multi-agent-market-rl

Release 0.0.1

Benjamin Suter

Sep 14, 2021

CONTENTS:

CHAPTER

ONE

INSTALLATION

For a local installation, follow the below instructions. We highly recommend installing the following application into a Conda environment.

1. Clone the main branch of the following repository.:

```
git clone https://github.com/jan-engelmann/multi-agent-market-rl
cd multi-agent-market-rl
```

2. Make a Conda environment named market-rl:

```
conda env create -f environment.yml
```

3. Verify that the new environment was installed correctly:

```
conda env list
```

4. Activate the environment in order to make use of the market environment:

```
conda activate market-rl
```

CHAPTER

TWO

QUICK GUIDE TO USING THE ENVIRONMENT

2.1 The agent dictionary

The agent dictionary is used to inform the environment of how many agents and of what type and with what specifications will be participating. The agent dictionary has a nested dictionary structure. More specifically, every agent with the role *seller* and every agent with the role *buyer* are collected in separate dictionaries.

```
agent_dict = {'sellers': seller_agent_dict, 'buyers': buyer_agent_dict}
```

The **seller_agent_dict** is again a nested dictionary made up of *n* (number of selling agent configurations) single agent dictionaries (same holds for the **buyer_agent_dict**).

```
seller_agent_dict = {1: single_agent_seller_1, ..., n: single_agent_seller_n}
buyer_agent_dict = {1: single_agent_buyer_1, ..., m: single_agent_buyer_m}
```

Finally a single agent dictionary is made up of the following `key:value` pairs:

Mandatory key:value pairs

- **'type' (str)** The name of the wanted agent class object
- **'reservation' (int)** The reservation price for this agent

Optional key:value pairs

- **'multiplicity' (int)** The multiplicity count of the specific agent implementation (default=1)
- ****kwargs** Additional keyword arguments specific to the chosen agent type

```
single_agent_dict = {'type': 'MyAgentType', 'reservation': 12, 'multiplicity': 3,
**kwargs}
```

The agent types currently implemented have the following type specific `kwargs`:

DQNAgent

- **network_type: str, optional (default="SimpleExampleNetwork")** Name of network class implemented in network_models.py
- **q_lr: float (default=0.001)** Learning rate provided to the Q-Network optimizer
- **save_weights_directory: str (default="../saved_agent_weights/default_path/{self.agent_name}/")** Directory to where model weights will be saved to
- **save_weights_file: str (default="default_test_file.pt")** File name of the saved weights. Must be a .pt or .pth file
- **load_weights_path: str (default=False)** If a path is provided, agent will try to load pretrained weights from there

ConstAgent

- **const_price: int (default=Mean value of action space)** The constant asking / bidding price

HumanReplayAgent

- **data_type: str (default='new_data')** Data set used (new_data or old_data). See the git directory 'HumanReplayData'
- **treatment: str (default='FullLimS')** Market treatment used. See https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3131004
- **id: int (default=954)** Player id. Must match with agent 'role', 'reservation', 'data_type' and 'treatment'. See the .csv files in the git directory 'HumanReplayData/data_type'

Example of a complete agent dictionary:

```
agent_dict = {'sellers': {1: {'type': 'DQNAgent', 'reservation': 5}},
              'buyers': {1: {'type': 'ConstAgent',
                             'reservation': 15,
                             'const_price': 7},
                         2: {'type': 'ConstAgent',
                             'reservation': 20,
                             'const_price': 18}}}
```

This would result in one **DQNAgent** seller with a reservation price of 5 and two **ConstAgent** buyers with reservation price and bidding price of (15, 7) and (20, 18) respectively. If all goes well, the **DQNAgent** should learn to sell his product to the **ConstAgent** bidding 18.

2.2 Initialising the environment

Now we can start by initialising the environment. For this we need the following arguments:

- **Agent dictionary (dict)** See *The agent dictionary*
- **Market setting (str)** You can specify what market engine to use by passing the market class name as a string. Optionally you can specify the market engine by providing a pre-initialised market class object. Currently the only implemented market engine is **MarketMatchHiLo**
- **Information setting (str)** You can specify what information setting to use by passing the information setting class name as a string. Optionally you can specify the information setting by providing a pre-initialised information setting class object. Currently the implemented information settings are **BlackBoxSetting**, **OfferInformationSetting**, **DealInformationSetting** and **TimeInformationWrapper**
- **Exploration setting (str)** You can specify what exploration setting to use by passing the exploration setting class name as a string. Optionally you can specify the exploration setting by providing a pre-initialised exploration setting class object. Currently the only implemented exploration setting is **LinearExplorationDecline**
- **Reward setting (str)** You can specify what reward setting to use by passing the reward setting class name as a string. Optionally you can specify the reward setting by passing a pre-initialised reward setting class object. Currently the only implemented reward setting is **NoDealPenaltyReward**
- **Optional kwargs** We can fine tune all market, information, exploration and reward settings by providing a keyword argument dictionary for every individual setting. In addition we can specify on what device the environment and on what device the agent networks should operate. The currently implemented keyword arguments are the following:
  - **market_settings**
    - **Global**

· **max_steps: int (default=30)** Maximum number of time steps before the game is reset.

* **MarketMatchHiLo** None

– **info_settings**

* **Global** None

* **BlackBoxSetting** None

* **OfferInformationSetting**

· **n_offers: int (default=1)** Number of offers to see. For instance, 5 would mean the agents see the best 5 bids and asks

* **DealInformationSetting**

· **n_deals: int (default=1)** Number of deals to see

* **TimeInformationWrapper**

· **base_setting: InformationSetting object (default="BlackBoxSetting")** The base information setting to add time to

– **exploration_settings**

* **Global** None

* **LinearExplorationDecline**

· **initial_expo: float (default=1.0)** Initial exploration probability

· **n_expo_steps: int (default=100000)** Number of time steps over which the exploration rate will decrease linearly

· **final_expo: float (default=0.0)** Final exploration rate

– **reward_settings**

* **Global** None

* **NoDealPenaltyReward**

· **no_deal_max: int (default=10)** Number of allowed time steps without making a deal before being punished

– **device: list (default=['cpu', 'cpu'])** Responsible for providing GPU support. The environment is thought to run on two GPUs. One GPU for the environment and one for the agent optimization. If provided should be a list of two GPU devices. First device will be for the environment, second device will be for agent networks. **'cpu'** refers to the current CPU device. **'cuda'** refers to the current GPU device. In order to differentiate between different GPU devices use **'cuda:i'** where **i** is the respective GPU index (starting from zero)

Example of a complete keyword argument dictionary fine tuning the environment settings as well as initialising an environment compatible with the chosen keyword arguments:

```
settings_kwargs = {'market_settings': {'max_steps': 45},
                   'info_settings': {'n_offers': 10},
                   'exploration_settings': {'initial_expo': 0.95,
                                            'n_expo_steps': 1e6,
                                            'final_expo': 1e-5},
                   'reward_settings': {'no_deal_max': 5},
                   'devices': ['cuda:0', 'cuda:1']}

```

(continues on next page)

(continued from previous page)

```
env = MultiAgentEnvironment(agent_dict,
                            'MarketMatchHiLo',
                            'OfferInformationSetting',
                            'LinearExplorationDecline',
                            'NoDealPenaltyReward',
                            **settings_kwargs)
```

2.3 Main functionalities of the environment

There are two main functionalities of the environment.

- **env.reset()** Will reset the environment to its initial settings.
- **env.step(random_action=False)** Will perform one single time step forward in the environment. If *random_action=True* all agents will perform a random action. In addition this function returns the current observations, current actions, current rewards, next observations, agent states (active or finished) and a 'done flag' indicating if the current game has finished or not.

CHAPTER

THREE

QUICK GUIDE TO USING THE DEEPQTRAINER

The **DeepQTrainer** is a ready built trainer copying the training procedure of the *Human-level control through deep reinforcement learning* paper, see https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf . Therefore the application of this trainer is mainly useful in the context of training **DQNAgents**.

3.1 Initialising the DeepQTrainer

The main idea of this trainer, is to use a replay buffer from which samples are randomly picked in order to train the agents. The **DeepQTrainer** uses the following arguments:

- **env: environment object** The current environment class object in which the agents are living. See *Initialising the environment*
- **memory_size: int** Total size of the ReplayBuffer. This corresponds to the total number of actions memorised for every agent.
- **replay_start_size: int** Number of ReplayBuffer slots to be initialised with a uniform random policy before learning starts

In addition, there are some optional keyword arguments to allow for further fine tuning of the trainer

- **discount: float, optional (default=0.99)** Multiplicative discount factor for Q-learning update
- **update_frq: int, optional (default=100)** Frequency (measured in episode/game counts) with which the target network is updated
- **max_loss_history: int, optional (default=None)** Number of previous episodes for which the loss will be saved for monitoring None –> All episode losses are saved
- **max_reward_history: int, optional (default=None)** Number of previous episodes for which the rewards will be saved for monitoring None –> All episode rewards are saved
- **max_action_history: int, optional (default=None)** Number of previous episodes for which the actions will be saved for monitoring None –> All episode actions are saved
- **loss_min: int, optional (default=-5)** Lower-bound for the loss to be clamped to
- **loss_max: int, optional (default=5)** Upper-bound for the loss to be clamped to
- **save_weights: bool, optional (default=False)** If true, all agent weights will be saved to the respective directory specified by the agent in question

Example initialisation of the DeepQTrainer

```
env = MultiAgentEnvironment(...)
mem_size = 10000
start_size = 500

trainer = DeepQTrainer(env, mem_size, start_size)
```

3.2 Training the agents

It is very easy to start the training process. We just need to make use of the **train(...)** methode. This takes the following arguments:

- **n_episodes: int** Number of episodes/games to train for
- **batch_size: int** Batch size used to update the agents network weights (Number of action/result pairs used in one learning step)

The trainer will return some statistics. Namely the average loss history for every agent, the average reward history for every agent and the action history of every agent. All three are returned as a list of torch.tensors.

Example use of the train(...) method

```
n_episodes = 100000
batch_size = 32
total_loss, total_rew, actions = trainer.train(n_episodes, batch_size)
```

3.3 Remark on implementing a custom training loop

One can implement a multitude of training loops tailored to ones custom made reinforcement learning agents. However, a replay buffer is often needed. Therefore we slightly modified the **tianshou.data.ReplayBuffer** to allow for all data to be of type *torch.tensor* and also modified the random sampling to return batches containing the following keywords:

- **obs (torch.tensor)** All current observations
- **act (torch.tensor)** All current actions
- **rew (torch.tensor)** All current rewards
- **done (bool)** Bool indicating if the episode/game has ended or not
- **obs_next (torch.tensor)** All observations from the next round (t+1)
- **a_states (torch.tensor)** All current agent states (active or done)

The original **tianshou.data.ReplayBuffer** can be found here https://github.com/thu-ml/tianshou

The modified ReplayBuffer can be directly imported from the **marl_env** directory and supports these main features

```
from tianshou.data import Batch
from replay_buffer import ReplayBuffer

# Initialise the buffer with a fixed size
buffer = ReplayBuffer(size=memory_size)

# Add a history batch to the ReplayBuffer --> we make use of tianshou.data.Batch
history_batch = Batch(
```

(continues on next page)

(continued from previous page)

```
            obs=obs,
            act=act,
            rew=rew,
            done=done,
            obs_next=obs_next,
            a_states=a_states,
        )
buffer.add(history_batch)

# Sample a random minibatch of transitions from the replay buffer
batch_data, indices = buffer.sample(batch_size=batch_size)
```

3.4 Further Usage Examples

For further usage examples pleas visit the git **Examples** directory.

CHAPTER

FOUR

DOCUMENTATION OF THE CODE

4.1 A quick overview

The Multi-Agent-Market has the following components:

The environment:

The environment module connects all other modules to creat a working engine. It starts by initialising all needed settings. These consist of all the *agents*, the *market*, the *information setting*, the *exploration setting* and the *reward setting*. In addition further initialisation settings can be defined in an optional keyword argument dictionary. A detailed explanation of all the environment arguments will be provided in

After initialising all agents and settings, the environment takes care of computing a single time step in the market environment. This consists of getting all current observations, computing all agent actions, computing all the deals which got realized at the current time step t as well as the associated rewards.

