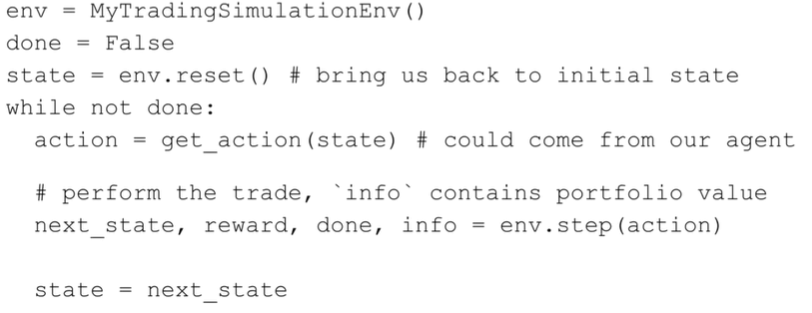
1. RL trader section introduction

* Usually, when people think about applying ML to the stock market
* They think about predicting the value of a stock
* Or even just the direction (will it go up/down?)
* This information by itself doesn’t make anything happen
* You must still physically sit down at your computer and make the trade
* Will you act on your model’s predictions?
* Maybe, but it’s up to you
* (un)supervised learning
* You get a prediction, but you must still take the action
* RL
* The agent takes an action which it believes will maximize the reward
* Rough outline
* You probably think of stock prices as a time series
* That sounds more like a prediction problem for RNN rather than RL
* How can we turn it into a RL problem?
* A matter of perspective
* Consider you are using some stock trading API
* api.buy(‘GOOG’, 10)
* If each share is $50, then $500 will be deducted from my bank account, an I will now own 10 shares of GOOG
* api.sell(‘AAPL’, 5)
* If each share is $30, then I will receive $150 in my bank account, and I also own 5 less APPL shares
* The act of calling the functions are actions an RL agent can perform
* The environment is the actual stock market
* There’s inherent randomness – you can’t predict tomorrow’s stock prices
* This has all the ingredients of an RL problem
* Actions = buy/sell/hold
* State = stock prices / #shares owned / amount of cash I have
* Reward = some function of portfolio value gained / lost
* Are you a RL agent?
* What do you do when you try to decide whether you want to buy / sell stocks?
* You try to follow the rule ‘buy low, sell high’
* Of course, you can’t see the future! You don’t know the current value is a dip or a peak

1. Data and environment

* Environment description
* We will work with historical stock data – this is a simulation
* How do we build an environment object to simulate this env
* Environment API



* In general, no matter what environment you use, it will look similar to that
* What are the state variables?
* What are the actions?
* What is the reward?
* Endless number of possibilities
* Need to simplify the problem
* State
* Many options
* Consider a time series (fixed window) of past and current stock prices
* Do we have enough cash to buy the stocks we want to buy?
* Given the prices of existing shares I own, is it worth selling them so I can buy more of a different stock?
* Let’s borrow some ideas from ‘Practical Deep Reinforcement Learning Approach for Stock Trading’
* They used a more advanced algorithm called DDPG
* State will consist of 3 parts
  + #1 – how many shares of each stock I own

E.g. [3, 5, 7] – 3 shares of APPL, 5 shares of MSI (Motorola), 7 shares of SBUX (Starbucks)

* + #2 – current price of each stock

E.g. [50, 20, 30] – APPL = $50/share, MSI = $20/share, SBUX = $30/share

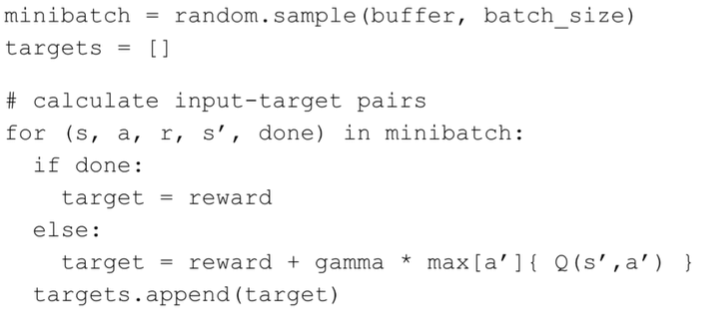
* + #3 – how much cash we have (uninvested)

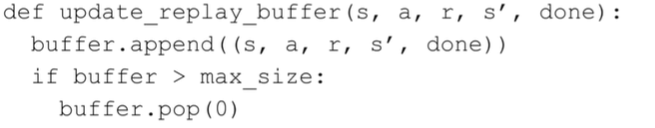
E.g. $100 – now my full state vector is [3,5,7,50,20,30,100]

* If we have N stocks, then the state will contain 2N + 1 components
* Actions:
* Many options to consider
* For any stock, I can buy / sell / hold
* In our environment, we will have 3 stocks to consider: APPL, MSI, SBUX
* Thus, we’d have 27 possibilities
  + E.g. [sell, sell, sell] – sell everything
  + E.g. [buy, sell, hold] – buy APPL, sell MSI, hold SBUX
* This still doesn’t tell us how many sell/buy
* Actions (simplified)
* Ignore transaction costs (usually this would cost ~$10 on your bank’s investing platform)
* 1) If we choose to sell -> sell all shares of that stock we own
* 2) If we buy -> buy as many as possible
* What does ‘as many as possible’ mean?
* Knapsack problem – could get complicated
* Of course, we don’t want that – it is NP hard
* Round robin fashion – loop through every stock, and buy 1 share until we run out of money
* 3) Sell before buy
* One action in our environment will involve performing all of these steps at once
* Reward
* Change in value of portfolio from one step(state s) to the next (state s’)
* How will we calculate the value of the portfolio?
* E.g.
  + we own 10 shares of AAPL, 5 shares of MSI, 3 shares of SBUX
  + share prices: APPL - $50, MSI - $20, SBUX - $30
  + cash = $100
  + value = 10\*50 + 5\*20 + 3\*30 + 100 = $790
* s = vector of #shares owned
* p = vector of share prices
* c = cash
* Summary
* Environment mimics the OpenAI Gym API
* State = env.reset(), next\_state, reward, done, info = env.step(action)
* We’ll consider 3 stocks: AAPL, MSI, SBUX
* State = [#shares owned, share prices, cash]
* Actions: buy, sell, hold
* All-or-nothing: if we buy, buy as much as we can, if we sell, sell all shares
* For 3 stocks, we have
* For N stocks, we have
* Reward = change in value of portfolio
* Portfolio value = # shares \* share price + cash

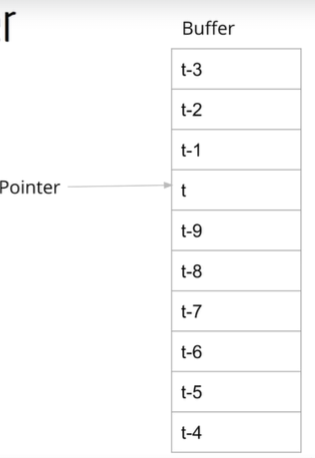
1. Replay buffer

* Efficient replay buffer / memory
* Naïve replay buffer
* Essentially a python list
* Each element will be a tuple of (s, a, r, s’, done) – transition
* Add new tuples as we encounter them
* Max size – when reaching max size, throw out oldest value
* Buffer.pop(0)
* Using the buffer (during training)

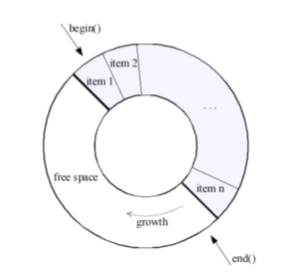




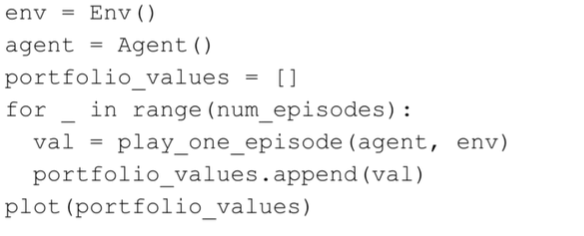
* Looks ok, but leads to memory leak
* Won’t be a huge issue with our modest-sized dataset, but problems will arise in more advanced applications
* Implement your own replay buffer
* Pre-allocate arrays to store
  + States (N x D array)
  + Actions (N array)
  + Rewards (N array)
  + Next states (N x D array)
  + Done flags (N array)
* We will never allocate more arrays, nor remove any existing arrays
* A pointer will tell us where to store the next value
* Our buffer is circular – when we reach the end, go back to beginning

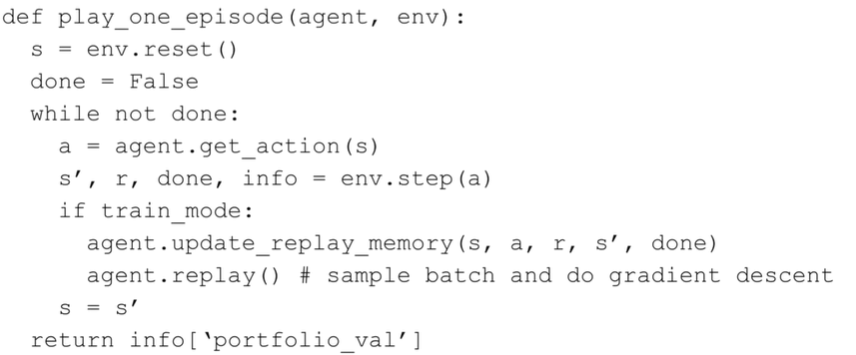


* What if the array is not yet full? We must keep track of the size of the buffer
* Max size may be 10, but if we have only stored 5 values so far, then the size is 5
* When we sample a minibatch, we can only sample from the first 5 elements
* Summary
* Avoid naïve approach of storing transitions in a list and popping off the oldest values
* Instead, use fixed-sized arrays, and store transitions in a circular fashion

