ICAPS 2018 Summer School Lab 2: RL for RDDL

This lab will introduce an infrastructure for running (deep) *reinforcement learning (RL)* experiments on RDDL planning problems. The lab will focus on using Q-learning for solving problems in the Wildfire domain. However, the infrastructure is applicable to any RDDL domain and problem defined over discrete state and action variables. Most of the lab will focus on a simplified Wildfire domain to allow for faster experiment times with the end exercise considering the full domain.

Goals of the Lab:

1) Provide students with hands-on experience using the infrastructure to train and test RL agents on RDDL domains. While the lab focuses on Q-learning (with and without experience replay), the infrastructure also provides an actor-critic algorithm called A2C [1].
2) Learn about important practical choices, including how to select which RL policy to return and reward normalization.
3) Experiment with a couple of algorithm options including the neural network architecture and experience replay.

Getting Started

This lab assumes that you have already installed the Summer School Lab VM and also successfully run the test at https://bitbucket.org/eshw/rl-lab/src/master/README.md.

It is important to make sure that you have the latest version of the RL repository. We have adjusted the repository after the first announcement of the VM infrastructure. To make sure you have the latest you can run the following commands:

```
   cd /vagrant
   rm -rf RL
   git clone https://eshw@bitbucket.org/eshw/rl-lab.git RL
   pip3 install -r /vagrant/RL/requirements.txt
   cd /vagrant/RL/src/rddl_parser && make && mv rddl-parser /vagrant/RL
   cd /vagrant/RL/src/search && make
   cd /vagrant/RL/src/search/.obj && g++ -shared -o clibxx.so -fPIC *.o utils/* -lbdd -lstdc++fs
&& mv clibxx.so /vagrant/RL
   cd /vagrant
```

Part #1: Running Q-Learning for Single-Action Wildfire

We will start with a highly simplified version of the Wildfire RDDL domain. This version of the domain and problem files can be found in the directory `/vagrant/RL/env/wildfire_single_action`, where `wildfire_single_mdp.rddl` is the domain file and `wildfire_single_action_inst_mdp__1.rddl` will be the problem instance that we focus on. The simplifications in this domain compared to the original Wildfire domain are:

1) There are no ‘cutout’ actions.
2) There is a single ‘putout’ action which takes no arguments. This action puts out any fire that is currently burning on the map.

So there are two actions in this domain (‘putout’ and ‘noop’). We would like to see an RL agent learn to use the ‘putout’ action when a fire is present and ‘noop’ otherwise. Note that since the immediate reward for ‘putout’ is -10 and for ‘noop’ is 0, a greedy agent that maximizes immediate reward (by only taking ‘noop’) will do poorly. So an agent must appreciate how its actions influence future reward in order to do well.

To start the RL infrastructure we first need to start the RDDL server, which is the same server used for the International Planning Competition. Do this from a Vagrant ssh prompt via the commands.

```
cd /vagrant/RDDLSim

./run rddl.competition.Server /vagrant/RL/env/wildfire_single_action
```

This will start the server with the simplified wildfire domain and the server will now wait to serve a requested problem instance to the RL agent.

To start the RL training open another ssh terminal and execute the following.

```
cd /vagrant/RL

python3 q_learning.py --inst wildfire_single_action_inst_mdp__1 --eps 0.3 --norm_reward
--path_suffix "normalized" --train_episodes 1000 --scratch
```

This will train a basic Q-learning agent on the wildfire problem for 1000 episodes **(--train_episodes 1000)** starting from scratch **(--scratch)** using ϵ-greedy exploration with a constant $\epsilon = 0.3$ **(--eps 0.3)**. Ignore the other command-line arguments for the moment.

The default policy architecture (illustrated in Figure 1) used for training is a neural network with two fully-connected hidden layers: (defined via **class QNet**)

- Input Layer: contains the state variables, which for the above problem instance are *out-of-fuel(x,y)* and *burning(x,y)* for $x \in \{x1, x2, x3\}$ and $y \in \{y1, y2, y3\}$.
- Hidden Layers: # of hidden units in each layer is the number of state variables times **hidden_unit_factor**, where the default for hidden_unit_factor is 3

- Output Layer: # of output units for each action, which for the above problem is ‘putout’ and ‘noop’.

