2_RL

April 20, 2022

1 Network Analysis Using Reinforcement Learning

1.1 Spring 2022: Shaoyu Pei, Avery Peiffer

1.2 Advisor: Dr. Mai Abdelhakim

1.3 Notebook 2: Applying Reinforcement Learning to the Problem

If you have not already read through and tested out the network setup, please take some time to do so. It will provide you with the necessary background to understand the problem.

The purpose of this notebook is to implement a reinforcement learning algorithm using our network environment code. This allows us to train reinforcement learning algorithms. We modified Laura D'Arcy's GraphRLnx repository to create this class.

Please refer to the `tutorials/` folder for an overview of reinforcement learning and Q-learning. Essentially, we wish for our algorithm to learn the most trustworthy path(s) from the source to the destination. A trustworthy path is one that is unlikely to be corrupted from past examples. We start by sending packets through the network randomly; we then use a function to see if the transmission was 'corrupted' in the way that we have previously defined it. We also take into consideration the number of hops the path takes. We adjust our incentive structure accordingly and, over time, the algorithm will learn what paths are optimal.

Our model makes the assumption that there is one reinforcement learning controller that is overseeing the entire process. Additionally, we wish to acknowledge the action and state spaces that make up our reinforcement learning space. The state space represents the possible list of states that the algorithm can take in a given time step. In this case, it is defined as the possible nodes that can be visited along a path from the source to destination node. Likewise, the action space represents the possible actions that can be taken from the given state. For this model, the action space is defined as the nodes that are neighboring the current node and that have not already been visited along the current path (this helps avoid cycles and infinite loops).

Note: If you get any errors due to packages not being installed, please add them to the cell below. We might have not encountered these errors due to already having those packages installed on our computers for other courses/projects/etc.

[1]:
```
!pip3 install gym
```

```
Requirement already satisfied: gym in c:\users\avery
peiffer\anaconda3_new\lib\site-packages (0.21.0)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\users\avery
peiffer\anaconda3_new\lib\site-packages (from gym) (1.5.0)
```

```
Requirement already satisfied: numpy>=1.18.0 in c:\users\avery
peiffer\anaconda3_new\lib\site-packages (from gym) (1.18.5)
```

[2]:
```
import gym
import math
import random
import scipy.signal
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

from gym import spaces
```

Here we incorporate everything from the first notebook into a `graphRL` class. This class uses the `gym` environment template, which allows for training of RL algorithms.

[3]:
```
class graphRL(gym.Env):

    # Add random edges to the graph until it is a complete graph (more detail␣
↪in previous notebook)
    def random_edge(self):
        edges = list(self.graph.edges)
        nonedges = list(nx.non_edges(self.graph))
        if len(edges) > 0:
            chosen_edge = random.choice(edges)
            chosen_nonedge = random.choice([x for x in nonedges if␣
↪chosen_edge[0] == x[0] or chosen_edge[0] == x[1]])
        else:
            chosen_nonedge = random.choice(nonedges)
        self.graph.add_edge(chosen_nonedge[0], chosen_nonedge[1])

    # Determine if a path is corrupted based on the attack probabilities of the␣
↪nodes that make it up
    # (more detail in previous notebook)
    def is_corrupted(self, path, verbose=False):
        for node in path:
            attack_prob = self.devices[node].attack_prob
            attacked = random.uniform(0,1) < attack_prob
            if attacked:
                if verbose:
                    print('Node = ', node)
                return True
        return False

    def __init__(self, fname=None, network_size=10, edge_prob=1, percent_mal=0,␣
↪attack_probs=[0, 0]):
        self.devices = []
        self.mal_nodes = []

        class device:
            pass

        # Read in a custom environment from a file and create the graph that way
        if fname != None:
            with open(fname) as f:
                lines = [line.rstrip() for line in f]

                self.network_size = int(lines[0])
                self.src = 0
                self.dst = self.network_size - 1

                edges = [eval(x) for x in lines[1:-1]]
                self.graph = nx.Graph()
                self.graph.add_nodes_from(list(range(self.network_size)))
                self.graph.add_edges_from(edges)

                attack_probs = [float(x) for x in lines[-1].split(',')]

                nodes = list(range(0, self.network_size))

                for node, prob in zip(nodes, attack_probs):
                    a = device()
                    a.node = node
                    a.attack_prob = prob

                    if prob > 0:
                        self.mal_nodes.append(node)
                        a.mal = True

                    else:
                        a.mal = False

                    self.devices.append(a)

        # Create a random graph based on a set of parameters
        else:
            self.network_size = network_size
            self.src = 0
            self.dst = network_size - 1

            self.graph = nx.gnp_random_graph(network_size, edge_prob)
            while not nx.is_connected(self.graph):
                self.random_edge()
            num_mal = network_size * percent_mal

            while num_mal > 0:
                rand = np.random.randint(0, network_size)
                if rand != src and rand != dst and rand not in self.mal_nodes:
                    self.mal_nodes.append(rand)
                    num_mal -= 1

            nodes = list(range(0, network_size))

            for node in nodes:
                a = device()
                a.node = node

                if node in self.mal_nodes:
                    a.mal = True
                    a.attack_prob = np.random.uniform(attack_probs[0],␣
↪attack_probs[1])
                else:
                    a.mal = False
                    a.attack_prob = 0

                self.devices.append(a)

        self.num_actions = self.network_size
        self.num_states  = self.network_size

        self.action_space = spaces.Discrete(self.num_actions)
        self.observation_space = spaces.Discrete(self.num_states)

    # Draw the graph
    def render(self):
        nx.draw(self.graph, with_labels=True)
        plt.show()
```

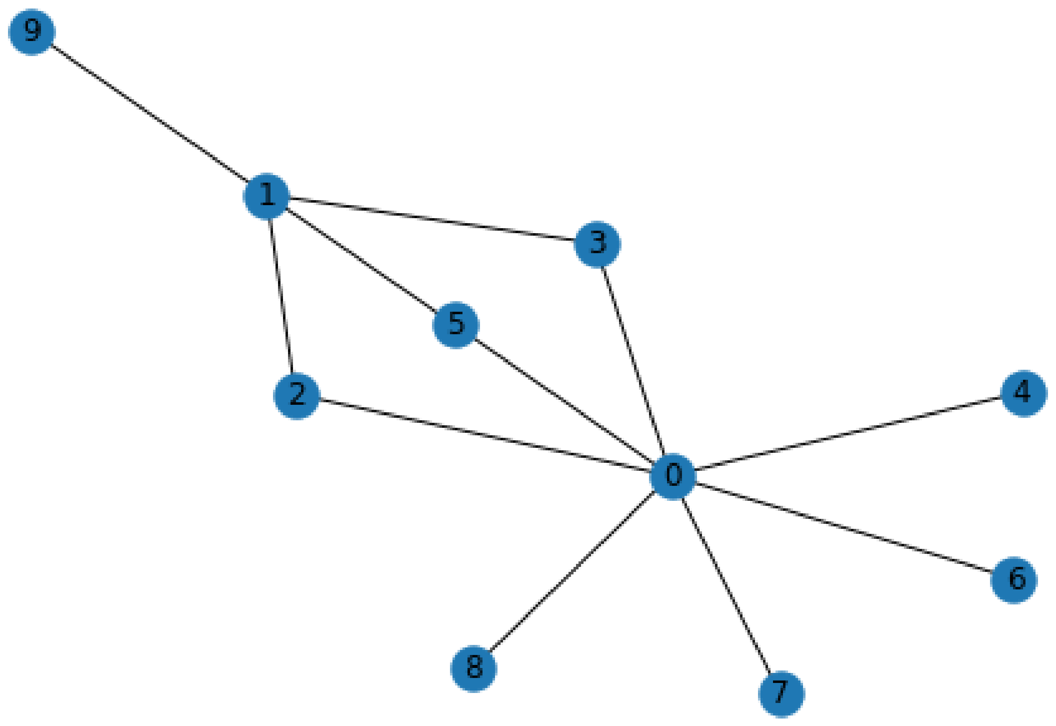
We create an instance of the graphRL class and render it. As a reminder of our convention, the source node is always Node 0 and the destination node is the highest numbered node. Additionally, we print out all of the nodes and whether they are malicious. This will help us confirm that our algorithm is learning to avoid the malicious nodes. By visual inspection, we can also confirm that our algorithm is learning the shortest paths from source to destination.

[4]:
```
# env = graphRL(network_size=10, edge_prob=0.1, percent_mal=0.4,␣
↪attack_probs=[0.3, 0.8])
env = graphRL('./env/env1.txt')

for a in env.devices:
    if a.mal:
        print(f'Node {a.node} IS malicious. Attack probability is {a.
↪attack_prob:.2f}.')
    else:
        print(f'Node {a.node} is NOT malicious.')

env.render()
```

```
Node 0 is NOT malicious.
Node 1 is NOT malicious.
Node 2 is NOT malicious.
Node 3 IS malicious. Attack probability is 0.64.
Node 4 is NOT malicious.
Node 5 IS malicious. Attack probability is 0.85.
Node 6 is NOT malicious.
Node 7 is NOT malicious.
Node 8 is NOT malicious.
Node 9 is NOT malicious.
```



Some hyperparameters we introduce when we move to a RL environment: * `corrupted_path_penalty` refers to the penalty our algorithm incurs when it takes a path that is corrupted. * `dead_end_penalty` refers to the penalty our algorithm incurs when it encounters a dead-end node. We don't need to consider this node for future pathfinding, so we put an enormous penalty on it. * `training_iterations` refers to the number of times we will try to find a path from the source to the destination. Too few iterations will lead to underfitting; too many will lead to overfitting.

The other hyperparameters are used for the Q-learning algorithm (definitions used from the RL tutorial source): * `alpha` is the learning rate ($0 < \alpha <= 1$). This captures the extent to which the Q-values are updated from one training iteration. * `gamma` is the discount factor ($0 < \gamma <= 1$). γ determines how much importance we want to give to future rewards. A lower γ makes our algorithm consider immediate rewards more, while a higher γ captures a long-term effective reward. * `epsilon` is the degree ($0 < \epsilon <= 1$) to which we want to explore the action space (i.e. pick a random value) or exploit our learned Q-values. A lower ϵ will result in more penalties during training because we are more often picking new, unexplored paths. A higher ϵ will result in fewer penalties during training but may lead to overfitting.

[5]:
```
q_table = np.zeros([env.num_states, env.num_actions])

training_iterations = 10000
corrupted_path_penalty = 10
clean_path_reward = corrupted_path_penalty / 10 # The clean path reward should␣
 ↪probably not be equal to the corrupted path penalty
dead_end_penalty = 1000 # Continue working on this
# Node: don't want this to be np.inf because it will fail in some edge cases␣
 ↪(since invalid neighbors are np.inf)

alpha = 0.1
gamma = 0.6
epsilon = 0.1
```

Some notes about this cell, because it is the most complicated cell in the notebook: * We are trying to find paths from the source to the destination, and we go one node at a time. Think of the current state as the node we are currently at. The possible actions we can choose from are the other nodes that we can visit from this node (so, its neighbors). However, this is at odds with how we define the Q-table, where all actions are technically "valid" from each state. For this reason, we keep a `valid_neighbors` array to reference when we select an action. If a node is a dead end, it will have no valid neighbors and we can stop the training iteration. * When we are finding paths from the source to the destination, we don't want to visit the same node twice (this is called a cycle and can easily lead to infinite loops). We use the `VISITED` flag and `visit_arr` to mark whether we've visited a node in the current pathfinding iteration. This also helps us when we look at the `valid_neighbors` of a node. * As stated above, `epsilon` represents the degree to which we randomly explore the action space. We want to decrease the value of `epsilon` as we train our algorithm to limit the amount that we explore the action space, because it becomes redundant. This is the purpose of the `epsilon_decay` function. We have defined it such that it linearly decreases from the initial value of `epsilon` to 0 over the training iterations, but alternative definitions could be more effective.

[6]:
```
VISITED = -1

# For plotting metrics
```

```python
all_epochs = []
all_penalties = []

# Linear function to represent how epsilon should decrease as we go through␣
↪training iterations
def epsilon_decay(x):
    return epsilon - epsilon*x/training_iterations

# Number of training iterations
for i in range(1, training_iterations+1):
    total_penalty = 0
    path = []
    eps = epsilon_decay(i) * epsilon
    state = env.src

    done = False

    visit_arr = np.zeros(env.network_size)

    while not done:
        path.append(state)
        dead_end = True
        visit_arr[state] = VISITED # Shows that we have visited a node

        valid_neighbors = list(env.graph.neighbors(state)) # These are the␣
↪valid possible actions

        # Check to see if the node is a dead end (if there are any valid␣
↪neighbors, they have been visited already)
        for neighbor in valid_neighbors:
            if visit_arr[neighbor] != VISITED:
                dead_end = False

        # This node is a dead end, so we put a giant penalty on it so we don't␣
↪go to it again in future iterations
        if dead_end:
            q_table[:, state] = -dead_end_penalty
            done = True

        # The node is not a dead end, so we continue trying to find paths to␣
↪the destination
        else:
            # Explore the action space by picking a random action
            if random.uniform(0, 1) < eps:
                action = env.action_space.sample()
                # Make sure the action is valid by referencing the valid␣
↪neighbors and visited arrays
                while action not in valid_neighbors or visit_arr[action] ==␣
↪VISITED:
                    action = env.action_space.sample()

            # Exploit learned values by selecting the best action from the␣
↪current state based on our Q-table
            else:
                slc = q_table[state]
                action = np.argmax(slc)

                # Make sure the action is valid by referencing the valid␣
↪neighbors and visited arrays
                while action not in valid_neighbors or visit_arr[action] ==␣
↪VISITED:
                    slc[action] = -np.Inf
                    action = np.argmax(slc)

            # Now, we've finally selected an action.
            # We can take the step and update the Q-table according to the␣
↪algorithm.
            next_state = action
            state = next_state

            if state == env.dst:
                path.append(state)
                done = True

            total_penalty += 1

    # Print out the path we took.
    # If we got to the destination, determine if that path was corrupted.␣
↪Update the reward accordingly
    # if dead_end == True:
        # print(f'Path taken = {path}. Encountered a dead end.')
    else:
        # print(f'Path taken = {path}.')

        if env.is_corrupted(path):
            # The path is corrupted, but our algorithm doesn't know what node␣
↪is the cause.
            # So, we penalize all of the nodes. Over time, the nodes that are␣
↪actually corrupted will be
            # penalized more often than the nodes that aren't.
            total_penalty += corrupted_path_penalty
            # print(f'This path was corrupted.')

        else:
            total_penalty -= clean_path_reward

        # The path[1:len(path)-2] is a way of not penalizing the source or␣
↪destination nodes, since we
        # assume that those two can't be corrupted.
        for node in path[1:len(path)-2]:
            q_table[:, node] -= total_penalty / len(path)

    all_epochs.append(i)
    all_penalties.append(total_penalty)
```

Now that the training is finished, we can print out the Q table and try to visually confirm that our algorithm has learned the optimal paths. The rows represent the current state, and the columns represent the possible actions from that state. We should see higher values for the actions that lead to clean, shorter paths. Likewise, we should see lower values for actions that result in a corrupted path. Additionally, we should see `-np.inf` for actions that aren't possible from the current state (not neighboring), and `-dead_end_penalty` for actions that result in a dead end.

[7]:
```
print(pd.DataFrame(q_table))
```

```
     0    1       2       3       4       5       6       7       8    9
0 -inf -inf -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0 -inf
1 -inf -inf    -inf -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
2 -inf  0.0 -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
3 -inf  0.0 -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
4  0.0  0.0 -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
5 -inf  0.0 -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
6  0.0  0.0 -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
7  0.0  0.0 -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
8  0.0  0.0 -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
9  0.0  0.0 -1005.0 -1021.5 -1000.0 -1020.0 -1000.0 -1000.0 -1000.0  0.0
```

We can also do a more formal verification of the results. This code uses some of our training code to rerun the algorithm and find what it has determined to be the most efficient, safest paths through the graph.

[8]:
```
total_hops, total_penalties = 0, 0
episodes = 100

for _ in range(episodes):
    path = []
    hops, penalties = 0, 0
    state = env.src
    visit_arr = np.zeros(env.network_size)
```

```python
    done = False

    while not done:
        hops += 1
        path.append(state)
        dead_end = True
        visit_arr[state] = VISITED

        valid_neighbors = list(env.graph.neighbors(state))

        for neighbor in valid_neighbors:
            if visit_arr[neighbor] != VISITED:
                dead_end = False

        if dead_end:
            total_penalties += 1
            done = True

        else:
            action = np.argmax(q_table[state])
            state = action

        if state == env.dst:
            done = True

    if env.is_corrupted(path):
        total_penalties += 1

    total_hops += hops

print(f"Results after {episodes} episodes:")
print(f"Average hops per episode: {total_hops / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")
```

```
Results after 100 episodes:
Average hops per episode: 2.0
Average penalties per episode: 1.0
```