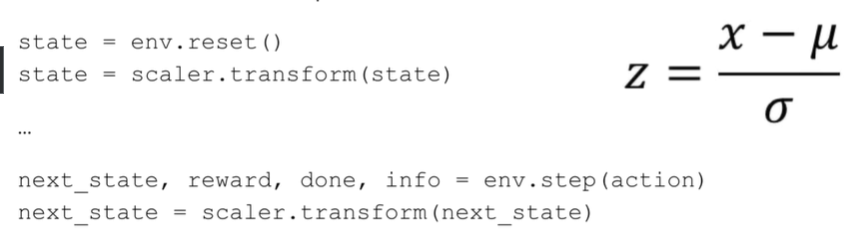


1. Program design and layout

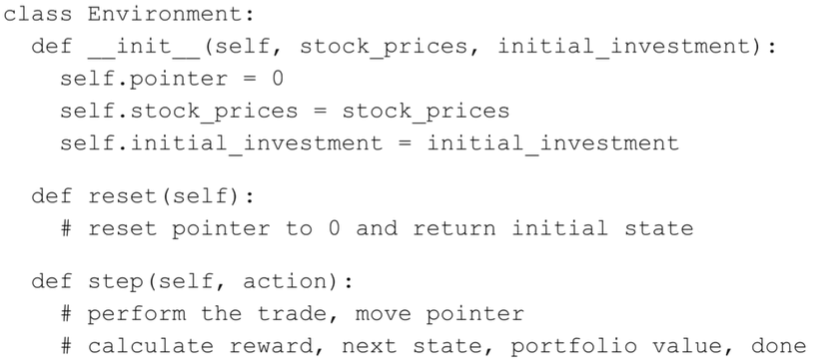
* RL trader layout and design
* 2 modes of operation: train and test
* Train data must come before test data
* Using the environment – organize the majority of our code
* Play\_one\_episode



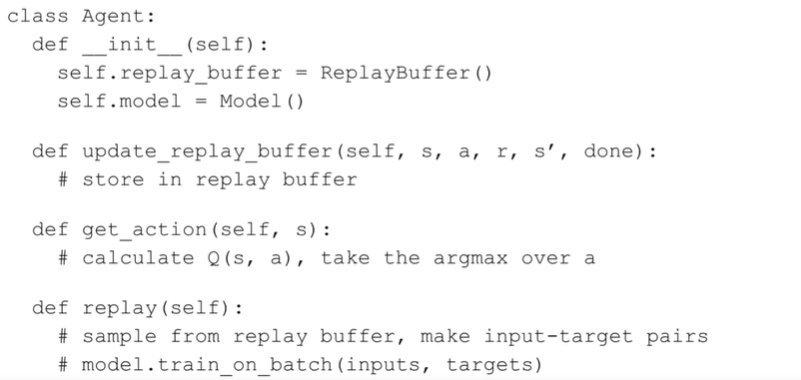
* Normalizing data
* Different parts of state have different ranges
* #shares, owned, stock prices, cash



* Environment



* Agent



* Summary
* Train mode and test mode
* Play one episode (again and again)
* Environment – takes in actions, produces states and rewards
* Agent – takes in state, produces action
* During training, store states / actions / rewards, update Q(s, a)

1. RL stock trader discussion

* This agent always makes a profit, does that mean it is really a good agent?
* What is our baseline?
* Compare it to an agent that takes completely random actions
* Equivalent to epsilon = 1
* Most of the distribution is > initial investment >< there is still a not-insignificant chance of losing money
* How does our agent compare?
* Our agent can beat random actions most of the time
* Still sensitive to hyperparameters
* How can a random strategy profit?
* Consider the dataset
* APPL, MSI, SBUX are all generally well-performing
* You would have to bee pretty unlucky to lose money
* To really test the algorithm, choose stocks that go up and down
* Synthetic datasets are very useful here!
* Project extensions
* Incorporate metadata (e.g., news and Twitter sentiment)
* Incorporate past stock price values into the state
  + No concept of the movement of the stock price
  + State = [#shares owned, current stock price, cash]
* Use returns instead of prices
  + Neural networks for regression are not great at extrapolation
  + Observe this yourself with synthetic data – check predictions outside training range
  + If stock prices are always going up – test vs. train very different
  + Returns are more stationary