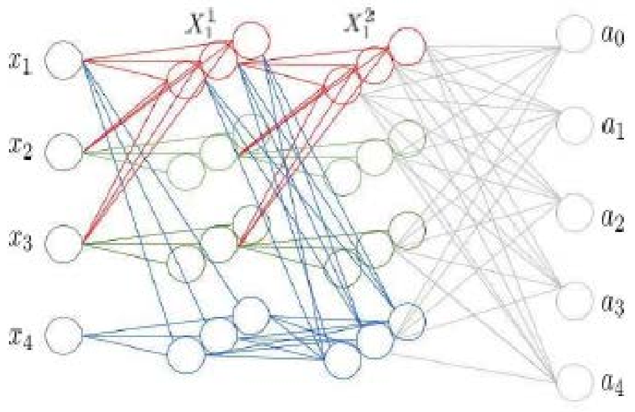

Figure 1. (Figure taken from [4]) Illustration of default neural network structure for a problem with 4 state variables $\{x_1, x_2, x_3, x_4\}$ and 4 actions $\{a_1, a_2, a_3, a_4\}$ where a_0 is considered to be the noop action. There are 2 hidden layers, each having 3x the number of units as the number of state variables. It is assumed that the layers are fully connected, so that there is a weight connecting each pair of units in neighboring layers. The figure does not show all of the weight connections.

Note that this default network architecture is well defined for any RDDL domain and the size scales with the number of state variables and actions. Note that the infrastructure uses the PyTorch libraries for all neural network structures and training.

After training is finished the returned policy is automatically evaluated by the RDDL server for 30 episodes. You can go to the ssh terminal of the server and see the actions of the agent. Also in the ssh terminal where training was executed you can scroll back and observe both the states and actions during the evaluation.

During training the algorithm performs and prints out an evaluation of the current policy after every 20 steps. Each evaluation involves running the greedy policy *without exploration* for a number of episodes and then averaging the results. The number of test episodes used for each evaluation can be set via the command-line parameter **--test_episodes**, which defaults to 30. The average reward of each test evaluation is printed as the algorithm runs. Further, the algorithm keep track of the best model found so far in terms of test episode performance and saves the current best model to disk. Whenever an improved model is found and saved the code will print “model saved”. The model is saved to:

```
/vagrant/RL/results/wildfire_single_action_inst_mdp__1/QLearning_normalized/wildfire_inst_mdp__1_model.p
```

Note that you will not likely need to access this file yourself. It is this best model that is used for the RDDL server evaluation at the end of learning. The code also creates graphs of the learning curves under the directory:

```
/vagrant/RL/results/wildfire_single_action_inst_mdp__1/QLearning_normalized/plots/
```

Note that the command-line argument **--pathsuffix <string>** is used to influence the naming of the directory (in this case **--pathsuffix "normalized"**). Graphs are also produced for both the test and training reward versus the number of training episodes. For the above RL run the best model likely achieved a score between -45 and -65, which appears to be close to optimal in this domain (taking inherent variance of returns into account).

We will often want to continue training a previously trained agent. To do this we just need to execute the above command again, but remove the **--scratch** argument:

```
	python3 q_learning.py --inst wildfire_single_action_inst_mdp__1 --eps 0.3 --norm_reward 
--path_suffix "normalized" --train_episodes 1000
```

When the **--scratch** argument is not present, the code first looks for an existing model from previous runs (at the above mentioned location) and if it exists, the model is loaded and used as the initial policy for the training. For Q-learning this corresponds to initializing the Q-function network with the previously saved network. When you execute the above command you will notice that the initial test evaluations are based on the previously learned good policy. If we instead wanted to start from scratch and forget the first learning run, then we would have kept the **--scratch** argument and the initial test evaluations during training would have had lower values.

Important Lesson: *RL learning curves are almost never smooth and often look like very noisy signals, ideally with an upward trend. This is important to remember when selecting a final policy to return after learning. To see this clearly look at the learning curve generated from the previous "continuation run", which started from a good policy and continued training for 1000 additional episodes.*

```
/vagrant/RL/results/wildfire_single_action/wildfire_single_action_inst_mdp__1/QLearning_normal
ized/plots/Test_Performance.png
```

You will likely see that the test performance of the current policy along the curve has non-trivial variation in expected value. This illustrates why it is critical to track and store the policy encountered during training that was estimated to have the highest test performance. Otherwise if one simply returns the policy that happens to be current at the end of training, the performance may be substantially worse than that of the best policy encountered during training.

Part #2: Normalization of Rewards

The magnitude of the reward signal can often influence the proper choice of parameters, especially parameters related to the learning rate. Our library currently uses a popular adaptive learning rate control algorithms called Adam [2], which is included in the PyTorch library. This algorithm has a parameter called 'lr' that controls how "aggressive" the learning is initially. Setting this parameter properly is often important for fast and good convergence. Unfortunately, the best value for the parameter can depend very much on the magnitude of the reward signal.

To see this experimentally, we will rerun the above experiment, but will remove the **--norm_reward** command-line argument.

```
        python3 q_learning.py --inst wildfire_single_action_inst_mdp__1 --eps 0.3 --path_suffix
"raw_reward" --train_episodes 1000 --scratch
```

This has the effect of having the Q-learning agent learn directly from the raw reward signal in the wildfire domain as is usually the case in algorithm descriptions. The rewards in this domain are of the order of magnitudes 10^2 to 10^3.

> *How does the resulting policy perform compared to the policy learned with --norm_reward turned on?*

You will likely see that the RL agent does not find a policy that achieve an average test reward in the range -45 to -65.

If your agent did not perform as well, then adjusting the learning rate parameter is one mechanism to improve. The learning rate parameter for the optimizer can be set via the command-line argument **--lr <positive float>** which has a default of 0.0001.

Exercise: *Vary the learning rate parameter (try 3 to 4 values) with the goal of getting the agent with unnormalized reward to learn a policy in the range -45 to -65 within 1000 episodes. For example, decreasing by an order of magnitude would yield the following command line:*

```
        python3 q_learning.py --inst wildfire_single_action_inst_mdp__1 --eps 0.3 --path_suffix
"raw_reward_lr_0.00001 --lr 0.00001 --train_episodes 1000 --scratch
```

> *Were you able to find a learning rate parameter that achieved our goals?*

For this small problem, finding an appropriate parameter may not be too difficult, since running the episodes is not too expensive and we don't need to run many of them to learn. For larger problems where training is very expensive, searching through learning rate parameters can be problematic.

If our goal is to develop an RL system for arbitrary RDDL domains, it is importan to have a robust and automatic mechanism for dealing with varying reward magnitudes, since RDDL domains can have arbitrary magnitudes. One way to do this is to normalize the rewards inside the algorithm by dividing each reward by the range of the reward values (difference between the maximum possible reward and the minimum possible reward). For RDDL domains it is possible to compute upper and lower bounds on the reward magnitude by analyzing the domain and problem definition. By including the **--norm_reward** in the command-line, this normalization is applied. Thus, the learning agent learns from the normalized reward rather than the raw reward. The default learning rate of 0.0001 has been found to work reasonably well with this reward normalization scheme. We note that this has not been explored in depth and it is likely that other approaches will ultimately improve on these choices.

Part #3: Neural Net Architecture

The above results used the default policy representation (2 hidden layers). One should always ask whether or not the complexity of a neural network is needed compared to simpler representations such as linear functions (i.e. zero hidden layers).

Exercise: *Modify q_learning.py to use and learn a linear policy, with zero hidden layers and compare its learning performance to the 2 hidden-layer network.*

To do this first make a copy of **q_learning.py** called **linear_q_learning.py.** Modify **linear_q_learning.py** as follows.

1) Comment out the two procedures __init__(self) and forward(self, input) in **class QNet(nn.Module)** under the heading """ **Create network with 2 hidden layers** """
2) Uncomment the procedures __init__(self) and forward(self, input) under the heading """ **Create network with 0 hidden layers** """

Now run Q-learning again using the resulting linear network.

```
        python3 linear_q_learning.py --inst wildfire_single_action_inst_mdp__1 --eps 0.3 --
norm_reward --path_suffix "normalized_linear" --train_episodes 1000 --scratch
```

Did you notice any significant difference between the linear and non-linear versions?

It is likely that the linear network worked just as well for this problem. Indeed, if you think about the problem, it is not hard to see that the Q-values of the actions can be reasonably approximated via linear functions.

For more complex functions, deeper neural networks structures are essential to achieving top performance. As an example let us consider a very simple domain called "xor". This domain has just two binary state variables 'code0' and 'code1' and a single action 'detect-xor'. At each time step the values of 'code0' and 'code1' is selected by flipping a fair coin. The reward function is defined as:

- If *xor(code0, code1) == true*, then 'detect-xor' gives a reward of 0 and otherwise selecting 'noop' gives a reward of -100.
- If *xor(code0, code1) == false* then 'noop' gives reward 0 and 'detect-xor' gives reward -100.

The optimal policy for the "xor" problem is to return 'detect-xor' iff xor(code0, code1) == true.

To run this experiment, kill the currently running RDDL server and start up the following server instance:

```
        cd /vagrant/RDDLSim

        ./run rddl.competition.Server /vagrant/RL/env/xor
```

Conceptually the default 2 hidden-layer network can represent the required xor function. However, to allow for faster and more reliable learning we will increase the size of the hidden layers. To do this in **q_learning.py** change the value of **hidden_unit_factor** from 3 to 15. Next, train this network for 1000 episodes.

```
        python3 q_learning.py --inst xor_inst_mdp__1 --eps 0.3 --path_suffix "normalized" --
train_episodes 1000 --scratch
```

This network should find a policy with an optimal average reward of 0 quite quickly.

Now let’s see if the linear network is able to learn an optimal policy.

```
        python3 linear_q_learning.py --inst xor_inst_mdp__1 --eps 0.3 --path_suffix
"normalized_linear" --train_episodes 1000 -scratch
```

What did you observe?

It is well established that linear networks are not capable of representing the xor function. So no matter how long you run this network, it will not achieve an optimal policy.

The simple xor example was a simple illustrative example. We can now consider a more complex example where we merge the xor concept into the single-action wildfire domain. To do this, kill the current RDDL server and startup the wildfire_single_action_xor domain:

```
        cd /vagrant/RDDLSim

        ./run rddl.competition.Server /vagrant/RL/env/wildfire_single_action_xor
```

This domain adds the ‘code0’ and ‘code1’ variables to single-action wildfire and has two ‘put-out-xor’ and ‘put-out-not-xor’. We think of the code variables as indicating which of two fire-fighting troops are available at a given time step.

- The “xor troop” is available iff *xor(code0, code1) == true*.
- The “not-xor troop” is available iff *xor(code0, code1) == false.*
- The only way to put out the fires at a time step is to send the available troop via the appropriate acton, either ‘put-out-xor’ or ‘put-out-not-xor’. Selecting an unavailable troop fails to put out any fires.

This problem is quite a bit more difficult to learn than the basic single-action wildfire domain. We already know that a linear network will not be able to learn an optimal policy for this domain, since it requires encoding the xor function in the policy.

Exercise: Try to learn a high-quality policy for this domain using a 2 hidden-layer network with **hidden_unit_factor = 15.**

```
        python3 q_learning.py --inst wildfire_single_action_xor_inst_mdp__1 --eps 0.3 --
path_suffix "normalized" --train_episodes 10000 --scratch
```

You should be able to learn a good policy for this domain (expected reward within -45 to -65) within 15K episodes. If the about 10K training episodes is not enough, then learn for another 5K episodes (make sure to remove the **--scratch** flag).

