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

Q-Learning with TensorFlow (Haskell)

In part 6 of the series, we used the ideas of Q-Learning together with TensorFlow. We got a more general solution to our agent that didn't need a table for every state of the game.

In this part, we'll take the final step and implement this TensorFlow approach in Haskell. We'll see how to integrate this library with our existing Environment system. It works out quite smoothly, with a nice separation between our TensorFlow logic and our normal environment logic! You can take a look at the code for this part on the tensorflow branch of our repo, particularly in FrozenLakeTensor.hs.

This article requires a working knowledge of the Haskell TensorFlow integration. If you're new to this, you should download our Guide showing how to work with this framework. You can also read our original Machine Learning Series for some more details! In particular, the second part will go through the basics of tensors.

BUILDING OUR TF MODEL

The first thing we want to do is construct a "model". This model type will store three items. The first will be the tensor for the weights we have. Then the second two will be functions in the TensorFlow Session monad. The first function will provide scores for the different moves in a position, so we can choose our move. The second will allow us to train the model and update the weights.

```
data Model = Model
  {  weightsT :: Variable Float
  , chooseActionStep :: TensorData Float -> Session (Vector Float)
  , learnStep :: TensorData Float -> TensorData Float -> Session ()
```

```
  }
```

The input for choosing an action is our world observation state, converted to a Float and put in a size 16-vector. The result will be 4 floating point values for the scores. Then our learning step will take in the observation as well as a set of 4 values. These are the "target" values we're training our model on.

We can construct our model within the Session monad. In the first part of this process we define our weights and use them to determine the score of each move (results).

```
createModel :: Session Model
createModel = do
  -- Choose Action
  inputs <- placeholder (Shape [1, 16])
  weights <- truncatedNormal (vector [16, 4]) >>= initializedVariable
  let results = inputs `matMul` readValue weights
  returnedOutputs <- render results
  ...
```

Now we make our "trainer". Our "loss" function is the reduced, squared difference between our results and the "target" outputs. We'll use the adam optimizer to learn values for our weights to minimize this loss.

```
createModel :: Session Model
createModel = do
  -- Choose Action
  ...

  -- Train Nextwork
  (nextOutputs :: Tensor Value Float) <- placeholder (Shape [4, 1])
  let (diff :: Tensor Build Float) = nextOutputs `sub` results
  let (loss :: Tensor Build Float) = reduceSum (diff `mul` diff)
  trainer_ <- minimizeWith adam loss [weights]
  ...
```

Finally, we wrap these tensors into functions we can call using runWithFeeds. Recall that each feed provides us with a way to fill in one of our placeholder tensors.

```
createModel :: Session Model
createModel = do
  -- Choose Action
  ...

  -- Train Network
  ...

  -- Create Model
  let chooseStep = \inputFeed ->
        runWithFeeds [feed inputs inputFeed] returnedOutputs
  let trainStep = \inputFeed nextOutputFeed ->
        runWithFeeds [ feed inputs inputFeed
                     , feed nextOutputs nextOutputFeed
                     ]
                     trainer_
  return $ Model weights chooseStep trainStep
```

Our model now wraps all the different tensor operations we need! All we have to do is provide it with the correct TensorData. To see how that works, let's start integrating with our EnvironmentMonad!

INTEGRATING WITH ENVIRONMENT

Our model's functions exist within the TensorFlow monad Session. So how then, do we integrate this with our existing Environment code? The answer is, of course, to construct a new monad! This monad will wrap Session, while still giving us our FrozenLakeEnvironment! We'll keep the environment within a State, but we'll also keep a reference to our Model.

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

instance (MonadState FrozenLakeEnvironment) FrozenLake where
```

```
  get = FrozenLake (fst <$> get)
  put fle = FrozenLake $ do
    (_, model) <- get
    put (fle, model)
```

Now we can start implementing the actual EnvironmentMonad instance. Most of our existing types and functions will work with trivial modification. The only real change is that runEnv will need to run a TensorFlow session and create the model. Then it can use evalStateT.

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

This is all we need to define the first class. But, with TensorFlow, our environment is only useful if we use the tensor model! This means we need to fill in LearningEnvironment as well. This has two functions, chooseActionBrain and learnEnv using our tensors. Let's see how that works.

CHOOSING AN ACTION

Choosing an action is straightforward. We'll once again start with the same format for sometimes choosing a random move:

```
chooseActionTensor :: FrozenLake FrozenLakeAction
chooseActionTensor = FrozenLake $ do
  (fle, model) <- get
  let (exploreRoll, gen') = randomR (0.0, 1.0) (randomGenerator fle)
  if exploreRoll < flExplorationRate fle
    then do
      let (actionRoll, gen'') = Rand.randomR (0, 3) gen'
```

```
      put $ (fle { randomGenerator = gen'' }, model)
      return (toEnum actionRoll)
    else do
      ...
```

As in Python, we'll need to convert an observation to a tensor type. This time, we'll create TensorData. This type wraps a vector, and our input should have the size 1x16. It has the format of a oneHot tensor. But it's easier to make this a pure function, rather than using a TensorFlow monad.

```
obsToTensor :: FrozenLakeObservation -> TensorData Float
obsToTensor obs = encodeTensorData (Shape [1, 16]) (V.fromList asList)
  where
    asList = replicate (fromIntegral obs) 0.0 ++
             [1.0] ++
             replicate (fromIntegral (15 - obs)) 0.0
```

Since we've already defined our chooseAction step within the model, it's easy to use this! We convert the current observation, get the result values, and then pick the best index!

```
chooseActionTensor :: FrozenLake FrozenLakeAction
chooseActionTensor = FrozenLake $ do
  (fle, model) <- get
  -- Random move
  ...
    else do
      let obs1 = currentObs fle
      let obs1Data = obsToTensor obs1

      -- Use model!
      results <- lift ((chooseActionStep model) obs1Data)
      let bestMoveIndex = V.maxIndex results
      put $ (fle { randomGenerator = gen' }, model)
      return (toEnum bestMoveIndex)
```

LEARNING FROM THE ENVIRONMENT

One unfortunate part of our current design is that we have to repeat some work in our learning function. To learn from our action, we need to use all the values, not just the chosen action. So to start our learning function, we'll call chooseActionStep again. This time we'll get the best index AND the max score.

```
learnTensor ::
  FrozenLakeObservation -> FrozenLakeObservation ->
  Reward -> FrozenLakeAction ->
  FrozenLake ()
learnTensor obs1 obs2 (Reward reward) action = FrozenLake $ do
  model <- snd <$> get
  let obs1Data = obsToTensor obs1

  -- Use the model!
  results <- lift ((chooseActionStep model) obs1Data)
  let (bestMoveIndex, maxScore) =
        (V.maxIndex results, V.maximum results)
  ...
```

We can now get our "target" values by substituting in the reward and max score at the proper index. Then we convert the second observation to a tensor, and we have all our inputs to call our training step!

```
learnTensor ::
  FrozenLakeObservation -> FrozenLakeObservation ->
  Reward -> FrozenLakeAction ->
  FrozenLake ()
learnTensor obs1 obs2 (Reward reward) action = FrozenLake $ do
  ...
  let (bestMoveIndex, maxScore) =
        (V.maxIndex results, V.maximum results)
  let targetActionValues = results V.//
```

```
        [(bestMoveIndex, double2Float reward + (gamma * maxScore))]
  let obs2Data = obsToTensor obs2
  let targetActionData = encodeTensorData
        (Shape [4, 1])
        targetActionValues

  -- Use the model!
  lift $ (learnStep model) obs2Data targetActionData

  where
    gamma = 0.81
```

Using these two functions, we can now fill in our LearningEnvironment class!

```
instance LearningEnvironment FrozenLake where
  chooseActionBrain = chooseActionTensor
  learnEnv = learnTensor
  -- Same as before
  explorationRate = ..
  reduceExploration = ...
```

We'll then be able to run this code just as we would our other Q-learning examples!

CONCLUSION

There's one more part to this series. In the eighth and final installment, we'll compare our current setup and the Gloss library. Gloss offers much more extensive possibilities for rendering our game and accepting input. So using it would expand the range of games we could play!

Revision #1
Created 2022-03-11 06:54:47 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick