal will be to move around the grid world and reach a specified target destination (a goal state). The idea is for the agent to find the most efficient path to it's goal state while remaining safe. In Figure 1. we have a visual of the first set of problem, with a planetary robot an initial state, a set of crates and a goal state (green).

The PEAS of our agent in the Navigation Domain

- Performance measure: Safety violation of an agent and the discounted accumulated reward in each state.
- Environment: Grid world with an initial state, a goal state and craters.
- Actuators: move forwards, backwards, to the right and to the left.
- Sensors: Ability of agent to inspect its current and adjacent states.

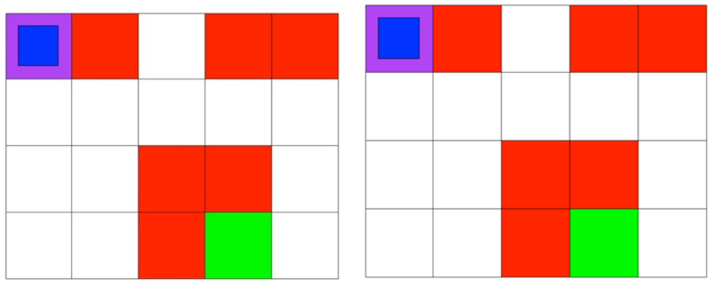
Figure 1. Visual of the first set of problems

2.2. Mineral Collection Domain

Our agent will start at a be located in a starting point (initial state referred to as the BASE and the agent's goal is to find a mineral located in a goal state, and bring is back to BASE. The idea is for the agent to find the most efficient path to from the BASE to the mineral ore and back while remaining safe. In Figure 2 we have a visual of the second set of problem, with the planetary robot, an initial state, a set of crates and a goal state (yellow mineral). The PEAS of our agent in Mineral Collection Domain:

- Performance measure: Safety violations of an Agent and the discounted accumulated reward in each state.
- Environment: Grid world with an initial state, a goal state (mineral) and craters.
- Actuators: move forwards, backwards, to the right, to the left or pick up a mineral.
- Sensors: Ability of agent to inspect its current and adjacent states.

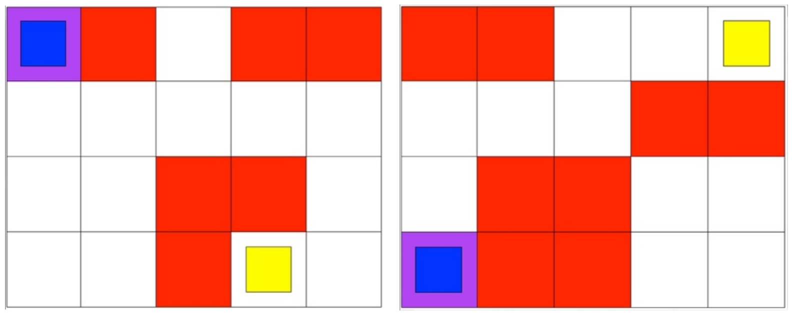
Figure 2. Visual of the second set of problems

3. Test Bed Architecture

The test bed will be divided in five main stages. First, the program will ask for the data necessary to construct the environment; number of states (specify the initial and the goal states), number of actions, rewards and a transition matrix. Once a user inputs the necessary data we will go through the following procedure in order to obtain the safest/optimum solution and execute it on a visual interface:

1. Input environment specification
2. Build the **Environment** from input data
3. Run **Q-learning algorithm** to obtain Q (s, a) for all possible state-action pairs
4. Build **Agent** based on Q-table and Environment
5. **SIMULATE** optimal policy obtained from Q-matrix from the environment.
6. **Collect Data on safety violations**.