Side note: The realised deals will depend on the market setting and the achieved rewards will depend on the reward setting.

At the end of each time step t, the environment will return the current observations, the current actions, the current rewards and the current agent status (done or active) of all agents at the time step t as well as the next observations of all agents at time step $t+1$. In addition the environment returns a flag indicating if the game has finished.

The market:

The market engine is in charge of computing all realized deals at the current time step t. Currently only the **MarketMatchHiLo** class is implemented. This market engine calculates deals by matching the highest buying offer with the lowest selling offer. The actual deal price is then taken as the mean value between the matches buying offer and selling offer.

The agents:

The **AgentSetting** class is an abstract base class for all agents. It takes care of initialising the role and reservation price as well as the action space of each agent. Again, custom agents can be created by adding classes overwriting specific methods of the **AgentSetting** class.

Currently the following agents are already implemented:

- **DQNAgent** This agent makes use of an artificial neural network in order to learn an optimal Q-function. This is achieved by making use of experience replay. The Q-Network is then iteratively updated using randomly selected experience minibatches.
- **HumanReplayAgent** This agent replays data gathered from human experiments. All data is obtained from the "Trading-in-a-Black-Box" repository. https://github.com/ikicab/Trading-in-a-Black-Box/tree/f9d05b1a83882d41610638b0ceecfbb51cb05a85
- **ConstAgent** This agent will always perform the same action.

The info setting:

The information setting dictates how much information agents get before deciding on an action. Currently all agents always have access to the same amount of information.

Currently the following information settings are implemented:

- **BlackBoxSetting** Every agent is aware of only its own last offer
- **OfferInformationSetting** Every agent is aware of the best *N* offers of either side (buyer and seller) of the last round.
- **DealInformationSetting** Every agent is aware of *N* deals of the last round
- **TimeInformationWrapper** Wrapper to include the current in game time in the observation.

The exploration setting:

The exploration setting determines the evolution of the probability that an agent will perform a random action (perform exploratory actions). All agents will make use of the same exploration setting.

Currently the following exploration setting is implemented:

- **LinearExplorationDecline** Exploration probability declines linearly from an initial staring value down to a final value over the cors of *n* steps.

The reward setting:

Calculates the reward achieved by a given agent after closing a deal

Currently the following reward setting is implemented:

- **NoDealPenaltyReward** Reward achieved by sellers is given by the difference of the deal and the reservation price of the seller. For buyers the reward is given buy the difference of the reservation and the deal price. In addition, buyers who spent more then N in game time steps without making a deal will receive a linearly increasing penalty (negative reward).

The trainer: Finally we need to be able to train agents living in a given environment. To achieve this, we can build a trainer, that plays through multiple episodes in order to train the agents.

Currently the following trainer is implemented:

- **DeepQTrainer** Trainer following the philosophy of the DQN agents.

4.2 The Agent Module

The *AgentSetting* class provides the basic building blocks required by all agents. This includes the following::

- **Basic initialisation of the agent** Every agent needs a role (buyer or seller), a reservation price, an observation space and an action space. The observation space is automatically determined according to the chosen information setting. The action space is automatically determined according to the agent role as well as the reservation prices of all other agents. Lastly every agent can optionally be assigned to a specific device (cpu or gpu).
- **All necessary methods (in the context of the DeepQTrainer)** The methods *get_action*, *random_action*, *get_q_value*, *get_target*, *reset_target_network*, *save_model_weights* and *load_model_weights* all need to be callable in the context of the DeepQTrainer. However, their functionality is highly agent dependent. Therefore all methods implemented in the *AgentSetting* base class will raise *NotImplementedError*. Therefore every agent class has to overwrite the *get_action* method by means of inheritance of the *AgentSetting* class and thereby assigning the wanted behaviour of each method.

4.2.1 Ready to use agents

The **DQNAgent** class represents an agent who aims to approximate the optimal Q-function via a neural network. This is done by having two identically initialised neural networks (the Q-network and the target-network). When learning, the Q-network is updated accordingly to the 'ground truths' provided by the target-network. Periodically the target-network is then reinitialised with the new weights from the Q-network.

The **HumanReplayAgent** class represents an agent capable of replaying human data gathered in a study. See this paper https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3131004 and/or the git directory 'HumanReplayData' containing the the data as well as additional information such as player ids and reservation prices…

The **ConstAgent** class represents an agent who bids/asks a constant price during the entire game.

4.2.2 Creating your custom agent

It is very easy to add your custom agent to the **agents.py** file. This is done in the following way

```
class MyCustomAgent(AgentSetting):
   def __init__(self,
                role,
                reservation,
                in_features,
                action_boundary,
                device=torch.device('cpu'),
                **kwargs):
      super(MyCustomAgent, self).__init__(role,
                                          reservation,
                                          in_features,
                                          action_boundary,
                                          device=device)

      <your custom code>

   def get_action(self, observation, epsilon=0.05):
      """
      Parameters
      ----------
      observation: torch.tensor
         Current observations
      epsilon: float, obtional (default=0.05)
         Probability for a random action

      Returns
      -------
      action: torch.tensor
      """



      <your custom code>

      return action

   ...
```

Every agent class that inherits from the **AgentSetting** class will have the following methods

- **get_action(self, observation, epsilon=0.05)** observation: torch.tensor, epsilon: float
- **random_action(self, observation=None, epsilon=None)** observation: torch.tensor, epsilon: float
- **get_q_value(self, observation, actions=None)** observation: torch.tensor, action: torch.tensor
- **get_target(self, observation, agent_state=None)** observation: torch.tensor, action: torch.tensor
- **reset_target_network(self)** No arguments
- **save_model_weights(self)** No arguments

By default all methods will raise *NotImplementedError*. In order for your custom agent to make proper use of these methods, you will have to overwrite them. This is achieved by defining the exact same function in your custom agent class as was done for the **get_action** method in the code example above. See the raw methods documentation for detailed description of input and output types.

Side note: Not all agent types will necessarily make use of all methods. For instance zero intelligence agents will never have to save weights. In order to circumvent the need to distinguish between agent types during training, we suggest nonetheless implementing the 'useless' method as a dummy function consisting of a *pass* statement.

Furthermore all intelligent agents can make use of your custom neural network model via the **NetworkSetting** class. First, you can define your own neural network model in the **network_models.py** file by inheriting the **NetworkSetting** base class and overwriting the **define_network()** method. Be sure to return a pytorch neural network model.

```
class MyCustomNetwork(NetworkSetting):
    def __init__(self, in_features, out_features, device=torch.device('cpu'), **kwargs):
        super(MyCustomNetwork, self).__init__(in_features,
                                              out_features,
                                              device=device,
                                              **kwargs)

    def define_network(self):
    """
    Defines a simple network for the purpose of being an example

    Returns
    -------
    network: torch.nn
        The wanted neural network
    """
    network = <your custom torch neural network>

    return network
```

In order to load your custom neural network into your intelligent agent, be sure to add a keyword argument "network_type"="MyCustomNetwork" to your agent kwargs dictionary, where "MyCustomNetwork" is a placeholder for the class name of your actual network. Now you can load the neural network as follows:

```
import network_models as network_models

class MyCustomAgent(AgentSetting):
    def __init__(self, role, reservation, in_features, ..., device, **kwargs):
        ...
        network_type = kwargs.pop("network_type", <name of default network>)
        out_features = len(self.action_space)
```

(continues on next page)

(continued from previous page)

```
    network_builder = getattr(network_models, network_type)(
        in_features, out_features, device=self.device, **kwargs
    )

    my_custom_network = network_builder.get_network()
```

The resulting network will already be located on the correct device (cpu or gpu). Also, if kwargs contains a "load_weights_path" keyword, the model weights will be loaded from the provided path.

Now you can make use of your custom agent in the same way as you use the other agents.

4.3 The Market Module

The **BaseMarketEngine** class forms the basis of all market engines. It takes care of initialising the number of sellers and the number of buyers as well as the max time duration of one episode/game and implements the buyer, seller and deal histories.

4.3.1 Ready to use market engines

The **MarketMatchHiLo** class is a ready implemented matching algorithm matching the highest bidding buyer with the lowest asking seller, second highest bidder with the second lowest asking seller, and so on. The actual deal price is then computed as the mean between the matches bidding and asking prices.

4.3.2 Creating your custom engine

Again you can creat your own market engine with custom deal matching by inheriting the **BaseMarketEngine** class and overwriting the **calculate_deals** method. However you are confined to returning the calculated deals for sellers and buyers separately as torch.tensors.

```
class MyCustomMarketEngine(BaseMarketEngine):

   def __init__(self, n_sellers, n_buyers, device=torch.device('cpu'), **kwargs):
      super(MyCustomMarketEngine, self).__init__(n_sellers,
                                                 n_buyers,
                                                 device=device,
                                                 **kwargs)


      <your custom code>

   def calculate_deals(self, s_actions, b_actions):
      """

      Parameters
      ----------
      s_actions: torch.Tensor
         Has shape n_sellers
      b_actions: torch.Tensor
         Has shape n_buyers

```

(continues on next page)

(continued from previous page)

```
        Returns
        -------
        deals_sellers: torch.Tensor
            Has shape n_sellers
        deals_buyers: torch.Tensor
            Has shape n_buyers
        """

        <your custom code>

        return deals_sellers, deals_buyers
```

Again, once you have implemented you custom market engine in the *markets.py* file, you can make use of the new market engine as usual in the environment.

4.4 The Information Setting Module

The **InformationSetting** class forms the basis of all information settings. It provides access to all environment variables as well as the **get_states(...)** method which can be overwritten in order to define your custom information states.

4.4.1 Ready to use information settings

The **BlackBoxSetting** class provides a setting where the information state of each agent consists of only its last action.

The **OfferInformationSetting** class provides a setting where the information state of each agent consists of the best N offers on both sides (buyer and seller). Therefore each agent will have the same 2*N offers as its information state. The **DealInformationSetting** class provides a setting where the information state of each agent consists of the best N deals of the last round. Again, every agent will have the same information state.

The **TimeInformationWrapper** class takes any other information setting class and adds the current time step to the information state.

4.4.2 Creating your custom information setting

As stated above, you can creat your custom information setting by inheriting the **InformationSetting** base class and overwriting the **get_states(...)** method. Again, the return object is the limiting factor to your creativity. It is expected, that the **get_states(...)** returns a *torch.tensor* object of size (n_agents, n_features). Where the n_agents dimension first contains all sellers and then contains all buyers. n_features represents the number of distinct information feature each agent gets.

```
class MyCustomInformationClass(InformationSetting):

        def __init__(self, env, **kwargs):
            super(MyCustomInformationClass, self).__init__(env)
            <your custom code>

        def get_states(self):
            """
            Returns
            -------
```

(continues on next page)

(continued from previous page)

```
            total_info: torch.tensor
                Return total_info as tensor with shape (n_agents, n_features) where
                n_features == number of infos Observations are ordered in the same way
                as res in MultiAgentEnvironment.get_actions().
                total_info[:n_sellers, :] contains all observations for the seller agents
                total_info[n_sellers:, :] contains all observations for the buyer agents
            """

            <your custom code>

            return total_info
```

Again, once you have implemented your custom information setting in the **info_setting.py**, you can make use of it in the same way as the ready to use info settings.

4.5 The Exploration Setting Module

The **ExplorationSetting** class forms the base class of all exploration settings. Currently it does absolutely nothing besides initialising the *epsilon* class variable representing the probability for a random (exploratory) action.

4.5.1 Ready to use exploration settings

The **LinearExplorationDecline** class provides a setting, where the exploration vale declines linearly from a given starting value to a given final value over a given number of steps.

4.5.2 Creating your custom exploration setting

As always, you can creat your custom exploration setting in the **exploration_setting.py** file by inheriting from the **ExplorationSetting** base class and overwriting the **update()** method. Again, the 'updated' probability to perform a random action must be assigned to the *epsilon* class variable in order to have an effect.

```
class MyCustomExploration(ExplorationSetting):

   def __init__(self, **kwargs):
      super(MyCustomExploration, self).__init__(**kwargs)

      <your custom code>

   def update(self):

      <your custom code>

      self.epsilon = <your new epsilon value>
```

Now you can make use of your newly made exploration setting in the same way as the ready made ones can be implemented into the environment.

4.6 The Reward Setting Module

The **RewardSetting** class forms the basis of all reward settings. It provides access to all environment variables.

4.6.1 Ready to use reward settings

The **NoDealPenaltyReward** class provides a reward setting, where buyers receive a linearly growing penalty if they do not manage to close a deal after N rounds. This has the objective of enticing the agents to act quickly.

4.6.2 Creating your custom reward setting

In order to creat your custom reward setting in the **reward_setting.py** file, you must inherit the **RewardSetting** base class and overwrite the **seller_reward**() and the **buyer_reward** methods. Again, you are restricted in the shape of the return object. Both methods need to return a torch.tensor object with shape (n_sellers,) and (n_buyers,) respectively.

```
class MyCustomReward(RewardSetting):

   def __init__(self, env, **kwargs):
      super(MyCustomReward, self).__init__(env)
      <your custom code>

   def seller_reward(self, seller_deals):
      """
      Parameters
      ----------
      seller_deals: torch.tensor
         Has shape (n_sellers,)

      Returns
      -------
      seller_rewards: torch.tensor
         Has shape (n_sellers,)
      """

      <your custom code>

      return seller_rewards

   def buyer_reward(self, buyer_deals):
      """
      Parameters
      ----------
      buyer_deals: torch.tensor
         Has shape (n_buyers,)

      Returns
      -------
      buyer_rewards: torch.tensor
         Has shape (n_buyers,)
      """

```

(continues on next page)

(continued from previous page)

```
        <your custom code>

        return buyer_rewards
```

Now you can make use of your custom reward setting in the same manner as the ready implemented reward settings are used.

4.7 The Neural Network Model Module

The **NetworkSetting** class is intended to provide easy to use modularity with regards to the neural networks used by intelligent agents. Following the same architecture as the previous modules, the **NetworkSetting** class forms the base class, providing all needed functionalities.

4.7.1 Ready to use network models

The **SimpleExampleNetwork** class provides a simple neural network for the purpose of being an example.

4.7.2 Creating your custom neural network

In order to define your own neural network in the **network_models.py** file, which can be used by intelligent agents, you have to inherit the **NetworkSetting** base class and overwrite the **define_network** methode. The class variables **in_features** and **out_features** provide the input and output shape of your neural network. The method must return a torch neural network (torch.nn).

```
class MyCustomNetworkModel(NetworkSetting):

    def __init__(self, in_features, out_features, device=torch.device('cpu'), **kwargs):
        super(SimpleExampleNetwork, self).__init__(in_features,
                                                   out_features,
                                                   device=device,
                                                   **kwargs)

    def define_network(self):
        """
        Returns
        -------
        network: torch.nn
            The wanted neural network
        """

        <your custom code>

        return network
```

Now you can import your newly created neural network model into your custom made agent class by making use of the **get_network** methode. Add something in the lines of this to your init function:

```
network_builder = getattr(network_models, "MyCustomNetworkModel")(
    in_features, out_features, device=self.device, **kwargs
```

(continues on next page)

(continued from previous page)

```
)
my_network = network_builder.get_network()
```

4.8 Raw module functions

class agents.**AgentSetting**(*role*, *reservation*, *in_features*, *action_boundary*, *device=device(type='cpu')*)

Abstract agent class

__init__(*role*, *reservation*, *in_features*, *action_boundary*, *device=device(type='cpu')*) → None

get_action(*observation*, *epsilon=0.05*) → NotImplementedError

get_q_value(*observation*, *actions=None*) → NotImplementedError

get_target(*observation*, *agent_state=None*) → NotImplementedError

random_action(*observation=None*, *epsilon=None*) → NotImplementedError

reset_target_network() → NotImplementedError

save_model_weights() → NotImplementedError

class agents.**ConstAgent**(*role*, *reservation*, *in_features*, *action_boundary*, *device=device(type='cpu')*, ***kwargs*)

class Optimizer

Dummy optimizer –> all members will pass

__init__()

step()

zero_grad()

__init__(*role*, *reservation*, *in_features*, *action_boundary*, *device=device(type='cpu')*, ***kwargs*) → None

role: str role can be 'buyer' or 'seller'

reservation: int The reservation price needs to be in the open interval of (0, infinity). In the case of a seller, the reservation price denotes the fixed costs and in the case of a buyer, the reservation price denotes the budget of the agent.

in_features: int Number of features observed by the agent

action_boundary: int In case the agent is a seller, action_boundary should equal the largest reservation price of all buyers. In case the agent is a buyer, action_boundary should equal the smallest reservation price of all sellers.

device: torch.device, optional (default=torch.device('cpu')) The device on which the agent will run (cpu or gpu)

kwargs:

const_price: int (default=(reservation + action_boundary)//2.0) The constant asking / bidding price

get_action(*observation*, *epsilon=0.05*)

Returns the constant action price of the agent

torch.Tensor of shape (1,) containing the action price of the agent

get_q_value(*observation*, *actions=None*)
Dummy function –> ConstAgent is a zero intelligence agent

get_target(*observation*, *agent_state=None*)
Dummy function –> ConstAgent is a zero intelligence agent

random_action(*observation=None*, *epsilon=None*) → NotImplementedError

reset_target_network()
Dummy network reset –> will pass

save_model_weights()
Dummy weight saver –> will pass

class agents.**DQNAgent**(*role*, *reservation*, *in_features*, *action_boundary*, *device=device(type='cpu')*, ***kwargs*)

__init__(*role*, *reservation*, *in_features*, *action_boundary*, *device=device(type='cpu')*, ***kwargs*) → None
Agents are implemented in such a manner, that asking and bidding prices are given as integer values. Therefore the unit of the price will be equal to the smallest possible unit of currency e.g. for CHF 1 == 0.05 CHF

role: str role can be 'buyer' or 'seller'

reservation: int The reservation price needs to be in the open interval of (0, infinity). In the case of a seller, the reservation price denotes the fixed costs and in the case of a buyer, the reservation price denotes the budget of the agent.

in_features: int Number of features observed by the agent

action_boundary: int In case the agent is a seller, action_boundary should equal the largest reservation price of all buyers. In case the agent is a buyer, action_boundary should equal the smallest reservation price of all sellers.

device: torch.device, optional (default=torch.device('cpu')) The device on which the agent will run (cpu or gpu)

kwargs:

network_type: str, optional (default="SimpleExampleNetwork") Name of network class implemented in network_models.py

q_lr: float, optional (default=0.001) Learning rate provided to the Q-Network optimizer

save_weights_directory: str, optional (default="../saved_agent_weights/default_path/{self.agent_name}/") Directory to where model weights will be saved to.

save_weights_file: str, optional (default="default_test_file.pt") File name of the saved weights. Must be a .pt or .pth file

get_action(*observation*, *epsilon=0.05*)
Determines the agent action

observation: torch.Tensor epsilon: float

epsilon defines the exploration rate [0,1]. With a probability of epsilon the agent will perform a random action.

action_price: int

get_q_value(*observation*, *actions=None*)

observation: torch.Tensor Agent observations. Should have shape (batch_size, observation_size)

actions: torch.Tensor, optional (default = False) Will provide the Q values corresponding to the provided actions. actions should have shape (batch_size, 1) If no actions are provided, the Q value will correspond to the maximal value

max_q: torch.Tensor Tensor containing all Q values. Has shape (batch_size, 1)

`get_target`(*observation*, *agent_state=None*)
observation agent_state

`random_action`(*observation=None*, *epsilon=None*)
A uniform random policy intended to populate the replay memory before learning starts

observation: None Dummy parameter copying get_action()

epsilon: None Dummy parameter copying get_action()

action_price: int Uniformly sampled action price.

`reset_target_network`()
Initialises the target network with the state dictionary of the Q-network

`save_model_weights`()
Saves the model weights in a given directory using a specific file name

class agents.`HumanReplayAgent`(*role*, *reservation*, *in_features*, *action_boundary*, *device=device(type='cpu')*, ***kwargs*)

class `Optimizer`
Dummy optimizer –> all members will pass

`__init__`()

`step`()

`zero_grad`()

`__init__`(*role*, *reservation*, *in_features*, *action_boundary*, *device=device(type='cpu')*, ***kwargs*) → None

role: str role can be 'buyer' or 'seller'

reservation: int The reservation price needs to be in the open interval of (0, infinity). In the case of a seller, the reservation price denotes the fixed costs and in the case of a buyer, the reservation price denotes the budget of the agent.

in_features: int Number of features observed by the agent

action_boundary: int In case the agent is a seller, action_boundary should equal the largest reservation price of all buyers. In case the agent is a buyer, action_boundary should equal the smallest reservation price of all sellers.

device: torch.device, optional (default=torch.device('cpu')) The device on which the agent will run (cpu or gpu)

kwargs:

data_type: str, optional (default='new_data') Data set used (new_data or old_data). See the git directory 'HumanReplayData'

treatment: str, optional (default='FullLimS') Market treatment used. See https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3131004

id: int, optional (default=954) Player id. Must match with agent 'role', 'reservation', 'data_type' and 'treatment'. See the .csv files in the git directory 'HumanReplayData/data_type'

get_action(*observation*, *epsilon=0.05*)
Returns the next bid from the data set for every market step. If all data steps have been used, we will restart from the beginning.

torch.Tensor of shape (1,) containing the bid price of the agent

get_q_value(*observation*, *actions=None*)
Dummy function –> HumanReplayAgent is a zero intelligence agent

get_target(*observation*, *agent_state=None*)
Dummy function –> HumanReplayAgent is a zero intelligence agent

random_action(*observation=None*, *epsilon=None*) → NotImplementedError

reset_target_network()
Dummy network reset –> will pass

save_model_weights()
Dummy weight saver –> will pass

class network_models.**NetworkSetting**(*in_features*, *out_features*, *device=device(type='cpu')*, ***kwargs*)
Abstract network setting class. Used to define your custom neural networks which can then be used by intelligent agents

__init__(*in_features*, *out_features*, *device=device(type='cpu')*, ***kwargs*)
Initialises in_features and out_features needed to construct a fitting neural network

in_features: int Determined by the info_setting –> How many features does the agent see

out_features: int Determined by the action space –> Number of legal actions including No action

device: torch.device, optional (default cpu) Device on which the neural network is intended to run (cpu or gpu)

kwargs: Optional additional keyword arguments

load_weights_path: str, optional (default=False) If a path is provided, agent will try to load pre-trained weights from there.

define_network()

network: torch.nn The wanted neural network

get_network()
Will return the wanted neural network model located on the intended device (cpu or gpu)

network: torch.nn The wanted neural network on the correct device

class network_models.**SimpleExampleNetwork**(*in_features*, *out_features*, *device=device(type='cpu')*, ***kwargs*)
A simple network fulfilling the role of being an example

__init__(*in_features*, *out_features*, *device=device(type='cpu')*, ***kwargs*)
Initialises in_features and out_features needed to construct a fitting neural network

in_features: int Determined by the info_setting –> How many features does the agent see

out_features: int Determined by the action space –> Number of legal actions including No action

device: torch.device, optional (default cpu) Device on which the neural network is intended to run (cpu or gpu)

kwargs: Optional additional keyword arguments

load_weights_path: str, optional (default=False) If a path is provided, agent will try to load pre-trained weights from there.

`define_network()`
Defines a simple network for the purpose of being an example

network: torch.nn The wanted neural network

`get_network()`
Will return the wanted neural network model located on the intended device (cpu or gpu)

network: torch.nn The wanted neural network on the correct device

`class info_setting.BlackBoxSetting(env, **kwargs)`
The agent is aware of only its own last offer.

`__init__(env, **kwargs)`
env: environment class object kwargs:

Additional keyword arguments

`get_states()`

total_info: torch.tensor Return total_info as tensor with shape (n_agents, n_features) where n_features == 1 Observations are ordered in the same way as res in MultiAgentEnvironment.get_actions(). total_info[:n_sellers, :] contains all observations for the seller agents total_info[n_sellers:, :] contains all observations for the buyer agents

`class info_setting.DealInformationSetting(env, **kwargs)`
The agent is aware of N deals of the last round.

