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

Basic Q-Learning

In the last two parts of this series, we've written two simple games in Haskell: Frozen Lake and Blackjack. Now that we've written the games, it's time to explore more advanced ways to write agents for them.

In this article, we'll explore the concept of Q-Learning. This is one of the simplest approaches in reinforcement learning. We'll write a little bit of Python code, following some examples for Frozen Lake. Then we'll try to implement the same ideas in Haskell. Along the way, we'll see more patterns emerge about our games' interfaces.

We won't be using Tensorflow in the article. But we'll soon explore ways to augment our agent's capabilities with this library! To learn about Haskell and Tensorflow, download our TensorFlow guide!

MAKING A Q-TABLE

Let's start by taking a look at this basic Python implementation of Q-Learning for Frozen Lake. This will show us the basic ideas of Q-Learning. We start out by defining a few global parameters, as well as Q, a variable that will hold a table of values.

```
epsilon = 0.9
min_epsilon = 0.01
decay_rate = 0.9
Total_episodes = 10000
max_steps = 100
learning_rate = 0.81
gamma = 0.96

env = gym.make('FrozenLake-v0')
Q = numpy.zeros((env.observation_space.n, env.action_space.n))
```

Recall that our environment has an action space and an observation space. For this basic version of the Frozen Lake game, an observation is a discrete integer value from 0 to 15. This represents the location our character is on. Then the action space is an integer from 0 to 3, for each of the four directions we can move. So our "Q-table" will be an array with 16 rows and 4 columns.

How does this help us choose our move? Well, each cell in this table has a score. This score tells us how good a particular move is for a particular observation state. So we could define a choose_action function in a simple way like so:

```
def choose_action(observation):
  return numpy.argmax(Q[observation, :])
```

This will look at the different values in the row for this observation, and choose the highest index. So if the "0" value in this row is the highest, we'll return 0, indicating we should move left. If the second value is highest, we'll return 1, indicating a move down.

But we don't want to choose our moves deterministically! Our Q-Table starts out in the "untrained" state. And we need to actually find the goal at least once to start back-propagating rewards into our maze. This means we need to build some kind of exploration into our system. So each turn, we can make a random move with probability epsilon.

```
def choose_action(observation):
  action = 0
  if np.random.uniform(0, 1) < epsilon:
    action = env.action_space.sample()
  else:
    action = numpy.argmax(Q[observation, :])
  return action
```

As we learn more, we'll diminish the exploration probability. We'll see this below!

UPDATING THE TABLE

Now, we also want to be able to update our table. To do this, we'll write a function that follows the Q-learning rule. It will take two observations, the reward for the second observation, and the action we took to get there.

```
def learn(observation, observation2, reward, action):
  prediction = Q[observation, action]
  target = reward + gamma * numpy.max(Q[observation2, :])
  Q[observation, action] = Q[observation, action] +
                           learning_rate * (target - prediction)
```

For more details on what happens here, you'll want to do some more in-depth research on Q-Learning. But there's one general rule.

Suppose we move from Observation O1 to Observation O2 with action A. We want the Q-table value for the pair (O1, A) to be closer to the best value we can get from O2. And we want to factor in the potential reward we can get by moving to O2. Thus our goal square should have the reward of 1. And squares near it should have values close to this reward!

PLAYING THE GAME

Playing the game now is straightforward, following the examples we've done before. We'll have a certain number of episodes. Within each episode, we make our move, and use the reward to "learn" for our Q-table.

```
for episode in range(total_episodes):
  obs = env.reset()
  t = 0
  if episode % 100 == 99:
    epsilon *= decay_rate
    epsilon = max(epsilon, min_epsilon)

  while t < max_steps:
    action = choose_action(obs)
    obs2, reward, done, info = env.step(action)
    learn(obs, obs2, reward, action)
    obs = obs2
    t += 1

    if done:
      if reward > 0.0:
        print("Win")
```

```
    else:
      print("Lose")
    break
```

Notice also how we drop the exploration rate epsilon every 100 episodes or so. We can run this, and we'll observe that we lose a lot at first. But by the end we're winning more often than not! At the end of the series, it's a good idea to save the Q-table in some sensible way.

HASKELL: ADDING A Q-TABLE

To translate this into Haskell, we first need to account for our new pieces of state. Let's extend our environment type to include two more fields. One will be for our Q-table. We'll use an array for this as well, as this gives convenient accessing and updating syntax. The other will be the current exploration rate:

```
data FrozenLakeEnvironment = FrozenLakeEnvironment
  { ...
  , qTable :: A.Array (Word, Word) Double
  , explorationRate :: Double
  }
```

Now we'll want to write two primary functions. First, we'll want to choose our action using the Q-Table. Second, we want to be able to update the Q-Table so we can "learn" a good path.

Both of these will use this helper function. It takes an Observation and the current Q-Table and produces the best score we can get from that location. It also provides us the action index. Note the use of a tuple section to produce indices.

```
maxScore ::
  Observation ->
  A.Array (Word, Word) Double ->
  (Double, (Word, Word))
maxScore obs table = maximum valuesAndIndices
```

```
  where
    indices = (obs, ) <$> [0..3]
    valuesAndIndices = (\i -> (table A.! i, i)) <$> indices
```

USING THE Q-TABLE

Now let's see how we produce our actions using this table. As with most of our state functions, we'll start by retrieving the environment. Then we'll get our first roll to see if this is an exploration turn or not.

```
chooseActionQTable ::
  (MonadState FrozenLakeEnvironment m) => m Action
chooseActionQTable = do
  fle <- get
  let (exploreRoll, gen') = randomR (0.0, 1.0) (randomGenerator fle)
  if exploreRoll < explorationRate fle
    ...
```

If we're exploring, we do another random roll to pick an action and replace the generator. Otherwise we'll get the best scoring move and derive the Action from the returned index. In both cases, we use toEnum to turn the number into a proper Action.

```
chooseActionQTable ::
  (MonadState FrozenLakeEnvironment m) => m Action
chooseActionQTable = do
  fle <- get
  let (exploreRoll, gen') = randomR (0.0, 1.0) (randomGenerator fle)
  if exploreRoll < explorationRate fle
    then do
      let (actionRoll, gen'') = Rand.randomR (0, 3) gen'
      put $ fle { randomGenerator = gen'' }
      return (toEnum actionRoll)
    else do
      let maxIndex = snd $ snd $
                       maxScore (currentObservation fle) (qTable fle)
      put $ fle {randomGenerator = gen' }
      return (toEnum (fromIntegral maxIndex))
```

The last big step is to write our learning function. Remember this takes two observations, a reward, and an action. We start by getting our predicted value for the original observation. That is, what score did we expect when we made this move?

```
learnQTable :: (MonadState FrozenLakeEnvironment m) =>
  Observation -> Observation -> Double -> Action -> m ()
learnQTable obs1 obs2 reward action = do
  fle <- get
  let q = qTable fle
      actionIndex = fromIntegral . fromEnum $ action
      prediction = q A.! (obs1, actionIndex)
  ...
```

Now we specify our target. This combines the reward (if any) and the greatest score we can get from our new observed state. We use these values to get a newValue, which we put into the Q-Table at the original index. Then we put the new table into our state.

```
learnQTable :: (MonadState FrozenLakeEnvironment m) =>
  Observation -> Observation -> Double -> Action -> m ()
learnQTable obs1 obs2 reward action = do
  fle <- get
  let q = qTable fle
      actionIndex = fromIntegral . fromEnum $ action
      prediction = q A.! (obs1, actionIndex)
      target = reward + gamma * (fst $ maxScore obs2 q)
      newValue = prediction + learningRate * (target - prediction)
      newQ = q A.// [((obs1, actionIndex), newValue)]
  put $ fle { qTable = newQ }
  where
    gamma = 0.96
    learningRate = 0.81
```

And just like that, we're pretty much done! We can slide these new functions right into our existing functions!

The rest of the code is straightforward enough. We make a couple tweaks as necessary to our gameLoop so that it actually calls our training function. Then we just update the exploration rate at appropriate intervals.

BLACKJACK AND Q-LEARNING

We can use almost the same process for Blackjack! Once again, we will need to express our Q-table and the exploration rate as part of the environment. But this time, the index of our Q-Table will need to be a bit more complex. Remember our observation now has three different parts: the user's score, whether the player has an ace, and the dealer's show-card. We can turn each of these into a Word, and combine them with the action itself. This gives us an index with four Word values.

We want to populate this array with bounds to match the highest value in each of those fields.

```
data BlackjackEnvironment = BlackjackEnvironment
  { ...
  , qTable :: A.Array (Word, Word, Word, Word) Double
  , explorationRate :: Double
  } deriving (Show)

basicEnv :: IO BlackjackEnvironment
basicEnv = do
  gen <- Rand.getStdGen
  let (d, newGen) = shuffledDeck gen
  return $ BlackjackEnvironment
    ...
    (A.listArray ((0,0,0,0), (30, 1, 12, 1)) (repeat 0.0))
    1.0
```

While we're at it, let's create a function to turn an Observation/Action combination into an index.

```
makeQIndex :: BlackjackObservation -> BlackjackAction
  -> (Word, Word, Word, Word)
makeQIndex (BlackjackObservation pScore hasAce dealerCard) action =
  ( pScore
  , if hasAce then 1 else 0
  , fromIntegral . fromEnum $ dealerCard
  , fromIntegral . fromEnum $ action
```

```
  )
```

With the help of this function, it's pretty easy to re-use most of our code from Frozen Lake! The action choice function and the learning function look almost the same!

WRITING A GAME LOOP

With our basic functions out of the way, let's now turn our attention to the game loop and running functions. For the game loop, we don't have anything too complicated. It's a step-by-step process.

Retrieve the current observation Choose the next action Use this action to step the environment Use our "learning" function to update the Q-Table If we're done, return the reward. Otherwise recurse. Here's what it looks like. Recall that we're taking our action choice function as an input. All our functions live in a similar monad, so this is pretty easy.

```
gameLoop :: (MonadIO m) =>
  StateT BlackjackEnvironment m BlackjackAction ->
  StateT BlackjackEnvironment m (BlackjackObservation, Double)
gameLoop chooseAction = do
  oldObs <- currentObservation <$> get
  newAction <- chooseAction
  (newObs, reward, done) <- stepEnv newAction
  learnQTable oldObs newObs reward newAction
  if done
    then do
      if reward > 0.0
        then liftIO $ putStrLn "Win"
        else liftIO $ putStrLn "Lose"
      return (newObs, reward)
    else gameLoop chooseAction
```

Now to produce our final output and run game iterations, we need a little wrapper code. We create (and reset) our initial environment. Then we pass it to an action that runs the game loop and reduces the exploration rate when necessary.

```
playGame :: IO ()
playGame = do
```

```
  env <- basicEnv
  env' <- execStateT resetEnv env
  void $ execStateT stateAction env'
  where
    numEpisodes = 10000
    decayRate = 1.0
    minEpsilon = 0.01

    stateAction :: StateT BlackjackEnvironment IO ()
    stateAction = do
      rewards <- forM [1..numEpisodes] $ \i -> do
        resetEnv
        when (i `mod` 100 == 99) $ do
          bje <- get
          let e = explorationRate bje
          let newE = max minEpsilon (e * decayRate)
          put $ bje { explorationRate = newE }
        (_, reward) <- gameLoop chooseActionQTable
        return reward
      lift $ print (sum rewards)
```

Now we can play Blackjack as well! Even with learning, we'll still only get around 40% of the points available. Blackjack is a tricky, luck-based game, so this isn't too surprising.

For more details, take a look at Frozen Lake with Q-Learning and Blackjack with Q-Learning on Github.

CONCLUSION

We've now got agents that can play Frozen Lake and Blackjack coherently using Q-Learning! In part 5 of our series, we'll find the common elements of these environments and refactor them into a typeclass! We'll see the similarities between the two games.

Revision #1
Created 11 March 2022 06:50:13 by gasick
Updated 11 March 2022 17:11:16 by gasick