Published in Towards Data Science

Adam King Follow
Apr 10, 2019 · 8 min read · · Listen

Save

Create custom gym environments from scratch — A stock market example

OpenAI's `gym` is an awesome package that allows you to create custom reinforcement learning agents. It comes with quite a few pre-built environments like CartPole, MountainCar, and a ton of free Atari games to experiment with.

These environments are great for learning, but eventually you'll want to setup an agent to solve a custom problem. To do this, you'll need to create a custom environment, specific to your problem domain. Later, we will create a custom stock market environment for simulating stock trades. All of the code for this article will be available on my **GitHub**.

First, let's learn about what exactly an environment is. An environment contains all the necessary functionality to run an agent and allow it to learn. Each environment must implement the following *gym* interface:

```
import gym
from gym import spaces

class CustomEnv(gym.Env):
  """Custom Environment that follows gym interface"""
  metadata = {'render.modes': ['human']}

  def __init__(self, arg1, arg2, ...):
    super(CustomEnv, self).__init__()

    # Define action and observation space
    # They must be gym.spaces objects
  def step(self, action):
    # Execute one time step within the environment
    ...

  def reset(self):
    # Reset the state of the environment to an initial state
    ...

  def render(self, mode='human', close=False):
    # Render the environment to the screen
    ...
```

In the constructor, we first define the type and shape of our `action_space`, which will contain all of the actions possible for an agent to take in the environment. Similarly, we'll define the `observation_space`, which contains all of the environment's data to be observed by the agent.

Our `reset` method will be called to periodically reset the environment to an initial state. This is followed by many `step`s through the environment, in which an action will be provided by the model and must be executed, and the next observation returned. This is also where rewards are calculated, more on this later.

Finally, the `render` method may be called periodically to print a rendition of the environment. This could be as simple as a print statement, or as complicated as rendering a 3D environment using openGL. For this example, we will stick with print statements.

Stock Trading Environment

To demonstrate how this all works, we are going to create a stock trading environment. We will then train our agent to become a profitable trader within the environment. Let's get started!

The first thing we'll need to consider is how a human trader would perceive their environment. What observations would they make before deciding to make a trade?

A trader would most likely look at some charts of a stock's price action, perhaps overlaid with a couple technical indicators. From there, they would combine this visual information with their prior knowledge of similar price action to make an informed decision of which direction the stock is likely to move.

So let's translate this into how our agent should perceive its environment.

Our `observation_space` contains all of the input variables we want our agent to consider before making, *or not making* a trade. In this example, we want our agent to "see" the stock data points (open price, high, low, close, and daily volume) for the last five days, as well a couple other data points like its account balance, current stock positions, and current profit.

The intuition here is that for each time step, we want our agent to consider the price action leading up to the current price, as well as their own portfolio's status in order to make an informed decision for the next action.

But this isn't enough; we need to know the amount of a given stock to buy or sell each time. Using gym's `Box` space, we can create an action space that has a discrete number of action types (buy, sell, and hold), as well as a continuous spectrum of amounts to buy/sell (0-100% of the account balance/position size respectively).

You'll notice the amount is not necessary for the hold action, but will be provided anyway. Our agent does not initially know this, but over time should learn that the amount is extraneous for this action.

The last thing to consider before implementing our environment is the reward. We want to incentivize profit that is sustained over long periods of time. At each step, we will set the reward to the account balance multiplied by some fraction of the number of time steps so far.

The purpose of this is to delay rewarding the agent too fast in the early stages and allow it to explore sufficiently before optimizing a single strategy too deeply. It will also reward agents that maintain a higher balance for longer, rather than those who rapidly gain money using unsustainable strategies.

Implementation

Now that we've defined our observation space, action space, and rewards, it's time to implement our environment. First, we need define the `action_space` and `observation_space` in the

```
  metadata = {'render.modes': ['human']}

  def __init__(self, df):
    super(StockTradingEnv, self).__init__()
    self.df = df
    self.reward_range = (0, MAX_ACCOUNT_BALANCE)

    # Actions of the format Buy x%, Sell x%, Hold, etc.
    self.action_space = spaces.Box(
      low=np.array([0, 0]), high=np.array([3, 1]), dtype=np.float16)

    # Prices contains the OHCL values for the last five prices
    self.observation_space = spaces.Box(
      low=0, high=1, shape=(6, 6), dtype=np.float16)
```

Next, we'll write the `reset` method, which is called any time a new environment is created or to reset an existing environment's state. It's here where we'll set the starting balance of each agent and initialize its open positions to an empty list.

```
def reset(self):
  # Reset the state of the environment to an initial state
  self.balance = INITIAL_ACCOUNT_BALANCE
  self.net_worth = INITIAL_ACCOUNT_BALANCE
  self.max_net_worth = INITIAL_ACCOUNT_BALANCE
  self.shares_held = 0
  self.cost_basis = 0
  self.total_shares_sold = 0
  self.total_sales_value = 0

  # Set the current step to a random point within the data frame
  self.current_step = random.randint(0, len(self.df.loc[:, 'Open'].values) -
6)

  return self._next_observation()
```

We set the current step to a random point within the data frame, because it essentially gives our agent's more unique experiences from the same data set. The `_next_observation` method compiles the stock data for the last five time steps, appends the agent's account information, and scales all the values to between 0 and 1.

```
def _next_observation(self):
  # Get the data points for the last 5 days and scale to between 0-1
  frame = np.array([
    self.df.loc[self.current_step: self.current_step +
                5, 'Close'].values / MAX_SHARE_PRICE,
    self.df.loc[self.current_step: self.current_step +
                5, 'Volume'].values / MAX_NUM_SHARES,
   ])

  # Append additional data and scale each value to between 0-1
  obs = np.append(frame, [[
    self.balance / MAX_ACCOUNT_BALANCE,
    self.max_net_worth / MAX_ACCOUNT_BALANCE,
    self.shares_held / MAX_NUM_SHARES,
    self.cost_basis / MAX_SHARE_PRICE,
    self.total_shares_sold / MAX_NUM_SHARES,
    self.total_sales_value / (MAX_NUM_SHARES * MAX_SHARE_PRICE),
  ]], axis=0)

  return obs
```

Next, our environment needs to be able to take a `step`. At each step we will take the specified action (chosen by our model), calculate the reward, and return the next observation.

```
def step(self, action):
  # Execute one time step within the environment
  self._take_action(action)

  self.current_step += 1

  if self.current_step > len(self.df.loc[:, 'Open'].values) - 6:
    self.current_step = 0

  delay_modifier = (self.current_step / MAX_STEPS)

  reward = self.balance * delay_modifier
  done = self.net_worth <= 0

  obs = self._next_observation()

  return obs, reward, done, {}
```

Now, our `_take_action` method needs to take the action provided by the model and either buy, sell, or hold the stock.

```
def _take_action(self, action):
  # Set the current price to a random price within the time step
  if action_type < 1:
    # Buy amount % of balance in shares
    total_possible = self.balance / current_price
    shares_bought = total_possible * amount
    prev_cost = self.cost_basis * self.shares_held
    additional_cost = shares_bought * current_price

    self.balance -= additional_cost
    self.cost_basis = (prev_cost + additional_cost) /
                            (self.shares_held + shares_bought)
    self.shares_held += shares_bought

  elif actionType < 2:
    # Sell amount % of shares held
    shares_sold = self.shares_held * amount .
    self.balance += shares_sold * current_price
    self.shares_held -= shares_sold
    self.total_shares_sold += shares_sold
    self.total_sales_value += shares_sold * current_price

  self.netWorth = self.balance + self.shares_held * current_price

  if self.net_worth > self.max_net_worth:
    self.max_net_worth = net_worth

  if self.shares_held == 0:
    self.cost_basis = 0
```

The only thing left to do now is `render` the environment to the screen. For simplicity's sake, we will just render the profit made so far and a couple other interesting metrics.

```
def render(self, mode='human', close=False):
  # Render the environment to the screen
  profit = self.net_worth - INITIAL_ACCOUNT_BALANCE

  print(f'Step: {self.current_step}')
  print(f'Balance: {self.balance}')
  print(f'Shares held: {self.shares_held}
          (Total sold: {self.total_shares_sold})')
  print(f'Avg cost for held shares: {self.cost_basis}
          (Total sales value: {self.total_sales_value})')
  print(f'Net worth: {self.net_worth}
          (Max net worth: {self.max_net_worth})')
  print(f'Profit: {profit}')
```

Our environment is complete. We can now instantiate a `StockTradingEnv` environment with a

```
import datetime as dt

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import PPO2

from env.StockTradingEnv import StockTradingEnv

import pandas as pd

df = pd.read_csv('./data/AAPL.csv')
df = df.sort_values('Date')

# The algorithms require a vectorized environment to run
env = DummyVecEnv([lambda: StockTradingEnv(df)])

model = PPO2(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=20000)

obs = env.reset()
for i in range(2000):
  action, _states = model.predict(obs)
  obs, rewards, done, info = env.step(action)
  env.render()
```

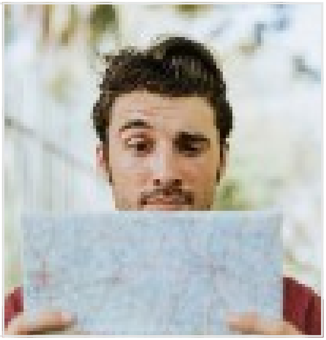
Now of course, this was all just for fun to test out creating an interesting, custom gym environment with some semi-complex actions, observations, and reward spaces. It's going to take a lot more time and effort if we really want to get rich with deep learning in the stock market…

Stay tuned for next week's article where we'll learn to create simple, yet elegant visualizations of our environments!

Creating Bitcoin trading bots that don't lose money

Let's make profitable cryptocurrency trading agents using deep reinforcement learning

towardsdatascience.com



• • •

Thanks for reading! As always, all of the code for this tutorial can be found on my GitHub. Leave a

Sponsor @notadamking on GitHub Sponsors

Hi, I'm Adam. I'm a developer, writer, and entrepreneur, specifically interested in financial applications of deep...

github.com

> *Github Sponsors is currently matching all donations 1:1 up to $5,000!*

Adam King is creating World Changing Content | Patreon

Hi, I'm Adam. I'm a developer, writer, and entrepreneur, specifically interested in financial applications of deep...

patreon.com