Experimenting with FinTech using the AI-Gym RL Trading Algorithms

Madhavi Patel (1147984)

Objective:

In this project we are going to investigate two trading algorithm to make a trading decision.

Installing the Library

- Python
- Pandas
- Stable-baselines
- TensorFlow=1.15.0
- matplotlib
- mpl_finance

Action :

Step 1 –

Collected data from Microsoft

Step 2 –

Gym environment create by using reference given by professor

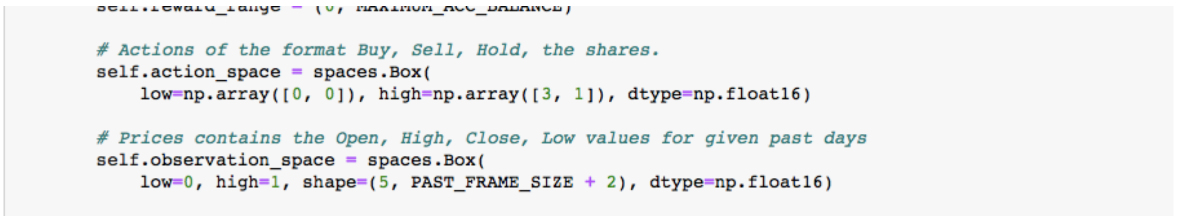
First we will implementing _init__ function of our environment class,
The observation and action spaces will be defined.

Observation Space –

The observation space can be discrete or continuous. .It is defined by cells, and the agent could be inside one of those cells, which contains all of the data in the environment that the agent would then observe.

Action Space –

In addition, the action space can be either continuous or discrete. We define the type and shape of our action space in the constructor, which will contain all of the actions which an agent could really take in the environment.

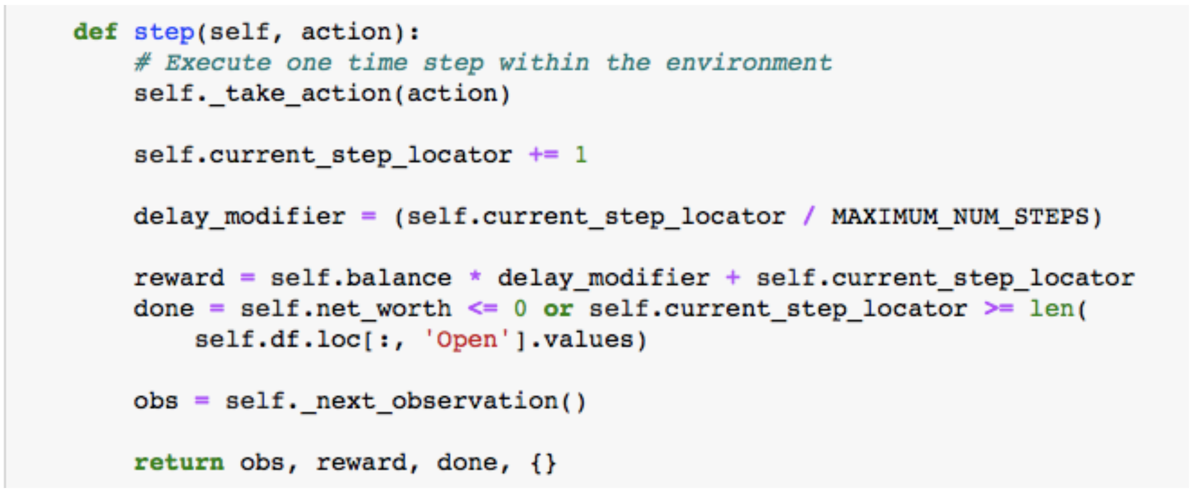


```
    self.reward_range = (0, MAXIMUM_ACC_BALANCE)

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

    # Prices contains the Open, High, Close, Low values for given past days
    self.observation_space = spaces.Box(
        low=0, high=1, shape=(5, PAST_FRAME_SIZE + 2), dtype=np.float16)
```

After that we will define functions in Env class

Step –

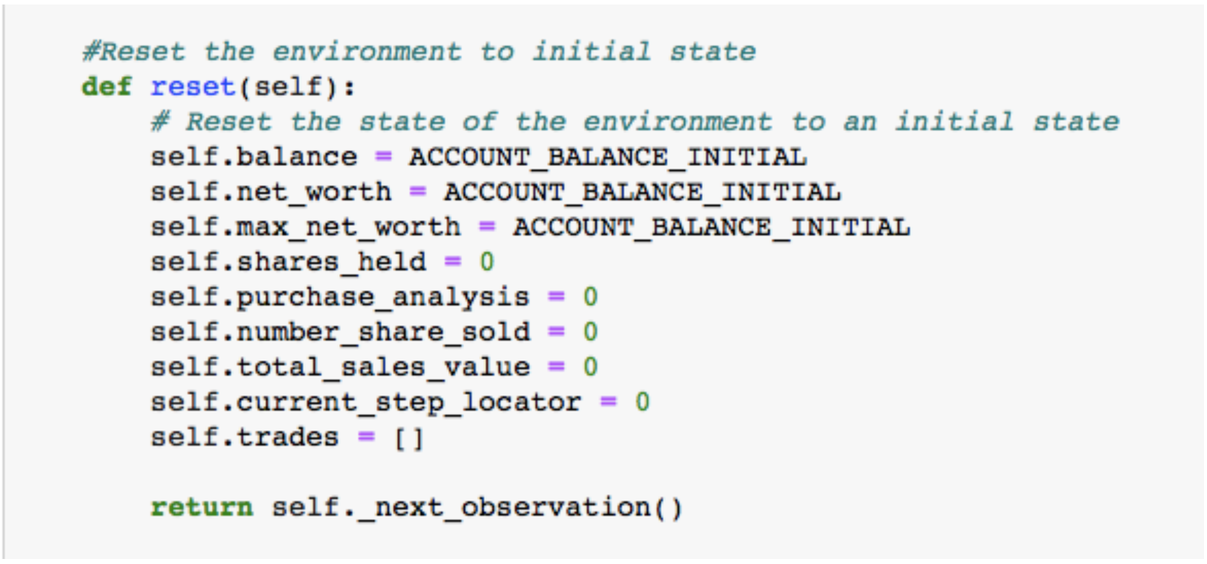
Applying an action to a step in the environment executes it. The new observation, reward, completion status, and other information are returned.



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

    self.current_step_locator += 1

    delay_modifier = (self.current_step_locator / MAXIMUM_NUM_STEPS)

    reward = self.balance * delay_modifier + self.current_step_locator
    done = self.net_worth <= 0 or self.current_step_locator >= len(
        self.df.loc[:, 'Open'].values)

    obs = self._next_observation()

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

Reset –

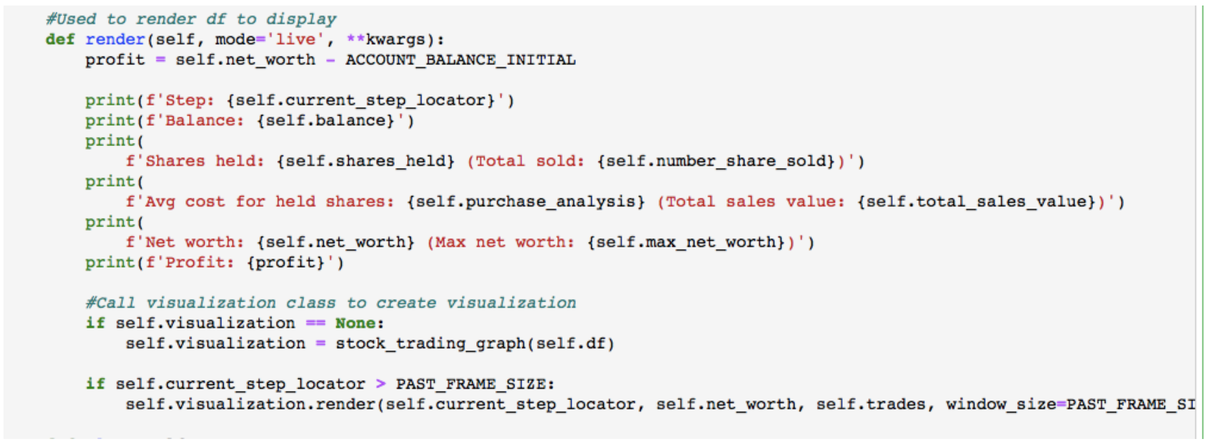
Returns the initial observation after resetting the environment to its initial state.



```
#Reset the environment to initial state
def reset(self):
    # Reset the state of the environment to an initial state
    self.balance = ACCOUNT_BALANCE_INITIAL
    self.net_worth = ACCOUNT_BALANCE_INITIAL
    self.max_net_worth = ACCOUNT_BALANCE_INITIAL
    self.shares_held = 0
    self.purchase_analysis = 0
    self.number_share_sold = 0
    self.total_sales_value = 0
    self.current_step_locator = 0
    self.trades = []

    return self._next_observation()
```

Render –

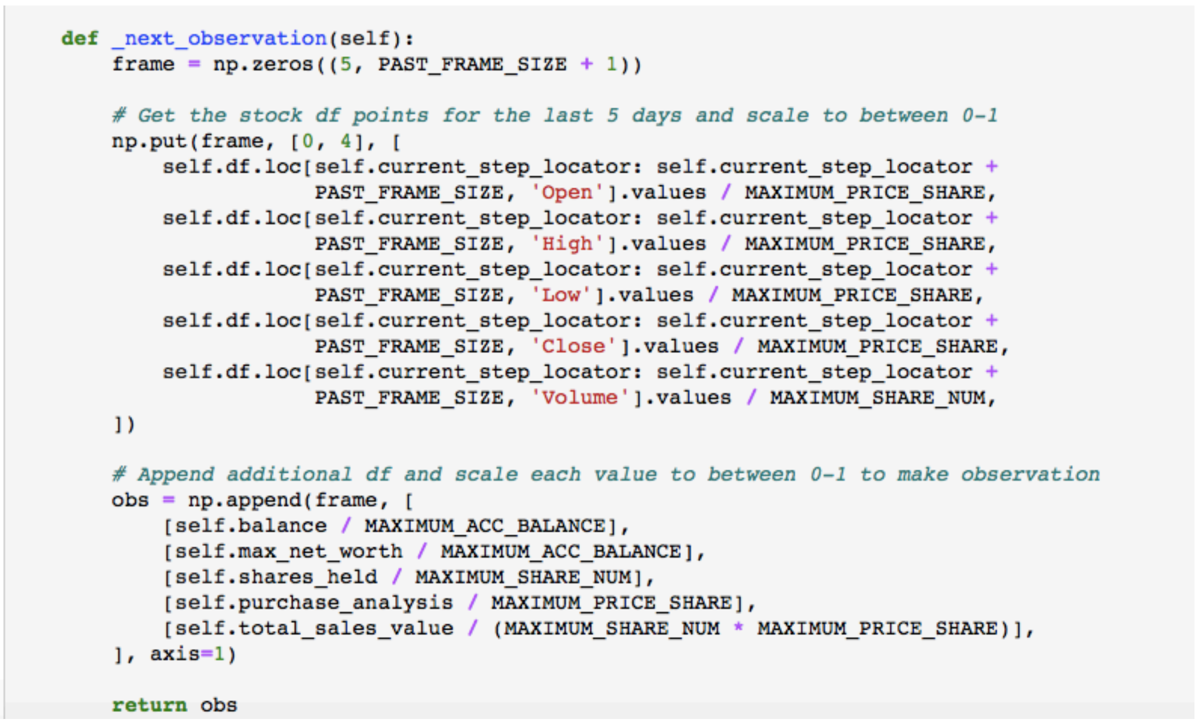
The render method can be used to print a rendition of the environment on a regular basis. It could be as straightforward as a print statement.



```
#Used to render df to display
def render(self, mode='live', **kwargs):
    profit = self.net_worth - ACCOUNT_BALANCE_INITIAL

    print(f'Step: {self.current_step_locator}')
    print(f'Balance: {self.balance}')
    print(
        f'Shares held: {self.shares_held} (Total sold: {self.number_share_sold})')
    print(
        f'Avg cost for held shares: {self.purchase_analysis} (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}')

    #Call visualization class to create visualization
    if self.visualization == None:
        self.visualization = stock_trading_graph(self.df)

    if self.current_step_locator > PAST_FRAME_SIZE:
        self.visualization.render(self.current_step_locator, self.net_worth, self.trades, window_size=PAST_FRAME_SI
```

_next_observation –

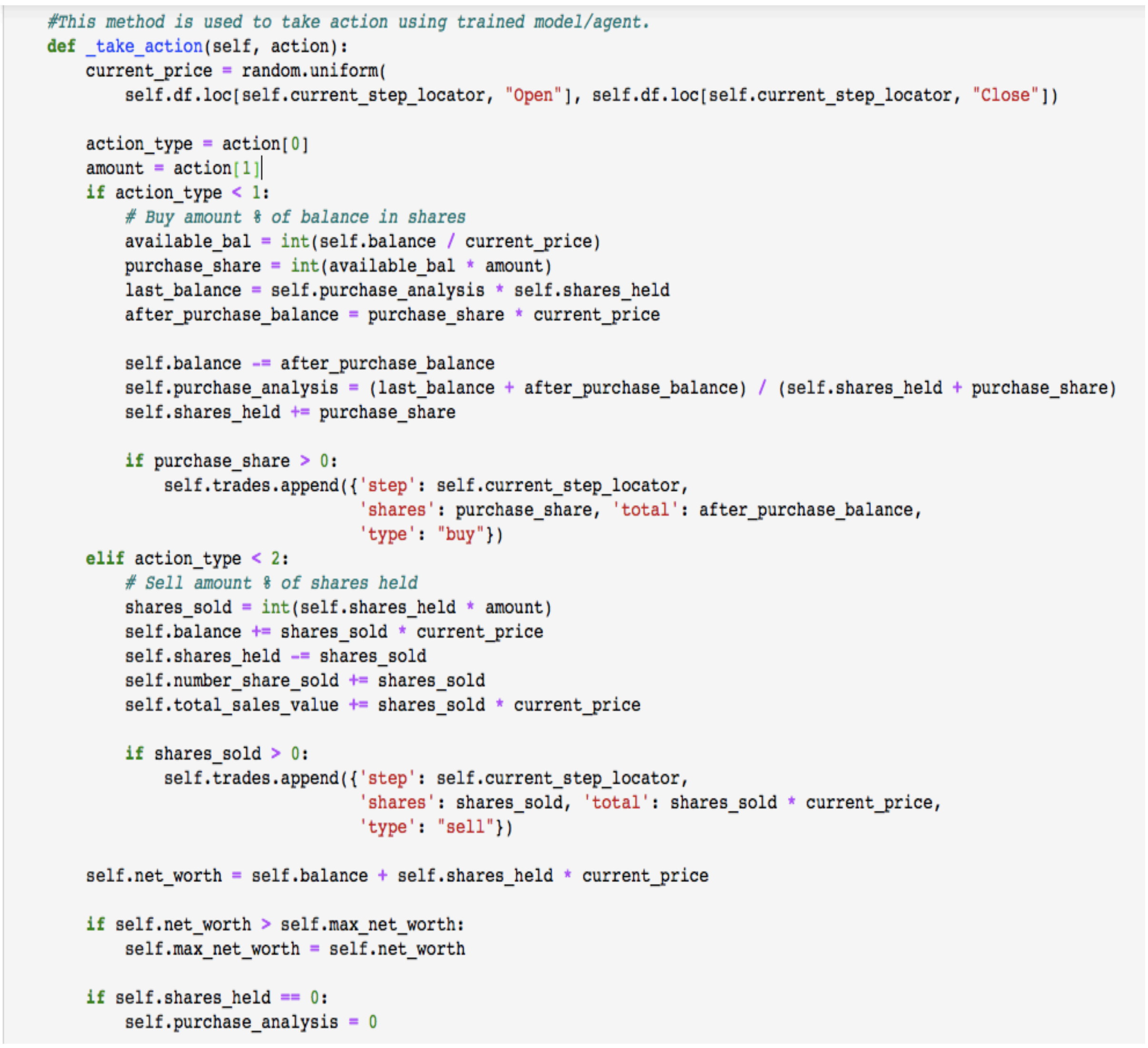
The _next observation method gathers stock data from the previous five days, adds the agent's account information, and scales all values to between 0 and 1.



```
def _next_observation(self):
    frame = np.zeros((5, PAST_FRAME_SIZE + 1))

    # Get the stock df points for the last 5 days and scale to between 0-1
    np.put(frame, [0, 4], [
        self.df.loc[self.current_step_locator: self.current_step_locator +
                    PAST_FRAME_SIZE, 'Open'].values / MAXIMUM_PRICE_SHARE,
        self.df.loc[self.current_step_locator: self.current_step_locator +
                    PAST_FRAME_SIZE, 'High'].values / MAXIMUM_PRICE_SHARE,
        self.df.loc[self.current_step_locator: self.current_step_locator +
                    PAST_FRAME_SIZE, 'Low'].values / MAXIMUM_PRICE_SHARE,
        self.df.loc[self.current_step_locator: self.current_step_locator +
                    PAST_FRAME_SIZE, 'Close'].values / MAXIMUM_PRICE_SHARE,
        self.df.loc[self.current_step_locator: self.current_step_locator +
                    PAST_FRAME_SIZE, 'Volume'].values / MAXIMUM_SHARE_NUM,
    ])

    # Append additional df and scale each value to between 0-1 to make observation
    obs = np.append(frame, [
        [self.balance / MAXIMUM_ACC_BALANCE],
        [self.max_net_worth / MAXIMUM_ACC_BALANCE],
        [self.shares_held / MAXIMUM_SHARE_NUM],
        [self.purchase_analysis / MAXIMUM_PRICE_SHARE],
        [self.total_sales_value / (MAXIMUM_SHARE_NUM * MAXIMUM_PRICE_SHARE)],
    ], axis=1)

    return obs
```

_take_action –

The take action method must execute the model's action and either buy, sell, or hold the stock.



```python
#This method is used to take action using trained model/agent.
def _take_action(self, action):
    current_price = random.uniform(
        self.df.loc[self.current_step_locator, "Open"], self.df.loc[self.current_step_locator, "Close"])

    action_type = action[0]
    amount = action[1]
    if action_type < 1:
        # Buy amount % of balance in shares
        available_bal = int(self.balance / current_price)
        purchase_share = int(available_bal * amount)
        last_balance = self.purchase_analysis * self.shares_held
        after_purchase_balance = purchase_share * current_price

        self.balance -= after_purchase_balance
        self.purchase_analysis = (last_balance + after_purchase_balance) / (self.shares_held + purchase_share)
        self.shares_held += purchase_share

        if purchase_share > 0:
            self.trades.append({'step': self.current_step_locator,
                                'shares': purchase_share, 'total': after_purchase_balance,
                                'type': "buy"})
    elif action_type < 2:
        # Sell amount % of shares held
        shares_sold = int(self.shares_held * amount)
        self.balance += shares_sold * current_price
        self.shares_held -= shares_sold
        self.number_share_sold += shares_sold
        self.total_sales_value += shares_sold * current_price

        if shares_sold > 0:
            self.trades.append({'step': self.current_step_locator,
                                'shares': shares_sold, 'total': shares_sold * current_price,
                                'type': "sell"})

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

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

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

We can now use a data frame to create a StockTradingEnv environment and test it with a model from stable-baselines.

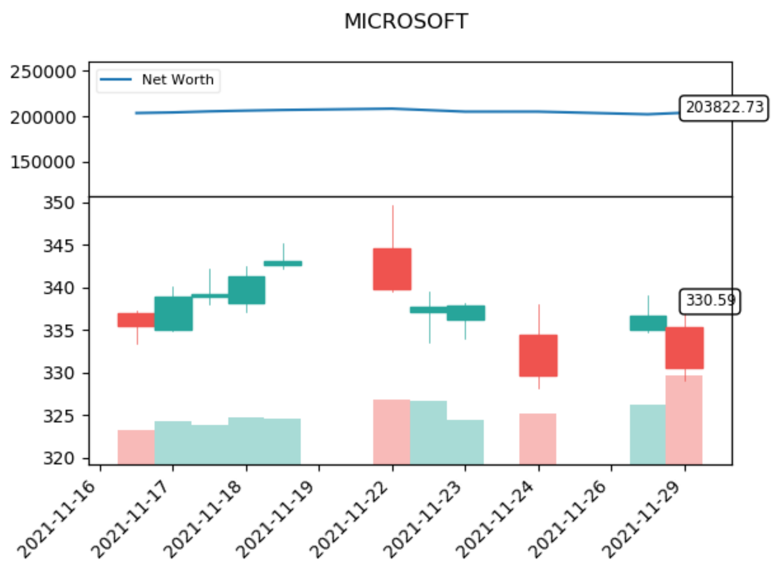
Now, using an agent that performs random actions, perform some actions in the environment.

I used the following algorithms in this project.

PPO2

- The A2C (multiple workers) and TRPO (proximal policy optimization) algorithms are combined in the Proximal Policy Optimization algorithm. PPO2 is a GPU-based implementation of OpenAI. It's a new policy gradient reinforcement learning approach that alternates between sampling data via environment interaction and sampling data directly. This method's goal is to allow multiple epochs of mini batch updates.

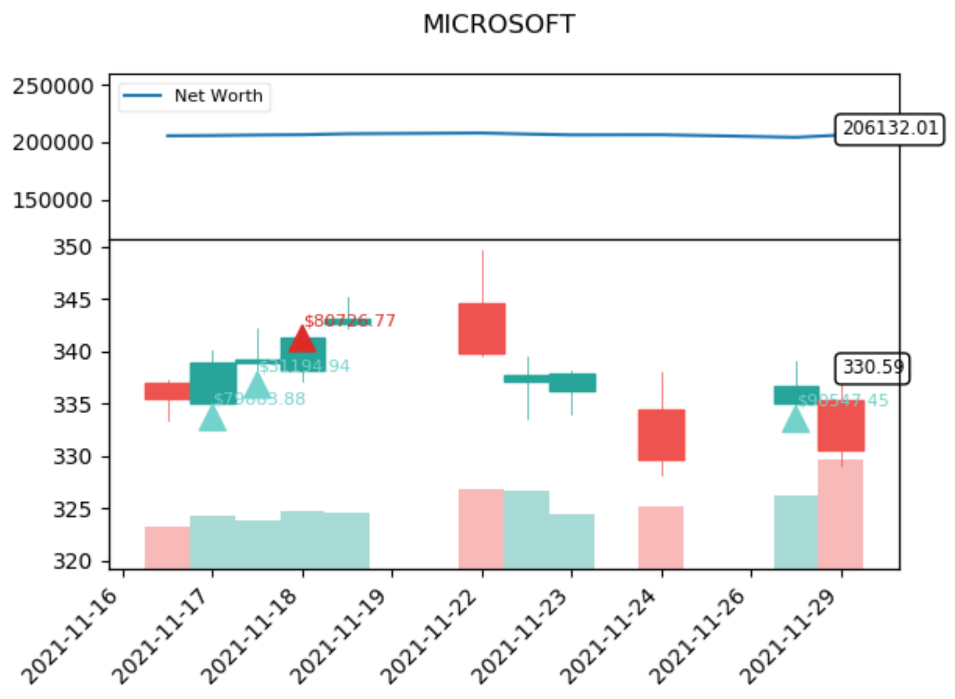
```
In [10]: # The algorithms require a vectorized environment to run
         vec_env = DummyVecEnv([lambda: stock_trading_env(df)])

         gym_model = PPO2(MlpPolicy, vec_env, verbose=1)
         gym_model.learn(total_timesteps = 1000)

         monitor = vec_env.reset()
         for i in range(200):
             action, _states = gym_model.predict(monitor)
             obs, rewards, done, info = vec_env.step(action)
             vec_env.render()
```



```
Step: 200
Balance: 326.6992218552914
Shares held: 607 (Total sold: 3612)
Avg cost for held shares: 318.90907627376737 (Total sales value: 1040634.8401118522)
Net worth: 203822.72887172102 (Max net worth: 208321.87796474388)
Profit: 53822.72887172102
```

ACKTR

- ACKTR combines three distinct techniques to achieve its goal: actor-critic methods, trust region optimization for more consistent improvement, and distributed Kronecker factorization to improve sample efficiency and scalability. Because it takes a step in the natural gradient direction rather than the gradient direction, ACKTR has a higher sample complexity than first-order methods like A2C.



```
Step: 200
Balance: 17295.126118071596
Shares held: 561 (Total sold: 4578)
Avg cost for held shares: 328.83178571034597 (Total sales value: 1244098.035563889)
Net worth: 206132.00612153288 (Max net worth: 207959.61550728392)
Profit: 56132.006121532875
```

Comparison

- As per the result we can see that ACKTR is giving better result than PPO2 but ACKTR is slow in processing than PPO2. ACKTR require less data to give human like performance , but in PPO2 its opposite as it requires large amount of data. Moreover , ACKTR also provide optimization method for model training.