3.1. Environment Specification

In order to run our agent in the specified domains we will first need the specifications for the environment we will be working with. The data can be inputted interactively or from a text file. So we will need the following data from the user in order to construct the environment:

1. **n**: Number of states, then the set of states to be considered will be [0, n-1]
2. **m**: Number of action, then the set of actins to be considered will be [0, m-1]
3. **flag**: {0,1} s.t. 1 if the environment is non-deterministic and 0 if it's deterministic
4. $\boldsymbol{s_0}$: Initial state of our agent
5. $\boldsymbol{s_{goal}}$: Goal state of out agent
6. $[(\boldsymbol{s_1},\boldsymbol{r_1}),\dots,(\boldsymbol{s_d},\boldsymbol{r_d})]$: Set of state reward pairs, which indicated what reward will an agent obtain if it where to land in state s_i
7. $[\boldsymbol{s_1},\dots,\boldsymbol{s_k}]$: Set of unsafe states.
8. **i**: Number of iterations for Q-learning algorithm convergence.
9. **T**: $n\times m$, **Transition Matrix** s.t. for all states s and action a, $T[s][a]$ will indicate what state should an agent go to if it's currently in state s and performs action a. **If the environment is non-deterministic this will be left empty.**
10. $\boldsymbol{P_a(s_a|s)}$: Probability distribution s.t. for all state-action pairs (s,a), P_a will return the probability of ending in a successor state $\boldsymbol{s_a}$ after performing action a on state s. **If the environment is deterministic this will be left empty.**

3.2. Environment

Environment (environment.py) is a class that will represent the domain in which our intelligent agent will act upon. We want to model the environment as a Markov Decision Process (MDP) so this class will have the following components:
We wanted to model the environment as a Markov Decision Process (MDP):

- States, $\boldsymbol{S}$: set of states s.t every s ∈ S is a state that can be explored in this environment.
- Actions, $\boldsymbol{A}$: set of actions s.t every a ∈ A is an action that an agent can perform in this environment.
- Transition Function, $\boldsymbol{tf}: \boldsymbol{S}, \boldsymbol{A}, \boldsymbol{S} \rightarrow \boldsymbol{p}$ s.t. for each tuple (a, s_0, s_a) if environment is **DETERMINISTIC** tf will output 1 for a state s_a if and only if T[s_0][a] = s_a, if not it will out out 0. But, if the environment is **NON-DETERMINISTIC** tf will return a probability p based on the Probability distribution $P_a(s_a|s_0)$.
- Reward Function, $\boldsymbol{r}: \boldsymbol{S}, \boldsymbol{A}, \boldsymbol{S} \rightarrow \boldsymbol{R}$ s.t. for each possible input (s_0, a, s_a), rf will output the reward r ∈ ℝ an agent would obtain by transitioning from state s_0 to s_a.
- Safety Boolean Function, $\boldsymbol{sb}: \boldsymbol{S}, \boldsymbol{A} \rightarrow \{\boldsymbol{True}, \boldsymbol{False}\}$ s.t. for each pair of state/action inputs (s,a), sb will output $True$ if the state obtained by applying a to s is safe and it will return $False$ if it is not safe.

3.3. Q-Learning Specification

The task is for the agent to learn a policy $\pi^*(s)$ that selects the optimal action a based on its current state s, for all states in the environment. This is done by choosing in each state the action that maximizes the sum of the immediate reward $r(s,a)$ plus the discounted $(0 < \gamma < 1)$ cumulative reward value $V^\pi(s_{i+1})$ of it's immediate successor state T[s][a] = s_{i+1}. The cumulative value $V^\pi(s_{i+1})$ is achieved by following a set of random actions a_i from an random state s_i, $T[s_i][a_i] = s_{i+1}$.

$$V^\pi(s_t) = r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma r(s_{t+2}, a_{t+2}) + \cdots$$

Now we can provide a formal definition for the optimal policy:

$$\pi^*(s) \equiv argmax_a \, [r(s_i, a) + V^\pi(s_i)], (\forall s)$$

The problem is, V^π can only be learnt if the agent is able to predict the reward of future states and the transition matrix *T*, which it's not possible in most applications. So instead, we will be using **Q-learning algorithm** to generate a value $Q\ (s,a)$ for each state-action pair in the environment. The value of $Q\ (s,a)$ is the maximum discounted cumulative reward that can be achieved by starting in state s and applying action a. Now we can estimate the cumulative reward value without having to predict future rewards or the transitions. In a deterministic environment the value of $Q\ (s,a)$ is recursively expressed as:

$$Q\ (s,a) = \mathrm{r(s,a)} + \gamma max_{a'} Q(s,a')$$

In a non-deterministic environment, the value of $Q\ (s,a)$ is recursively expressed as:

$$Q\ (s,a) = \mathrm{E[r(s,a)]} + \gamma \sum P_a(s'|s) \cdot max_{a'} Q(s',a')$$

Once we have the maximum discounted cumulative reward $Q\ (s,a)$ for each possible state-action pair, we will store them in our Q matrix (i.e. Q[s][a] = $Q\ (s,a)$). Now, the optimal policy can be calculated as:

$$\pi^*(s) \equiv argmax_a Q[s][a]$$

The optimal policy will be used to determine what action should an agent take next at any given state the agent is currently on.

<u>Q learning algorithm:</u>

```
For each $s,a$ initialize the table entry Q[s][a] to zero.
Observe the current state s
Perform until convergence ($\varepsilon$):
   -  Select an action $a$ and execute it
   -  Receive immediate reward r
   -  Observe the new state s'
   -  Update the table entry for Q[s][a] as follows:
        Deterministic: Q[s][a] ⇐ r + $\gamma \cdot max_a Q[s][a]$
        Non-deterministic: Q[s][a] ⇐ E[r] + $\gamma \sum P_a(s'|s) \cdot max_{a'} Q(s',a')$
```

3.4. Intelligent Agent Specification

The intelligent agent will be specified as a class (i.e. i-agent.py), to build a complete agent we need the environment it will run on and a matrix (obtained from Q-learning) that will represent the agent's knowledge on which action will provide with the maximum cumulative rewards in each state. This class will have the following components:

- State, **s**: the current state the agent finds himself in.
- Environment, **environment class $\boldsymbol{e}$**, such that e will be the specific environment the agent is acting on. This component will serve as a sensor to retrieve data from it's environment.
- Knowledge Base: **matrix $\boldsymbol{Q}$** s.t. for each pair of state-action inputs (s, a), $Q[s][a]$ is equivalent to the $Q(s, a)$ obtained from a Q-learning algorithm.
- Transition Function, $\boldsymbol{tf}: \boldsymbol{S}, \boldsymbol{A}, \boldsymbol{S} \rightarrow \boldsymbol{p}$ s.t. for each tuple (a, s_0, s_a) if environment is **DETERMINISTIC** tf will output 1 for a state s_a if and only if T[s_0][a] = s_a, if not it will out out 0. But, if the environment is **NON-DETERMINISTIC** tf will return a probability p based on the Probability distribution $P_a(s_a|s_0)$.
- Agent Function, $\boldsymbol{af}: \boldsymbol{S} \rightarrow \boldsymbol{A}$ s.t. for each input state s, af will output the action a that provides the maximum reward based on matrix Q.
- Actuator: $\boldsymbol{act}: \boldsymbol{A} \rightarrow \boldsymbol{S}$, this function will update the agent's current state based on the action chosen to perform.

This will be the architecture of our agent:

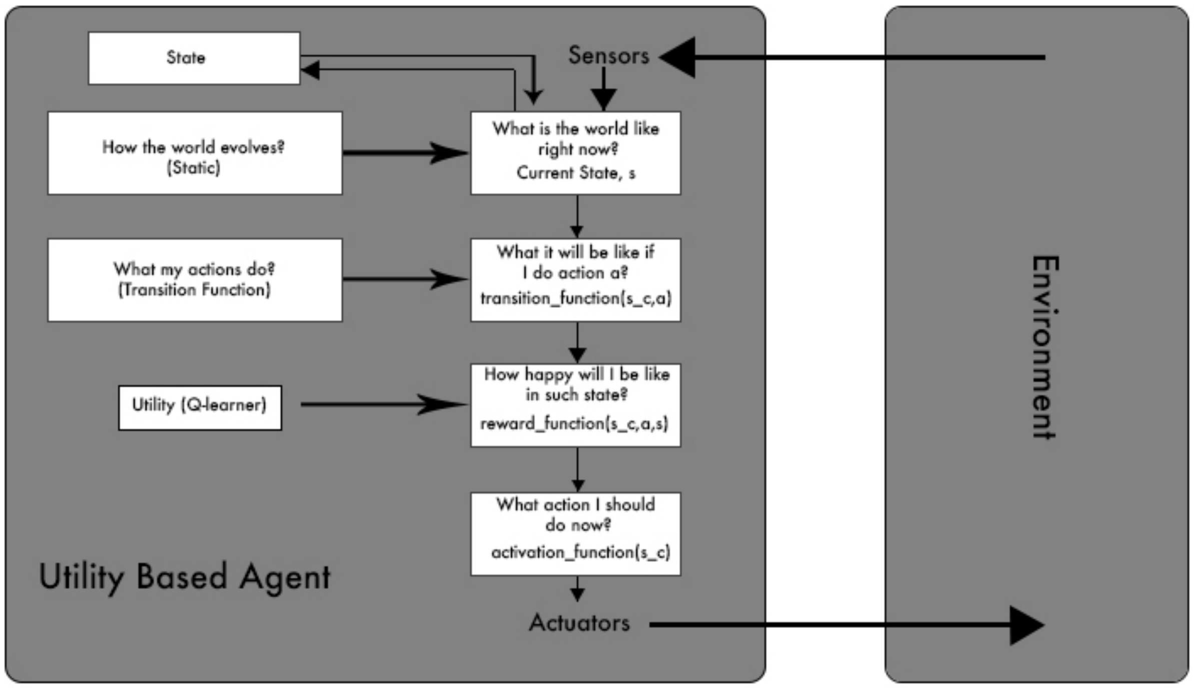

Figure 3. Architecture of the Intelligent agent, this is note specific to these domains, it can be adapted to multiple problems (Russell & Norvig AI: A Modern Approach Ch 2 slides at goo.gl/EHNsnU pp.25-6)

4. Safety

During each execution we will keep track of the amount of safety violations, in general we define a safety violation as any instance in which the intelligent agent transitions into on of the unsafe states $[\boldsymbol{s_1}, \ldots, \boldsymbol{s_k}]$. In our simulation these unsafe states will be represented as craters with a negative reward associated to them. The grid map in which the planetary robot will be operating will have a certain number of craters, the safety concern will be to make sure the robot avoids falling into the crates while searching for it's goal. The crate will be represented as a dangerous situation by providing a negative reward (punishment) if the robot were to move in it's direction. For other domains, where we do not have crater, the algorithm will adapt to any environment as long as there is a negative reward value (punishment) to any state that challenges the safety of the agent.

Given that the agent can interact with non-deterministic environments, we will have to take into consideration the probability distribution $\boldsymbol{P_a}(\boldsymbol{s_a}|\boldsymbol{s_0})$ when performing each action. $\boldsymbol{P_a}(\boldsymbol{s_a}|\boldsymbol{s_0})$ may cause the agent to transition into an unsafe state (i.e. crater), if this happens we will have to keep track of this instance in order to output at the end of our simulation the safety violation data. For example, in our simulations every time the agent chooses to perform an action there is a 15% chance that the agent isn't able to transition into the desired state and instead transition to a different adjacent state, if the agent were to be next to a crater there is the possibility of our agent falling into a crater even though it wasn't the action it intended to perform. We will keep track of a safety counter to determine the amount of times the agent has been in an unsafe position.

5. Sample Output

Here are the results of our intelligent performance on both domains:

NAVIGATION **DOMAIN:**

```
from (0,0) -> move DOWN to: (0,1)
from (0,1) -> move RIGHT to: (1,1)
from (1,1) -> move RIGHT to: (2,1)
from (2,1) -> move RIGHT to: (3,1)
from (3,1) -> move RIGHT to: (4,1)
from (4,1) -> move DOWN to: (4,2)
from (4,2) -> move DOWN to: (4,3)
from (4,3) -> move LEFT to: (3,3)
You have found the minerals!
CONGRATULATIONS!


SAFETY DATA: Your Rover has fallen 1
time/s.
```

MINERAL COLLECTION **DOMAIN**

```
from (0,3) -> move UP to: (0,2)
from (0,2) -> move UP to: (0,1)
from (0,1) -> move RIGHT to: (1,1)
from (1,1) -> move RIGHT to: (2,1)
from (2,1) -> move UP to: (2,0)
from (2,0) -> move RIGHT to: (2,0)
from (3,0) -> move RIGHT to: (4,0)
You have found the minerals!
CONGRATULATIONS!
from (4,0) -> move LEFT to: (3,0)
from (3,0) -> move LEFT to: (2,0)
from (2,0) -> move DOWN to: (2,1)
from (2,1) -> move LEFT to: (1,1)
Something happened, your Rover wasn't able
to go Left, instead it went (Up)
from (1,1) -> move UP to: (1,0)
ATTN: Your Rover has FALLEN, CAREFUL!!! GET
OUT!!!
from (1,0) -> move DOWN to: (1,1)
from (1,1) -> move LEFT to: (0,1)
from (0,1) -> move DOWN to: (0,2)
from (0,2) -> move DOWN to: (0,3)
You are back in BASE with the MINERALS,
CONGRATULATIONS!

SAFETY DATA: Your Rover has fallen 1 time/s.
```

A. Appendix

A.1. Instructions for simulations

There are two programs that you can execute:

1. **grid-world.py**, which simulates the Navigation Domain
2. **mineral-world.py,** which simulates the Navigation Domain

In order to execute the program, you can run any of the following commands:

```
python3 grid-world.py < input.txt
python3 grid-world.py < input-2.txt
python3 grid-world.py < input-3.txt
```

or

```
python3 mineral-world.py < input.txt
python3 mineral-world.py < input-2.txt
python3 mineral-world.py < input-3.txt
```

These are the requirements you need installed in order to execute simulation

1. Python 3.6
2. python.Numpy 1.11.3

A.2. Pseudo-Code for Navigation Domain

```
Input:
s_0: initial state for our agent.
S: set of states of size n
A: set of actions of size m
R: set of state-reward pairs
T: transition matrix of size n×m s.t. T[s_0][a] stores the state s_1 that an agent would
arrive to if it were to perform action a on s_0
Algorithm:
Build Environment based on Input
# will consider non-deterministic environment when running Q-learning algorithm
Run Q-learning algorithm on environment and obtain Q(s,a) for each pair state-action.
Build Agent ag based on Input, Environment and Q-learning algorithm
list_actions <= empty
While agent is not in goal state then i=0…N
   Action_to_perform = ag.next_action()
   # next_action() is the agent's agent_function based on the Q(s,a) obtained during Q
   learning algorithm
   Action_to_perform = non_determinisic_transition(Action_to_perform,other_actions)
   # safety
   If ag.not_safe_state(Action_to_perform) then
      ag.safety_count <= ag.safety_count + 1
      list_actions.append(Action_to_perform)
      Action_to_perform <= ag.exit_unsafe_state()
   list_actions.append(Action_to_perform)
end while loop
# Show path obtained in Interface simulation
For action in list_actions
   ag.perform_action(action)
end
Output actions to go from initial state to goal state
Ouput ag.safety_count
```

A.3. Pseudo-Code for Mineral Collection Domain

```
Input:
s_0: initial state for our agent.
S: set of states of size n
A: set of actions of size m
R: set of state-reward pairs
T: transition matrix of size n×m s.t. T[s_0][a] stores the state s_1 that an agent would
arrive to if it were to perform action a on s_0

Algorithm:
Build Environment based on Input
# will consider non-deterministic environment when running Q-learning algorithm
Run Q-learning algorithm on environment and obtain Q(s,a) for each pair state-action.
Build Agent ag based on Input, Environment and Q-learning algorithm
list_actions <= empty
While not found MINERAL then
   Action_to_perform = ag.next_action()
   # next_action() is the agent's agent_function based on the Q(s,a) obtained during Q
   learning algorithm
   Action_to_perform = non_determinisic_transition(Action_to_perform,other_actions)
   # safety
   If ag.not_safe_state(Action_to_perform) then
      ag.safety_count <= ag.safety_count + 1
      list_actions.append(Action_to_perform)
      Action_to_perform <= ag.exit_unsafe_state()
    list_actions.append(Action_to_perform)
end while loop
While agent is not in BASE then
   Action_to_perform = ag.next_action()
   # next_action() is the agent's agent_function based on the Q(s,a) obtained during Q
   learning algorithm
   Action_to_perform = non_determinisic_transition(Action_to_perform,other_actions)
   # safety
   If ag.not_safe_state(Action_to_perform) then
      ag.safety_count <= ag.safety_count + 1
      list_actions.append(Action_to_perform)
      Action_to_perform <= ag.exit_unsafe_state()
    list_actions.append(Action_to_perform)
end while loop
# Show path obtained in Interface simulation
For action in list_actions
    ag.perform_action(action)
end
Output actions to go from initial state to goal state
Ouput ag.safety_count
```