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

Generalizing Our Environments

In part 4 of this series, we applied the ideas of Q-learning to both of our games. You can compare the implementations by looking at the code on Github: Frozen Lake and Blackjack. At this point, we've seen enough in common with these that we can make a general typeclass for them!

CONSTRUCTING A CLASS

Now if you look very carefully at the gameLoop and playGame code in both examples, it's nearly identical! We'd only need to make a few adjustments to naming types. This tells us we have a general structure between our different games. And we can capture that structure with a class.

Let's look at the common elements between our environments. These are all functions we call from the game loop or runner:

Resetting the environment Stepping the environment (with an action) Rendering the environment (if necessary) Apply some learning method on the new data Diminish the exploration rate So our first attempt at this class might look like this, looking only at the most important fields:

```
class Environment e where
  resetEnv :: (Monad m) => StateT e m Observation
  stepEnv :: (Monad m) => Action
    -> StateT e m (Observation, Double, Bool)
  renderEnv :: (MonadIO m) => StateT e m ()
  learnEnv :: (Monad m) =>
    Observation -> Observation -> Double -> Action -> StateT e m ()

instance Environment FrozenLakeEnvironment where
```

```
  ...

instance Environment BlackjackEnvironment where
  ...
```

We can make two clear observations about this class. First, we need to generalize the Observation and Action types! These are different in our two games and this isn't reflected above. Second, we're forcing ourselves to use the State monad over our environment. This isn't necessarily wise. It might force us to add extra fields to the environment type that don't belong there.

The solution to the first issue is to make this class a type family! Then we can associate the proper data types for observations and actions. The solution to the second issue is that our class should be over a monad instead of the environment itself.

Remember, a monad provides the context in which a computation takes place. So in our case, our game, with all its stepping and learning, is that context!

Doing this gives us more flexibility for figuring out what data should live in which types. It makes it easier to separate the game's internal state from auxiliary state, like the exploration rate.

Here's our second try, with associated types and a monad.

```
newtype Reward = Reward Double

class (MonadIO m) => EnvironmentMonad m where
  type Observation m :: *
  type Action m :: *
  resetEnv :: m (Observation m)
  currentObservation :: m (Observation m)
  stepEnv :: (Action m) -> m (Observation m, Reward, Bool)
  renderEnv :: m ()
  learnEnv ::
    (Observation m) -> (Observation m) ->
    Reward -> (Action m) -> m ()
  explorationRate :: m Double
  reduceExploration :: Double -> Double -> m ()
```

There are a couple undesirable parts of this. Our monad has to be IO to account for rendering. But it's possible for us to play the game without needing to render. In fact, it's also possible for us to

play the game without learning!

So we can separate this into more typeclasses! We'll have two "subclasses" of our Environment. We'll make a separate class for rendering. This will be the only class that needs an IO constraint. Then we'll have a class for learning functionality. This will allow us to "run" the game in different contexts and limit the reach of these effects.

```
newtype Reward = Reward Double

class (Monad m) => EnvironmentMonad m where
  type Observation m :: *
  type Action m :: *
  currentObservation :: m (Observation m)
  resetEnv :: m (Observation m)
  stepEnv :: (Action m) -> m (Observation m, Reward, Bool)

class (MonadIO m, EnvironmentMonad m) =>
  RenderableEnvironment m where
    renderEnv :: m ()

class (EnvironmentMonad m) => LearningEnvironment m where
  learnEnv ::
    (Observation m) -> (Observation m) ->
    Reward -> (Action m) -> m ()
  explorationRate :: m Double
  reduceExploration :: Double -> Double -> m ()
```

A COUPLE MORE FIELDS

There are still a couple extra pieces we can add that will make these classes more complete. One thing we're missing here is a concrete expression of our state. This makes it difficult to run our environments from normal code. So let's add a new type to the family for our "Environment" type, as well as a function to "run" that environment. We'll also want a generic way to get the current observation.

```
class (Monad m) => EnvironmentMonad m where
  type Observation m :: *
  type Action m :: *
  type EnvironmentState m :: *
  runEnv :: (EnvironmentState m) -> m a -> IO a
  currentObservation :: m (Observation m)
  resetEnv :: m (Observation m)
  stepEnv :: (Action m) -> m (Observation m, Reward, Bool)
```

Forcing run to use IO is more restrictive than we'd like. But it's trickier than you'd expect to have this function within our monad class.

We can also add one more item to our LearningEnvironment for choosing an action.

```
class (EnvironmentMonad m) => LearningEnvironment m where
  learnEnv ::
    (Observation m) -> (Observation m) -> Reward -> (Action m) -> m ()
  chooseActionBrain :: m (Action m)
  explorationRate :: m Double
  reduceExploration :: Double -> Double -> m ()
```

Now we're in pretty good shape! Let's try to use this class!