Note: the N deals need not be sorted on deal price. It depends the order the matcher matches deals, see `MarketEngine.matcher`.
kwargs: dict

n_deals: int, optional (default=1) Number of deals to see.
`__init__(env, **kwargs)`
env: environment class object kwargs:

'n_deals' (int)

`get_states()`

total_info: torch.tensor Return total_info as tensor with shape (n_agents, n_features) where n_features == n_deals Observations are ordered in the same way as res in MultiAgentEnvironment.get_actions(). total_info[:n_sellers, :] contains all observations for the seller agents total_info[n_sellers:, :] contains all observations for the buyer agents

`class info_setting.InformationSetting(env)`
Abstract information setting class.
env: Environment object The current environment class object.
`__init__(env)`
env: environment class object

`get_states()`
Compute the observations of agents given the market object.

class info_setting.**OfferInformationSetting**(*env*, ***kwargs*)

The agent is aware of the best N offers of either side of the last round.

kwargs: dict

n_offers: int, optional (default=1) Number of offers to see. For instance, 5 would mean the agents see the best 5 bids and asks.

__init__(*env*, ***kwargs*)

env: environment class object kwargs:

'n_offers' (int)

get_states()

total_info: torch.tensor Return total_info as tensor with shape (n_agents, n_features) where n_features == 2*n_offers Observations are ordered in the same way as res in MultiAgentEnvironment.get_actions(). total_info[:n_sellers, :] contains all observations for the seller agents total_info[n_sellers:, :] contains all observations for the buyer agents

class info_setting.**TimeInformationWrapper**(*env*, ***kwargs*)

Wrapper to include the time in the observation.

This class takes as input another information setting and adds the time of the market to the observations of that information setting. This allows certain fixed agents to adopt time-dependent strategies.

kwargs: dict

base_setting: InformationSetting object The base information setting to add time to.

__init__(*env*, ***kwargs*)

env: environment class object kwargs:

'base_setting' (str)

get_states()

total_info: torch.tensor Return total_info as tensor with shape (n_agents, n_features) where n_features == base_features + 1 Observations are ordered in the same way as res in MultiAgentEnvironment.get_actions(). total_info[:n_sellers, :] contains all observations for the seller agents total_info[n_sellers:, :] contains all observations for the buyer agents

class exploration_setting.**ExplorationSetting**(***kwargs*)

Abstract exploration setting class

__init__(***kwargs*)

update()

class exploration_setting.**LinearExplorationDecline**(***kwargs*)

kwargs:

initial_expo: float, optional (default=1.0) Initial exploration probability

n_expo_steps: int, optional (default=100000) Number of time steps over which the exploration rate will decrease linearly

final_expo: float, optional (default=0.0) Final exploration rate

__init__(***kwargs*)

reset()

update()

class reward_setting.**NoDealPenaltyReward**(*env*, ***kwargs*)

Punishes buyers with a linearly increasing negative reward if they do not manage to close a deal after N rounds.

kwargs:

no_deal_max: int, optional (default=10) Number of allowed time steps without making a deal before being punished

__init__(*env*, ***kwargs*)

buyer_reward(*buyer_deals*)

buyer_deals: torch.tensor Has shape (n_buyers)

rew: torch.tensor Has shape (n_buyers)

seller_reward(*seller_deals*)

seller_deals: torch.tensor Has shape (n_sellers,)

rew: torch.tensor Has shape (n_sellers)

class reward_setting.**RewardSetting**(*env*)

Abstract reward setting class

__init__(*env*)

buyer_reward(*buyer_deals*)

seller_reward(*seller_deals*)

class environment.**MultiAgentEnvironment**(*agent_dict*, *market*, *info_setting*, *exploration_setting*, *reward_setting*, ***kwargs*)

__init__(*agent_dict*, *market*, *info_setting*, *exploration_setting*, *reward_setting*, ***kwargs*)

agent_dict: dict

Nested dictionary of shape {

'sellers': {1: {

'type': 'DQNAgent', 'reservation': 12, 'multiplicity': 3, **kwargs }

… }

'buyers': {1: {

'type': 'DQNAgent', 'reservation': 5, **kwargs }

… }

}

Containing meta_info for all wanted sellers and buyers

Mandatory keywords are:

'type': str Name of the wanted agents class object

'reservation': int Reservation price for this agent

Optional keywords are:

'multiplicity': int Multiplicity count of the agent (default=1)

****kwargs:** Additional keyword arguments specific to the agent type

market: str or markets class object info_setting: str or info_setting class object exploration_setting: str or exploration_setting class object reward_setting: str or reward_setting class object kwargs: dict, optional

Nested dictionary of shape { 'market_settings': {....}, 'info_settings': {....}, 'exploration_settings': {....}, 'reward_settings': {....}, 'devices':... }

Allowing to fine tune keyword arguments of the individual settings. 'device' is responsible for providing GPU support. The environment is thought to run on two GPUs. One GPU for the environment and one for the agent optimization. If provided should be a list of two GPU devices. First device will be for the environment, second device will be for agent networks. Default is CPU.

calculate_rewards(*deals_sellers*, *deals_buyers*)

get_actions()

reset()

step(*random_action=False*)

random_action

current_observations: torch.Tensor All agent observations at the current time step t. The zero dimension has size self.n_agents current_observations[:n_sellers, :] contains all observations for the seller agents current_observations[n_sellers:, :] contains all observations for the buyer agents

current_actions: torch.Tensor All agent actions at the current time step t. The last dimension has size self.n_agents current_actions[:n_sellers] contains all actions for the seller agents current_actions[n_sellers:] contains all actions for the buyer agents

current_rewards: torch.Tensor All agent rewards at the current time step t. The last dimension has size self.n_agents current_rewards[:n_sellers] contains all rewards for the seller agents current_rewards[n_sellers:] contains all rewards for the buyer agents

next_observation: torch.Tensor All agent observations at the next time step t + 1. The zero dimension has size self.n_agents next_observation[:n_sellers, :] contains all observations for the seller agents next_observation[n_sellers:, :] contains all observations for the buyer agents

self.done: bool True if all agents are done trading or if the the game has come to an end

store_observations()

environment.**generate_agents**(*agent_dict*, *device*)

environment.**get_agent_actions**(*agent*, *observation*, *epsilon*, *random_action*)

agent: Abstract agent class observation: torch.Tensor epsilon: float

The probability of returning a random action in epsilon-greedy exploration

random_action: bool If true action is drawn from a uniform random policy or from a specifically implemented random policy called 'random_action'

action: torch.Tensor

environment.**get_basic_agent_info**(*agent_dict*)

class markets.**BaseMarketEngine**(*n_sellers*, *n_buyers*, *device=device(type='cpu')*, ***kwargs*)

__init__(*n_sellers*, *n_buyers*, *device=device(type='cpu')*, ***kwargs*)

n_sellers: int Number of agent sellers

n_buyers: int Number of agent buyers

device: torch.device Allows to allocate the market engine to a cpu or gpu device

kwargs: optional max_steps (default=30), max time steps of one episode/game

`calculate_deals`(*s_actions: torch.Tensor*, *b_actions: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

`reset`()
Reset the market to its initial unmatched state.

`step`(*s_actions: torch.Tensor*, *b_actions: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Does a market step appending actions and deals to history and forwarding market time

s_actions [torch.Tensor] Tensor of seller actions, shape (n_sellers,)

b_actions [torch.Tensor] Tensor of buyer actions, shape (n_buyers,)

typing.Tuple[torch.Tensor torch.Tensor] (deals_sellers, deals_buyers) with shapes (n_sellers,), (n_buyers,)

class markets.**MarketMatchHiLo**(*n_sellers*, *n_buyers*, *device=device(type='cpu')*, ***kwargs*)
Market engine using mechanism that highest buying offer is matched with lowest selling offer

`__init__`(*n_sellers*, *n_buyers*, *device=device(type='cpu')*, ***kwargs*)

n_sellers: int Number of agent sellers

n_buyers: int Number of agent buyers

device: torch.device Allows to allocate the market engine to a cpu or gpu device

kwargs: optional max_steps (default=30), max time steps of one episode/game

`calculate_deals`(*s_actions*, *b_actions*)

`reset`()
Reset the market to its initial unmatched state.

`step`(*s_actions: torch.Tensor*, *b_actions: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Does a market step appending actions and deals to history and forwarding market time

s_actions [torch.Tensor] Tensor of seller actions, shape (n_sellers,)

b_actions [torch.Tensor] Tensor of buyer actions, shape (n_buyers,)

typing.Tuple[torch.Tensor torch.Tensor] (deals_sellers, deals_buyers) with shapes (n_sellers,), (n_buyers,)

class trainer.**DeepQTrainer**(*env*, *memory_size*, *replay_start_size*, ***kwargs*)

`__init__`(*env*, *memory_size*, *replay_start_size*, ***kwargs*)

env: Environment object The current environment class object.

memory_size: int ReplayBuffer size

replay_start_size: int Number of ReplayBuffer slots to be initialised with a uniform random policy before learning starts

kwargs: Optional keyword arguments

discount: float, optional (default=0.99) Multiplicative discount factor for Q-learning update

update_frq: int, optional (default=100) Frequency (measured in episode counts) with which the target network is updated

max_loss_history: int, optional (default=None) Number of previous episodes for which the loss will be saved for monitoring None –> All episode losses are saved

max_reward_history: int, optional (default=None) Number of previous episodes for which the rewards will be saved for monitoring None –> All episode rewards are saved

max_action_history: int, optional (default=None) Number of previous episodes for which the actions will be saved for monitoring None –> All episode actions are saved

loss_min: int, optional (default=-5) Lower-bound for the loss to be clamped to

loss_max: int, optional (default=5) Upper-bound for the loss to be clamped to

save_weights: bool, optional (default=False) If true, all agent weights will be saved to the respective directory specified by the agent in question

`generate_Q_targets`(*obs_next*, *reward*, *agent_state*, *end_of_eps*, *discount=0.99*)

Generates the Q-value targets used to update the agent QNetwork Parameters ———- obs_next: torch.Tensor

Tensor containing all observations for sampled time steps t_j + 1

reward: torch.Tensor Tensor containing all rewards for the sampled time steps t_j

agent_state: torch.Tensor Tensor indicating if an individual agent was finished at sampled time steps t_j

end_of_eps: ndarray Bool array indicating if episode was finished at sampled time steps t_j

discount: float Discount factor gamma used in the Q-learning update.

targets: torch.Tensor Tensor containing all targets y_j used to perform a gradient descent step on (y_j - Q(s_j, a_j; theta))**2 Tensor will have shape (batch_size, n_agents) targets[:, :n_sellers] contains all targets for the seller agents targets[:, n_sellers:] contains all targets for the buyer agents

`generate_Q_values`(*obs*, *act*)

Generates the Q-values for each agent

obs: torch.tensor All current observations

act: torch.tensor All current actions

q_values: torch.tensor The Q-values of the individual agents

`static get_agent_Q_target`(*agent*, *observations*, *agent_state*)

Returns all the Q-targets of the different agents

agent: agent class instance observations: torch.tensor

All current observations

agent_state: torch.tensor The state (active/finished) of all agents

targets: torch.tensor Q-targets of all agents

`static get_agent_Q_values`(*agent*, *observations*, *actions=None*)
Returns all the Q-values of the different agents

agent: agent class instance observations: torch.tensor

All current observations

actions: torch.tensor All current actions

q_values: torch.tensor Q-values of all agents

`mse_loss`(*q_targets*, *q_values*)
Custom MSE-loss with clamping

q_targets: torch.tensor All the Q-value targets

q_values: torch.tensor All the Q-values chosen by the agents

loss: torch.tensor The clamped mean squared error loss

`set_replay_buffer`(*memory_size*, *replay_start_size*)
Initializes the first N replay buffer entries with random actions. Parameters ———- memory_size: int

Total size of the replay buffer

replay_start_size: int Number of buffer entries to be initialized with random actions (corresponds to the integer N)

buffer: tianshou.data.ReplayBuffer Initialised replay buffer

`train`(*n_episodes*, *batch_size*)
Training method.

n_episodes: int Number of episodes (games) to train for

batch_size: int Batch size used to update the agents network weights

list list[0]: Average loss history list[1]: Average reward history list[2]: Action history

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

G

H

I

L

M

N

O

R

S

T

U

Z