Part #4: DQN: Experience Replay

The basic Q-learning algorithm processes each state transition once and then throws away the transition data. This can be wasteful, especially when collecting data is expensive (e.g. an expensive simulator). As discussed in the lecture, experience replay is a mechanism to more effectively using transition data by saving experience and performing multiple updates on experience tuples. A recent example of Q-learning with experience replay is the DQN algorithm, which was the first to demonstrate successful RL performance on Atari games based on pixel-level input using convolutional neural network (CNN) policies [3].

Brief DQN Description: DQN is quite simple. It follows an exploration policy and stores each transition tuple (s, a, r, s') in a replay buffer of some specified maximum size (randomly removing tuples when the size is exceeded). After each transition DQN samples a batch of tuples from the replay buffer and does a batch gradient update to the Q-function using the standard Q-learning target value for each tuple in the batch. Further, DQN also uses the concept of a “target network” and “performance network”. The target network is used to compute the target values used to update the performance network during learning. After every C state transitions the current performance network is copied to the target network. This idea appears to sometimes stabilize Q-learning.

DQN is implemented in **q_learning_exp_replay.py**. In addition to the command line arguments for **q_learning.py**, DQN has the following arguments:

- --capacity <int> specifies the maximum replay buffer size
- --batch_size <int> specifies the batch size used for each Q-function update
- --target_update <int> specifies T, the number of steps between updates of the target network

We can get an algorithm that is effectively equivalent to the basic Q-learning algorithm explored earlier in the lab by using arguments: **--capacity 1 --batch_size 1 --target_update 1**.

Exercise: Observe whether or not DQN can find a good policy for single-action wildfire in fewer trajectories (on average) compared to regular Q-learning.

To do this, first kill and restart the RDDL server for the wildfire_single_action domain. Run DQN via the following:

```
        python3 q_learning_exp_replay.py --inst wildfire_single_action_inst_mdp__1 --eps 0.3 --
norm_reward --path_suffix "normalized" --train_episodes 10000 --scratch --capacity 1000 --
batch_size 64 --target_update 10
```

Do this several times to see on average how many episodes it takes to find a policy in the good range (-45 to -65). Do the same for standard Q-learning (make sure to set **hidden_unit_factor** in q_learning.py back to 3 to be comparable with DQN. The above DQN parameters were just an initial guess that seemed to work, but you may want to play with them to see the impact (if any) of each one.

Was there a clear difference between the "episode efficiency" of DQN versus pure Q-learning?

While DQN will often be more efficient in terms of number of episodes required to reach a certain performance level, each episode takes more computational time compared to Q-learning. Thus, DQN can has a larger potential to improve overall runtime when the environment simulator is more computationally expensive to apply.

Part #5: Full Wildfire

Now that you've got some experience, you can go ahead and attempt to learn a policy for the original wildfire domain. Kill and restart the RDDL server for wildfire:

```
cd /vagrant/RDDLSim
```

```
./run rddl.competition.Server /vagrant/RL/env/wildfire
```

This is a much harder problem, since now there are two actions for each cell: 'cut-out(x,y)' and 'put-out(x,y)'. For the first problem instance with 9 cells, this gives a total of 19 actions (considering noop). We should expect learning to take significantly more time. Indeed, during learning in order to one experience that shows the utility of an action it is necessary for there to be a fire in a cell and then to take the appropriate action for that cell via exploration.

It appears that the basic Q-learning agent can achieve near optimal performance in 10K to 15K episodes. The achievable reward is close to that achievable in the single-action case.

Exercise: Learn a near optimal policy for this problem using any combination of the lessons learned above. As time allows, consider variations and observe the impact on learning efficiency (both in terms of number of episodes and time).

References

[1] V. Mnih, A. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. (2016). Asynchronous methods for deep reinforcement learning. *International Conference on Machine Learning*.

[2] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen. (2015). Human-level control through deep reinforcement learning. *Nature*. Feb;518(7540):529.

[4] Murugeswari Issakkimuthu, Alan Fern, and Prasad Tadepalli. (2018). Training Deep Reactive Policies for Probabilistic Planning Problems. *International Conference on Automated Planning and Scheduling Systems.*