

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

Frozen Lake in Haskell

In part 1 of this series, we began our investigation into Open AI Gym. We started by using the Frozen Lake toy example to learn about environments. An environment is a basic wrapper that has a specific API for manipulating the game.

Part 1's work was mostly in Python. But in this part, we're going to do a deep dive into Haskell and consider how to write the Frozen Lake example. We'll see all the crucial functions from the Environment API as well as how to play the game. You can take a look at our Github repository to see any extra details about this code throughout this series! For this part in particular, for this part, you'll want to look at `FrozenLakeBasic.hs`.

This process will culminate with training agents to complete these games with machine learning. This will involve TensorFlow. So if you haven't already, download our Haskell Tensor Flow Guide. It will teach you how to get the framework up and running on your machine.

CORE TYPES

In the previous part, we started defining our environment with generic values. For example, we included the action space and observation space. For now, we're actually going to make things more specific to the Frozen Lake problem. This will keep our example much simpler for now. In the rest of the series, we'll start examining how to generalize the idea of an environment and spaces.

We need to start with the core types of our application. We'll begin with a `TileType` for our board, as well as observations and actions.

```
data TileType =  
  Start |  
  Goal |  
  Frozen |
```

```

Hole
deriving (Show, Eq)

type Observation = Word

data Action =
  MoveLeft |
  MoveDown |
  MoveRight |
  MoveUp
deriving (Show, Eq, Enum)

```

As in Python, each observation will be a single number indicating where we are on the board. We'll have four different actions. The Enum instance will help us convert between these constructors and numbers.

Now let's consider the different elements we actually need within the environment. The game's main information is the grid of tiles. We'll store this as an Array. The indices will be our observation values, and the elements will be the TileType. For convenience, we'll also store the dimensions of our grid:

```

data FrozenLakeEnvironment = FrozenLakeEnvironment
  { grid :: Array Word TileType
  , dims :: (Word, Word) -- Rows, Columns
  ...
  }

```

We also need some more information. We need the current player location, an Observation. We'll want to know the previous action, for rendering purposes. The game also stores the chance of slipping each turn. The last piece of state we want is the random generator. Storing this within our environment lets us write our step function in a pure way, without IO.

```

data FrozenLakeEnvironment = FrozenLakeEnvironment
  { grid :: Array Word TileType
  , dims :: (Word, Word) -- Rows, Cols
  , currentObservation :: Observation
  , previousAction :: Maybe Action
  , slipChance :: Double

```

```
, randomGenerator :: Rand.StdGen  
}
```

API FUNCTIONS

Now our environment needs its API functions. We had three main ones last time. These were `reset`, `render`, and `step`. In part 1 we wrote these to take the environment as an explicit parameter. But this time, we'll write them in the State monad. This will make it much easier to chain these actions together later. Let's start with `reset`, the simplest function. All it does is set the observation as 0 and remove any previous action.

```
resetEnv :: (Monad m) => StateT FrozenLakeEnvironment m Observation  
resetEnv = do  
  let initialObservation = 0  
  fle <- get  
  put $ fle { currentObservation = initialObservation  
            , previousAction = Nothing }  
  return initialObservation
```

Rendering is a bit more complicated. When resetting, we can use any underlying monad. But to `render`, we'll insist that the monad allows IO, so we can print to console. First, we get our environment and pull some key values out of it. We want the current observation and each row of the grid.

```
renderEnv :: (MonadIO m) => StateT FrozenLakeEnvironment m ()  
renderEnv = do  
  fle <- get  
  let currentObs = currentObservation fle  
      elements = A.assocs (grid fle)  
      numCols = fromIntegral . snd . dimensions $ fle  
      rows = chunksOf numCols elements  
  ...
```

We use `chunksOf` with the number of columns to divide our grid into rows. Each element of each row-list is the pairing of the "index" with the tile type. We keep the index so we can compare it to the current observation. Now we'll write a helper to render each of these rows. We'll have another

helper to print a character for each tile type. But we'll print X for the current location.

```
renderEnv :: (MonadIO m) => StateT FrozenLakeEnvironment m ()
renderEnv = do
  ...
  where
    renderRow currentObs row = do
      forM_ row \(idx, t) -> liftIO $ if idx == currentObs
        then liftIO $ putChar 'X'
        else liftIO $ putChar (tileToChar t)
      putChar '\n'

tileToChar :: TileType -> Char
...
```

Then we just need to print a line for the previous action, and render each row:

```
renderEnv :: (MonadIO m) => StateT FrozenLakeEnvironment m ()
renderEnv = do
  file <- get
  let currentObs = currentObservation file
      elements = A.assocs (grid file)
      numCols = fromIntegral . snd . dims $ file
      rows = chunksOf numCols elements
  liftIO $ do
    putStrLn $ case (previousAction file) of
      Nothing -> ""
      Just a -> "  " ++ show a
    forM_ rows (renderRow currentObs)
  where
    renderRow = ...
```

STEPPING

Now let's see how we update our environment! This will also be in our State monad (without any IO constraint). It will return a 3-tuple with our new observation, a "reward", and a boolean for if we finished. Once again we start by gathering some useful values.

```

stepEnv :: (Monad m) => Action
  -> StateT FrozenLakeEnvironment m (Observation, Double, Bool)
stepEnv act = do
  fle <- get
  let currentObs = currentObservation fle
  let (slipRoll, gen') = Rand.randomR (0.0, 1.0) (randomGenerator fle)
  let allLegalMoves = legalMoves currentObs (dims fle)
  let (randomMoveIndex, finalGen) =
    randomR (0, length AllLegalMoves - 1) gen'
  ...

-- Get all the actions we can do, given the current observation
-- and the number of rows and columns
legalMoves :: Observation -> (Word, Word) -> [Action]
...

```

We now have two random values. The first is for our "slip roll". We can compare this with the game's slipChance to determine if we try the player's move or a random move. If we need to do a random move, we'll use randomMoveIndex to figure out which random move we'll do.

The only other check we need to make is if the player's move is "legal". If it's not we'll stand still. The applyMoveUnbounded function tells us what the next Observation should be for the move. For example, we add 1 for moving right, or subtract 1 for moving left.

```

stepEnv :: (Monad m) => Action
  -> StateT FrozenLakeEnvironment m (Observation, Double, Bool)
stepEnv act = do
  ...
  let newObservation = if slipRoll >= slipChance fle
    then if act `elem` allLegalMoves
      then applyMoveUnbounded
        act currentObs (snd . dims $ fle)
      else currentObs
    else applyMoveUnbounded
      (allLegalMoves !! nextIndex)
      currentObs
      (snd . dims $ fle)
  ...

```

```
applyMoveUnbounded ::  
  Action -> Observation -> Word -> Observation  
...
```

To wrap things up we have to figure out the consequences of this move. If it lands us on the goal tile, we're done and we get a reward! If we hit a hole, the game is over but our reward is 0. Otherwise there's no reward and the game isn't over. We put all our new state data into our environment and return the necessary values.

```
stepEnv :: (Monad m) => Action  
  -> StateT FrozenLakeEnvironment m (Observation, Double, Bool)  
stepEnv act = do  
  ...  
  let (done, reward) = case (grid fle) A.! newObservation of  
    Goal -> (True, 1.0)  
    Hole -> (True, 0.0)  
    _ -> (False, 0.0)  
  put $ fle { currentObservation = newObservation  
    , randomGenerator = finalGen  
    , previousAction = Just act }  
  return (newObservation, reward, done)
```

PLAYING THE GAME

One last step! We want to be able to play our game by creating a gameLoop. The final result of our loop will be the last observation and the game's reward. As an argument, we'll pass an expression that can generate an action. We'll give two options. One for reading a line from the user, and another for selecting randomly. Notice the use of toEnum, so we're entering numbers 0-3.

```
gameLoop :: (MonadIO m) =>  
  StateT FrozenLakeEnvironment m Action ->  
  StateT FrozenLakeEnvironment m (Observation, Double)  
gameLoop chooseAction = do  
  ...  
  
chooseActionUser :: (MonadIO m) => m Action
```

```
chooseActionUser = (toEnum . read) <$> (liftIO getLine)

chooseActionRandom :: (MonadIO m) => m Action
chooseActionRandom = toEnum <$> liftIO (Rand.randomRIO (0, 3))
```

Within each stage of the loop, we render the environment, generate a new action, and step the game. Then if we're done, we return the results. Otherwise, recurse. The power of the state monad makes this function quite simple!

```
gameLoop :: (MonadIO m) =>
  StateT FrozenLakeEnvironment m Action ->
  StateT FrozenLakeEnvironment m (Observation, Double)
gameLoop chooseAction = do
  renderEnv
  newAction <- chooseAction
  (newObs, reward, done) <- stepEnv newAction
  if done
    then do
      liftIO $ print reward
      liftIO $ putStrLn "Episode Finished"
      renderEnv
      return (newObs, reward)
    else gameLoop chooseAction
```

And now to play our game, we start with a simple environment and execute our loop!

```
basicEnv :: IO FrozenLakeEnvironment
basicEnv = do
  gen <- Rand.getStdGen
  return $ FrozenLakeEnvironment
    { currentObservation = 0
    , grid = A.listArray (0, 15) (charToTile <$> "SFFFFHFHFFFHHFFG")
    , slipChance = 0.0
    , randomGenerator = gen
    , previousAction = Nothing
    , dims = (4, 4)
    }

playGame :: IO ()
```

```
playGame = do
  env <- basicEnv
  void $ execStateT (gameLoop chooseActionUser) env
```

CONCLUSION

This example illustrates two main lessons. First, the state monad is very powerful for managing any type of game situation. Second, defining our API makes implementation straightforward. Next up is part 3, where we'll explore Blackjack, another toy example with a different state space. This will lead us on the path to generalizing our data structure.

Revision #2

Created 11 March 2022 06:41:37 by gasick

Updated 11 March 2022 17:11:16 by gasick