Если вы видите что-то необычное, просто сообщите мне.

Open AI Gym Primer: Frozen Lake

Well to our series on Haskell and the Open AI Gym! The Open AI Gym is an open source project for teaching the basics of reinforcement learning. It provides a framework for understanding how we can make agents that evolve and learn. It's written in Python, but some of its core concepts work very well in Haskell. So over the course of this series, we'll be implementing many of the main ideas in our preferred language.

In this first part, we'll start exploring what exactly these core concepts are, so we'll stick to the framework's native language. We'll examine what exactly an "environment" is and how we can generalize the concept. If you already know some of the basics of Open AI Gym and the Frozen Lake game, you should move on to part 2 of the series, where we'll start using Haskell!

We'll ultimately use machine learning to train our agents. So you'll want some guidance on how to do that in Haskell. Read our Machine Learning Series and download our Tensor Flow guide to learn more!

FROZEN LAKE

To start out our discussion of AI and games, let's go over the basic rules of one of the simplest examples, Frozen Lake. In this game, our agent controls a character that is moving on a 2D "frozen lake", trying to reach a goal square. Aside from the start square ("S") and the goal zone ("G"), each square is either a frozen tile ("F") or a hole in the lake ("H"). We want to avoid the holes, moving only on the frozen tiles. Here's a sample layout:

SFFF FHFH FFFH HFFG So a safe path would be to move down twice, move right twice, down again, and then right again. What complicates the matter is that tiles can be "slippery". So each turn, there's a chance we won't complete our move, and will instead move to a random neighboring tile.

PLAYING THE GAME

Now let's see what it looks like for us to actually play the game using the normal Python code. This will get us familiar with the main ideas of an environment. We start by "making" the environment and setting up a loop where the user can enter their input move each turn:

```
import gym
env = gym.make('FrozenLake-v0')
env.reset()

while True:
  move = input("Please enter a move:")
  ...
```

There are several functions we can call on the environment to see it in action. First, we'll render it, even before making our move. This lets us see what is going on in our console. Then we have to step the environment using our move. The step function makes our move and provides us with 4 outputs. The primary ones we're concerned with are the "done" value and the "reward". These will tell us if the game is over, and if we won.

```
while True:
  env.render()
  move = input("Please enter a move:")
  action = int(move)
  observation, reward, done, info = env.step(action)
  if done:
    print(reward)
    print("Episode finished")
    env.render()
    break
```

We use numbers in our moves, which our program converts into the input space for the game. (0 = Left, 1 = Down, 2 = Right, 3 = Up).

We can also play the game automatically, for several iterations. We'll select random moves by using action_space.sample(). We'll discuss what the action space is in the next part. We can also use reset on our environment at the end of each iteration to return the game to its initial state.

```
for i in range(20):
  observation = env.reset()
  for t in range(100):
    env.render()
    print(observation)
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    if done:
      print("Episode finished after {} timesteps".format(t + 1))
      break

env.close()
```

These are the basics of the game. Let's go over some of the details of how an environment works, so we can start imagining how it will work in Haskell.

OBSERVATION AND ACTION SPACES

The first thing to understand about environments is that each environment has an "observation" space and an "action" space. The observation space gives us a numerical representation of the state of the game. This doesn't include the actual layout of our board, just the mutable state. For our frozen lake example, this is only the player's current position. We could use two numbers for the player's row and column. But in fact we use a single number, the row number multiplied by the column number.

Here's an example where we print the observation after moving right twice, and then down. We have to call reset before using an environment. Then calling this function gives us an observation we can print. Then, after each step, the first return value is the new observation.

```
import gym
env = gym.make('FrozenLake-v0')
o = env.reset()
print(o)
o, _, _, _ = env.step(2)
```

```
print(o)
o, _, _, _ = env.step(2)
print(o)
o, _, _, _ = env.step(1)
print(o)


# Console output
0
1
2
6
```

So, with a 4x4 grid, we start out at position 0. Then moving right increases our position index by 1, and moving down increases it by 4.

This particular environment uses a "discrete" environment space of size 16. So the state of the game is just a number from 0 to 15, indicating where our agent is. More complicated games will naturally have more complicated state spaces.

The "action space" is also discrete. We have four possible moves, so our different actions are the integers from 0 to 3.

```
import gym
env = gym.make('FrozenLake-v0')
print(env.observation_space)
print(env.action_space)

# Console Output
Discrete(16)
Discrete(4)
```

The observation space and the action space are important features of our game. They dictate the inputs and outputs of the each game move. On each turn, we take a particular observation as input, and produce an action as output. If we can do this in a numerical way, then we'll ultimately be able to machine-learn the program.

TOWARDS HASKELL

Now we can start thinking about how to represent an environment in Haskell. Let's think about the key functions and attributes we used when playing the game.

1. Observation space
2. Action space
3. Reset
4. Step
5. Render How would we represent these in Haskell? To start, we can make a type for the different numeric spaces can have. For now we'll provide a discrete space option and a continuous space option.

```
data NumericSpace =
  Discrete Int |
  Continuous Float
Now we can make an Environment type with fields for these spaces. We'll give it parameters for
the observation type and the action type.

data Environment obs act = Environment
  { observationSpace :: NumericSpace
  , actionSpace :: NumericSpace
  ...
  }
```

We don't know yet all the rest of the data our environment will hold. But we can start thinking about certain functions for it. Resetting will take our environment and return a new environment and an observation. Rendering will be an IO action.

```
resetEnv :: Environment obs act -> (obs, Environment obs act)

renderEnv :: Environment obs act -> IO ()
```

The step function is the most important. In Python, this returns a 4-tuple. We don't care about the 4th "info" element there yet. But we do care to return our environment type itself, since we're in a functional language. So we'll return a different kind of 4-tuple.

```
stepEnv :: Environment obs act -> act
  -> (obs, Float, Bool, Environment obs act)
```

It's also possible we'll use the state monad here instead, as that could be cleaner. Now this isn't the whole environment obviously! We'd need to store plenty of unique internal state. But what we see here is the start of a typeclass that we'll be able to generalize across different games. We'll see how this idea develops throughout the series!

CONCLUSION

Hopefully you've got a basic idea now of what makes up an environment we can run. You can take a look at part 2, where we'll push a bit further with our Haskell and implement Frozen Lake !

Revision #1
Created 2022-03-11 06:39:54 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick