

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

Open AI Gym: Blackjack

So far in this series, the Frozen Lake example has been our basic tool. In part 2, we wrote it in Haskell. We'd like to start training agents for this game. But first, we want to make sure we're set up to generalize our idea of an environment.

So in this part, we're going to make another small example game. This time, we'll play Blackjack. This will give us an example of an environment that needs a more complex observation state. When we're done with this example, we'll be able to compare our two examples. The end goal is to be able to use the same code to train an algorithm for either of them. You can find the code for this part on Github [here](#).

BASIC RULES

If you don't know the basic rules of casino blackjack, take a look [here](#). Essentially, we have a deck of cards, and each card has a value. We want to get as high a score as we can without exceeding 21 (a "bust"). Each turn, we want to either "hit" and add another card to our hand, or "stand" and take the value we have.

After we get all our cards, the dealer must then draw cards under specific rules. The dealer must "hit" until their score is 17 or higher, and then "stand". If the dealer busts or our score beats the dealer, we win. If the scores are the same it's a "push".

Here's a basic Card type we'll work with to represent the card values, as well as their scores.

```
data Card =  
  Two | Three | Four | Five |  
  Six | Seven | Eight | Nine |  
  Ten | Jack | Queen | King | Ace  
  deriving (Show, Eq, Enum)
```

```
cardScore :: Card -> Word
cardScore Two = 2
cardScore Three = 3
cardScore Four = 4
cardScore Five = 5
cardScore Six = 6
cardScore Seven = 7
cardScore Eight = 8
cardScore Nine = 9
cardScore Ten = 10
cardScore Jack = 10
cardScore Queen = 10
cardScore King = 10
cardScore Ace = 1
```

The Ace can count as 1 or 11. We account for this in our scoring functions:

```
-- Returns the base sum, as well as a boolean if we have
-- a "usable" Ace.
baseScore :: [Card] -> (Word, Bool)
baseScore cards = (score, score <= 11 && Ace `elem` cards)
  where
    score = sum (cardScore <$> cards)

scoreHand :: [Card] -> Word
scoreHand cards = if hasUsableAce then score + 10 else score
  where
    (score, hasUsableAce) = baseScore cards
```

CORE ENVIRONMENT TYPES

As in Frozen Lake, we need to define types for our environment. The "action" type is straightforward, giving only two options for "hit" and "stand":

```
data BlackjackAction = Hit | Stand
  deriving (Show, Eq, Enum)
```

Our observation is more complex than in Frozen Lake. We have more information that can guide us than just knowing our location. We'll boil it down to three elements. First, we need to know our own score. Second, we need to know if we have an Ace. This isn't clear from the score, and it can give us more options. Last, we need to know what card the dealer is showing.

```
data BlackjackObservation = BlackjackObservation
  { playerScore :: Word
  , playerHasAce :: Bool
  , dealerCardShowing :: Card
  } deriving (Show)
```

Now for our environment, we'll once again store the "current observation" as one of its fields.

```
data BlackjackEnvironment = BlackjackEnvironment
  { currentObservation :: BlackjackObservation
  ...
  }
```

The main fields are about the cards in play. We'll have a list of cards for our own hand. Then we'll have the main deck to draw from. The dealer's cards will be a 3-tuple. The first is the "showing" card. The second is the hidden card. And the third is a list for extra cards the dealer draws later.

```
data BlackjackEnvironment = BlackjackEnvironment
  { currentObservation :: BlackjackObservation
  , playerHand :: [Card]
  , deck :: [Card]
  , dealerHand :: (Card, Card, [Card])
  ...
  }
```

The last pieces of this will be a boolean for whether the player has "stood", and a random generator. The boolean helps us render the game, and the generator helps us reset and shuffle without using IO.

```
data BlackjackEnvironment = BlackjackEnvironment
  { currentObservation :: BlackjackObservation
  , playerHand :: [Card]
  , deck :: [Card]
```

```
, dealerHand :: (Card, Card, [Card])
, randomGenerator :: Rand.StdGen
, playerHasStood :: Bool
} deriving (Show)
```

Now we can use these to write our main game functions. As in Frozen Lake, we'll want functions to render the environment and reset it. We won't go over those in this article. But we will focus on the core step function.

PLAYING THE GAME

Our step function starts out simply enough. We retrieve our environment and analyze the action we get.

```
stepEnv :: (Monad m) => BlackjackAction ->
  StateT BlackjackEnvironment m (BlackjackObservation, Double, Bool)
stepEnv action = do
  bje <- get
  case action of
    Stand -> ...
    Hit -> ...
```

Below, we'll write a function to play the dealer's hand. So for the Stand branch, we'll update the state variable for the player standing, and call that helper.

```
stepEnv action = do
  bje <- get
  case action of
    Stand -> do
      put $ bje { playerHasStood = True }
      playOutDealerHand
    Hit -> ...
```

When we hit, we need to determine the top card in the deck. We'll add this to our hand to get the new player score. All this information goes into our new observation, and the new state of the game.

```

stepEnv action = do
  bje <- get
  case action of
    Stand -> ...
    Hit -> do
      let (topCard : remainingDeck) = deck bje
          pHand = playerHand bje
          currentObs = currentObservation bje
          newPlayerHand = topCard : pHand
          newScore = scoreHand newPlayerHand
          newObservation = currentObs
          { playerScore = newScore
            , playerHasAce = playerHasAce currentObs ||
              topCard == Ace }
      put $ bje { currentObservation = newObservation
                  , playerHand = newPlayerHand
                  , deck = remainingDeck }
  ...

```

Now we need to analyze the player's score. If it's greater than 21, we've busted. We return a reward of 0.0 and we're done. If it's exactly 21, we'll treat that like a "stand" and play out the dealer. Otherwise, we'll continue by returning False.

```

stepEnv action = do
  bje <- get
  case action of
    Stand -> ...
    Hit -> do
      ...
      if newScore > 21
      then return (newObservation, 0.0, True)
      else if newScore == 21
      then playOutDealerHand
      else return (newObservation, 0.0, False)

```

PLAYING OUT THE DEALER

To wrap up the game, we need to give cards to the dealer until their score is high enough. So let's start by getting the environment and scoring the dealer's current hand.

```
playOutDealerHand :: (Monad m) =>
  StateT BlackjackEnvironment m (BlackjackObservation, Double, Bool)
playOutDealerHand = do
  bje <- get
  let (showCard, hiddenCard, restCards) = dealerHand bje
  currentDealerScore = scoreHand (showCard : hiddenCard : restCards)
```

If the dealer's score is less than 17, we can draw the top card, add it to their hand, and recurse.

```
playOutDealerHand :: (Monad m) => StateT BlackjackEnvironment m (BlackjackObservation, Double, Bool)
playOutDealerHand = do
  ...
  if currentDealerScore < 17
  then do
    let (topCard : remainingDeck) = deck bje
    put $ bje { dealerHand =
      (showCard, hiddenCard, topCard : restCards)
      , deck = remainingDeck}
    playOutDealerHand
  else ...
```

Now all that's left is analyzing the end conditions. We'll score the player's hand and compare it to the dealer's. If the dealer has busted, or the player has the better score, we'll give a reward of 1.0. If they're the same, the reward is 0.5. Otherwise, the player loses. In all cases, we return the current observation and True as our "done" variable.

```
playOutDealerHand :: (Monad m) => StateT BlackjackEnvironment m (BlackjackObservation, Double, Bool)
playOutDealerHand = do
  bje <- get
  let (showCard, hiddenCard, restCards) = dealerHand bje
  currentDealerScore = scoreHand
    (showCard : hiddenCard : restCards)
  if currentDealerScore < 17
  then ...
  else do
    let playerScore = scoreHand (playerHand bje)
```

```
currentObs = currentObservation bje
if playerScore > currentDealerScore || currentDealerScore > 21
then return (currentObs, 1.0, True)
else if playerScore == currentDealerScore
then return (currentObs, 0.5, True)
else return (currentObs, 0.0, True)
```

ODDS AND ENDS

We'll also need code for running a loop and playing the game. But that code though looks very similar to what we used for Frozen Lake. This is a promising sign for our hopes to generalize this with a type class. Here's a sample playthrough of the game. As inputs, 0 means "hit" and 1 means "stand".

So in this first game, we start with a King and 9, and see the dealer has a 6 showing. We "stand", and the dealer busts.

```
6 X

K 9
19 # Our current score
1  # Stand command

1.0 # Reward
Episode Finished

6 9 8 # Dealer's final hand
23 # Dealer's final (busted) score

K 9
19
```

In this next example, we try to hit on 13, since the dealer has an Ace. We bust, and lose the game.

```
A X

3 J
```

```
13
0

0.0
Episode Finished

A X

K 3 J
23
```

CONCLUSION

Of course, there are a few ways we could make this more complicated. We could do iterated blackjack to allow card-counting. Or we could add advanced moves like splitting and doubling down. But that's not necessary for our purposes. The main point is that we have two fully functional games we can work with!

In part 4, we'll start digging into the machine learning process. We'll learn about Q-Learning with the Open Gym in Python and translate those ideas to Haskell.

We left out quite a bit of code in this example, particularly around setup. Take a look at Github to see all the details!

Revision #1

Created 11 March 2022 06:48:20 by gasick

Updated 11 March 2022 17:11:16 by gasick