Index

June 27, 2019

1 Reinforcement Learning Using OpenAI Gym

1.1 Quick Introduction to RL (Reinforcement Learning)

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

It differs from supervised learning in that labelled input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

The environment is typically formulated as a Markov decision process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible.

source: https://en.wikipedia.org/wiki/Reinforcement_learning

1.2 Experiments with OpenAI Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano.

The gym library is a collection of test problems — environments — that you can use to work out your reinforcement learning algorithms. These environments have a shared interface, allowing you to write general algorithms.

source: https://gym.openai.com/docs/

1.3 Learn by doing – A reinforcement learning experiment using OpenAI Gym

1.3.1 Creating a Gym environment

To start using OpenAI Gym, one must install it and import it into a Python environment. One can either use a pre-made environment for testing, or build their own.

For this experiment, I will be using the pre-built 'Acrobot' environment. The state consists of the sin() and cos() of the two rotational joint angles and the joint angular velocities [cos(theta1) sin(theta1) cos(theta2) sin(theta2) thetaDot1 thetaDot2]. The action is either applying +1, 0 or -1

torque on the joint between the two pendulum links. The goal is to get the movable joint above the thin black line. The reward is either 1 or -1, depending on the success of the agent.

We can view the action and observation space as follows.

[1]:
```
import gym

env = gym.make("Acrobot-v1")
observation = env.reset()

print(f"Action Space Shape: {env.action_space}")
print(f"Action Example: {env.action_space.sample()}")

print(f"Observation Space Shape: {env.observation_space}")
print(f"Observation Example: {observation}")
```

```
Action Space Shape: Discrete(3)
Action Example: 1
Observation Space Shape: Box(6,)
Observation Example: [ 0.99993867  0.01107472  0.99939632 -0.03474194
0.08784581  0.05837633]
```

We can also render the environment, until an episode completes, as follows.

[2]:
```
# Enables recording
# import gym
# from gym.wrappers import Monitor
# env = gym.make("Acrobot-v1")
# env = Monitor(env, './video')

env.reset()
done = False

while not done:
  env.render()
  action = env.action_space.sample()
  observation, reward, done, info = env.step(action)

env.close()
```

In the prior code block, we take random actions and only utilize the `done` information to determine if a session has finished. You can either run the code to see the animation for youself, or view the video below.

1.4 Building a smarter agent

1.4.1 Tabular

Tabular methods use arrays and tables to hold an approximate of the value functions. More simply, they store every combination of state/action pairs and hold a perceived value of that pair. This simplifies the learning process because tables are very quick to compute and can give decent

results given the state and action spaces are small enough. One can also reduce the size of the state and action spaces to make the agent learn evey quicker due to a smaller table to fill (at the cost of a reduced accuracy). One idea that can be implemented, that I will not use in the Acrobot example, is to transform state values before binning them. For example, one could use a sigmoid function to spread the state values out so that the center distrubtion gets put into more bins while the outliers get grouped together.

To apply a tabular method to this Acrobot example, we simply need a table that holds every possible state/action pair. Due to the size of this table, I bin the state values into 5 buckets so that there are 3 * 6^5 values to compute. I also add a randomness factor to force my agent to explore more in the beginning then slowly exploit more as it learns. I have the training render a session every 10,000 sessions. To disable this, comment out the render or change the `RENDER_STEPS` variable.

[ ]:
```
# Configuration Values
```

[ ]:
```
# Table Class
```

[ ]:
```
# Training
```

[ ]:
```
# VIDEO (HTML) DEMONSTRATION
```

1.4.2 DQN

1.4.3 PPO