GAME LOOPS

In previous iterations, we had gameLoop functions for each of our different environments. We can now write these in a totally generic way! Here's a simple loop that plays the game once and produces a result:

```
gameLoop :: (EnvironmentMonad m) =>
  m (Action m) -> m (Observation m, Reward)
gameLoop chooseAction = do
  newAction <- chooseAction
  (newObs, reward, done) <- stepEnv newAction
  if done
    then return (newObs, reward)
    else gameLoop chooseAction
```

If we want to render the game between moves, we add a single renderEnv call before selecting the move. We also need an extra IO constraint and to render it before returning the final result.

```
gameRenderLoop :: (RenderableEnvironment m) =>
  m (Action m) -> m (Observation m, Reward)
gameRenderLoop chooseAction = do
  renderEnv
  newAction <- chooseAction
  (newObs, reward, done) <- stepEnv newAction
  if done
    then renderEnv >> return (newObs, reward)
    else gameRenderLoop chooseAction
```

Finally, there are a couple different loops we can write for a learning environment. We can have a generic loop for one iteration of the game. Notice how we rely on the class function chooseActionBrain. This means we don't need such a function as a parameter.

```
gameLearningLoop :: (LearningEnvironment m) =>
  m (Observation m, Reward)
gameLearningLoop = do
  oldObs <- currentObservation
  newAction <- chooseActionBrain
  (newObs, reward, done) <- stepEnv newAction
  learnEnv oldObs newObs reward newAction
  if done
    then return (newObs, reward)
    else gameLearningLoop
```

Then we can make another loop that runs many learning iterations. We reduce the exploration rate at a reasonable interval.

```
gameLearningIterations :: (LearningEnvironment m) => m [Reward]
gameLearningIterations = forM [1..numEpisodes] $ \i -> do
  resetEnv
  when (i `mod` 100 == 99) $ do
    reduceExploration decayRate minEpsilon
  (_, reward) <- gameLearningLoop
  return reward
  where
```

```
    numEpisodes = 10000
    decayRate = 0.9
    minEpsilon = 0.01
```

CONCRETE IMPLEMENTATIONS

Now we want to see how we actually implement these classes for our types. We'll show the examples for FrozenLake but it's an identical process for Blackjack. We start by defining the monad type as a wrapper over our existing state.

```
newtype FrozenLake a = FrozenLake (StateT FrozenLakeEnvironment IO a)
  deriving (Functor, Applicative, Monad)
```

We'll want to make a State instance for our monads over the environment type. This will make it easier to port over our existing code. We'll also need a MonadIO instance to help with rendering.

```
instance (MonadState FrozenLakeEnvironment) FrozenLake where
  get = FrozenLake get
  put fle = FrozenLake $ put fle

instance MonadIO FrozenLake where
  liftIO act = FrozenLake (liftIO act)
```

Then we want to change our function signatures to live in the desired monad. We can pretty much leave the functions themselves untouched.

```
resetFrozenLake :: FrozenLake FrozenLakeObservation

stepFrozenLake ::
  FrozenLakeAction -> FrozenLake (FrozenLakeObservation, Reward, Bool)

renderFrozenLake :: FrozenLake ()
```

Finally, we make the actual instance for the class. The only thing we haven't defined yet is the runEnv function. But this is a simple wrapper for evalStateT.

```
instance EnvironmentMonad FrozenLake where
  type (Observation FrozenLake) = FrozenLakeObservation
  type (Action FrozenLake) = FrozenLakeAction
  type (EnvironmentState FrozenLake) = FrozenLakeEnvironment
  baseEnv = basicEnv
  runEnv env (FrozenLake action) = evalStateT action env
  currentObservation = FrozenLake (currentObs <$> get)
  resetEnv = resetFrozenLake
  stepEnv = stepFrozenLake

instance RenderableEnvironment FrozenLake where
  renderEnv = renderFrozenLake
```

There's a bit more we could do. We could now separate the "brain" portions of the environment without any issues. We wouldn't need to keep the Q-Table and the exploration rate in the state. This would improve our encapsulation. We could also make our underlying monads more generic.

PLAYING THE GAME

Now, playing our game is simple! We get our basic environment, reset it, and call our loop function! This code will let us play one iteration of Frozen Lake, using our own input:

```
main :: IO ()
main = do
  (env :: FrozenLakeEnvironment) <- basicEnv
  _ <- runEnv env action
  putStrLn "Done!"
  where
    action = do
      resetEnv
      (gameRenderLoop chooseActionUser
        :: FrozenLake (FrozenLakeObservation, Reward))
```

Once again, we can make this code work for Blackjack with a simple name substitution.

We can also make this work with our Q-learning code as well. We start with a simple instance for LearningEnvironment.

```
instance LearningEnvironment FrozenLake where
  learnEnv = learnQTable
  chooseActionBrain = chooseActionQTable
  explorationRate = flExplorationRate <$> get
  reduceExploration decayRate minEpsilon = do
    fle <- get
    let e = flExplorationRate fle
    let newE = max minEpsilon (e * decayRate)
    put $ fle { flExplorationRate = newE }
```

And now we use gameLearningIterations instead of gameRenderLoop!

```
main :: IO ()
main = do
  (env :: FrozenLakeEnvironment) <- basicEnv
  _ <- runEnv env action
  putStrLn "Done!"
  where
    action = do
      resetEnv
      (gameLearningIterations :: FrozenLake [Reward])
```

You can see all the code for this on Github!

1. Generic Environment Classes
2. Frozen Lake with Environment
3. Blackjack with Environment
4. Playing Frozen Lake
5. Playing Blackjack

CONCLUSION

We're still pulling in two "extra" pieces besides the environment class itself. We still have specific implementations for basicEnv and action choosing. We could try to abstract these behind the class

as well. There would be generic functions for choosing the action as a human and choosing at random. This would force us to make the action space more general as well.

But for now, it's time to explore some more interesting learning algorithms. For our current Q-learning approach, we make a table with an entry for every possible game state. This doesn't scale to games with large or continuous observation spaces! In part 6, we'll see how TensorFlow allows us to learn a Q function instead of a direct table.

We'll start in Python, but soon enough we'll be using TensorFlow in Haskell. Take a look at our guide for help getting everything installed!

Revision #1
Created 2022-03-11 06:52:20 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick