tensortrade Documentation

Release 0.1.0-rc1

Adam King

Nov 21, 2019

Contents

TensorTrade is an open source Python framework for building, training, evaluating, and deploying robust trading algorithms using reinforcement learning. The framework focuses on being highly composable and extensible, to allow the system to scale from simple trading strategies on a single CPU, to complex investment strategies run on a distribution of HPC machines.

Under the hood, the framework uses many of the APIs from existing machine learning libraries to maintain high quality data pipelines and learning models. One of the main goals of TensorTrade is to enable fast experimentation with algorithmic trading strategies, by leveraging the existing tools and pipelines provided by `numpy`, `pandas`, `gym`, `keras`, and `tensorflow`.

Every piece of the framework is split up into re-usable components, allowing you to take advantage of the general use components built by the community, while keeping your proprietary features private. The aim is to simplify the process of testing and deploying robust trading agents using deep reinforcement learning, to allow you and I to focus on creating profitable strategies.

The goal of this framework is to enable fast experimentation, while maintaining production-quality data pipelines.

Feel free to also walk through the Medium tutorial.

CHAPTER 1

Guiding principles

Inspired by Keras' guiding principles.

User friendliness. TensorTrade is an API designed for human beings, not machines. It puts user experience front and center. TensorTrade follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

Modularity. A trading environment is a conglomeration of fully configurable modules that can be plugged together with as few restrictions as possible. In particular, exchanges, feature pipelines, action schemes, reward schemes, trading agents, and performance reports are all standalone modules that you can combine to create new trading environments.

Easy extensibility. New modules are simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making TensorTrade suitable for advanced research and production use.

1.1 Getting Started

You can get started testing on Google Colab or your local machine, by viewing our many examples

---

1.2 Installation

TensorTrade requires Python >= 3.6 for all functionality to work as expected.

You can install the package from PyPi via pip or from the Github repo.

```
pip install tensortrade
```

OR

```
pip install git+https://github.com/notadamking/tensortrade.git
```

Some functionality included in TensorTrade is optional. To install all optional dependencies, run the following command:

```
pip install tensortrade[tf,tensorforce,baselines,ccxt,fbm]
```

OR

```
pip install git+https://github.com/notadamking/tensortrade.git[tf,tensorforce,
↪baselines,ccxt,fbm]
```

1.3 Docker

To run the commands below ensure Docker is installed. Visit https://docs.docker.com/install/ for more information

1.3.1 Run Jupyter Notebooks

To run a jupyter notebook execute the following

```
make run-notebook
```

which will generate a link of the form 127.0.0.1:8888/?token=... Paste this link into your browers and select the notebook you'd like to explore

1.3.2 Build Documentation

To build documentation execute the following

```
make run-docs
```

1.3.3 Run Test Suite

To run the test suite execute the following

```
make run-tests
```

1.4 Code Structure

The TensorTrade library is modular. The `tensortrade` library usually has a common setup:

1. An abstract `MetaABC` class that highlights the methods that will generally be called inside of the main `TradingEnvironment`.
2. Specific applications of that abstract class are then specified later to make more detailed specifications.

1.4.1 Example of Structure:

A good example of this structure is the `Exchange` component. It represents all exchange interactions.

The beginning of the code in Exchange is seen here.

```
class Exchange(object, metaclass=ABCMeta):
    """An abstract exchange for use within a trading environment."""

    def __init__(self, base_instrument: str = 'USD', dtype: TypeString = np.float32,␣
↪feature_pipeline: FeaturePipeline = None):
        """
        Arguments:
            base_instrument: The exchange symbol of the instrument to store/measure␣
↪value in.
            dtype: A type or str corresponding to the dtype of the `observation_
↪space`.
            feature_pipeline: A pipeline of feature transformations for transforming␣
↪observations.
        """
        self._base_instrument = base_instrument
        self._dtype = dtype
        self._feature_pipeline = feature_pipeline
```

As you can see above, the Exchange has a large majority of the instantiation details that carries over to all other reprentations of that type of class. `ABCMeta` represents that all classes that inherit it shall be recognizable as an instance of `Exchange`. This is nice when you need to do type checking.

When creating a new exchange type (everything that's an inheritance of the `Exchange`), one needs to add further details for how information should be declared by default. Once you create a new type of exchange, you can have new rules placed in by default. Let's look at the SimulatedExchange and it can have parameters dynamically set via the `**kwargs` arguement in later exchanges.

SimulatedExchange:

```
class SimulatedExchange(Exchange):
    """An exchange, in which the price history is based off the supplied data frame␣
↪and
    trade execution is largely decided by the designated slippage model.
    If the `data_frame` parameter is not supplied upon initialization, it must be set␣
↪before
    the exchange can be used within a trading environment.
    """

    def __init__(self, data_frame: pd.DataFrame = None, **kwargs):
        super().__init__(
            dtype=self.default('dtype', np.float32),
            feature_pipeline=self.default('feature_pipeline', None)
        )

        self._commission_percent = self.default('commission_percent', 0.3, kwargs)
        self._base_precision = self.default('base_precision', 2, kwargs)
        self._instrument_precision = self.default('instrument_precision', 8, kwargs)
        self._min_trade_amount = self.default('min_trade_amount', 1e-6, kwargs)
        self._max_trade_amount = self.default('max_trade_amount', 1e6, kwargs)

        self._initial_balance = self.default('initial_balance', 1e4, kwargs)
        self._observation_columns = self.default(
```

(continues on next page)

(continued from previous page)

```
            'observation_columns',
            ['open', 'high', 'low', 'close', 'volume'],
            kwargs
        )
        self._price_column = self.default('price_column', 'close', kwargs)
        self._window_size = self.default('window_size', 1, kwargs)
        self._pretransform = self.default('pretransform', True, kwargs)
        self._price_history = None

        self.data_frame = self.default('data_frame', data_frame)

        model = self.default('slippage_model', 'uniform', kwargs)
        self._slippage_model = slippage.get(model) if isinstance(model, str) else␣
↪model()
```

Everything that inherits SimulatedExchange uses the specified kwargs to set the parameters.

Therefore, even when we don't directly see the parameters inside of FBMExchange, all of the defaults are being called.

An example:

```
exchange = FBMExchange(base_instrument='BTC', timeframe='1h', base_precision=4) # we
↪'re replacing the default base precision.
```

1.5 Train and Evaluate

[2]:
```
from tensorforce.agents import Agent
from tensorforce.environments import Environment

from tensortrade.environments import TradingEnvironment
from tensortrade.exchanges.simulated import FBMExchange
from tensortrade.features.scalers import MinMaxNormalizer
from tensortrade.features.stationarity import FractionalDifference
from tensortrade.features import FeaturePipeline
from tensortrade.rewards import SimpleProfit
from tensortrade.actions import DiscreteActions
from tensortrade.strategies import TensorforceTradingStrategy

normalize = MinMaxNormalizer(inplace=True)
difference = FractionalDifference(difference_order=0.6,
                                  inplace=True)
feature_pipeline = FeaturePipeline(steps=[normalize, difference])

reward_scheme = SimpleProfit()
action_scheme = DiscreteActions(n_actions=20, instrument='ETH/BTC')

exchange = FBMExchange(base_instrument='BTC',
                       timeframe='1h',
                       should_pretransform_obs=True)

environment = TradingEnvironment(exchange=exchange,
                                 action_scheme=action_scheme,
                                 reward_scheme=reward_scheme,
```

(continues on next page)

(continued from previous page)

```
                                    feature_pipeline=feature_pipeline)

agent_spec = {
    "type": "ppo_agent",
    "step_optimizer": {
        "type": "adam",
        "learning_rate": 1e-4
    },
    "discount": 0.99,
    "likelihood_ratio_clipping": 0.2,
}

network_spec = [
    dict(type='dense', size=64, activation="tanh"),
    dict(type='dense', size=32, activation="tanh")
]

strategy = TensorforceTradingStrategy(environment=environment, agent_spec=agent_spec,␣
↪network_spec=network_spec)

performance = strategy.run(episodes=1, testing=True)

performance[-5:]
```

```
WARNING: The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.
↪md
  * https://github.com/tensorflow/addons
If you depend on functionality not listed there, please file an issue.
```

```
100%|| 1/1 [00:08<00:00,  8.94s/it]
```

```
Finished running strategy.
Total episodes: 1 (1666 timesteps).
Average reward: 1150.3350630537668.
```

[2]:

```
          balance     net_worth
1661  9999.829222  10000.187790
1662  9997.354264   9998.232905
1663  9994.879919   9995.990028
1664  9991.169320   9992.191048
1665  9991.535304   9992.226192
```

[4]:

```
from tensortrade.environments import TradingEnvironment
from tensortrade.exchanges.simulated import FBMExchange
from tensortrade.features.scalers import MinMaxNormalizer
from tensortrade.features.stationarity import FractionalDifference
from tensortrade.features import FeaturePipeline
from tensortrade.rewards import SimpleProfit
from tensortrade.actions import DiscreteActions
from tensortrade.strategies import StableBaselinesTradingStrategy

normalize = MinMaxNormalizer(inplace=True)
difference = FractionalDifference(difference_order=0.6,
```

(continues on next page)

(continued from previous page)

```
                                        inplace=True)
feature_pipeline = FeaturePipeline(steps=[normalize, difference])

reward_scheme = SimpleProfit()
action_scheme = DiscreteActions(n_actions=20, instrument='ETH/BTC')

exchange = FBMExchange(base_instrument='BTC',
                       timeframe='1h',
                       should_pretransform_obs=True)

environment = TradingEnvironment(exchange=exchange,
                                 action_scheme=action_scheme,
                                 reward_scheme=reward_scheme,
                                 feature_pipeline=feature_pipeline)

strategy = StableBaselinesTradingStrategy(environment=environment)

performance = strategy.run(episodes=10)

performance[-5:]
```

```
Finished running strategy.
Total episodes: 10 (16660 timesteps).
Average reward: 1.1827063286237227.
```

```
[4]:            balance     net_worth
     1660  11226.614898  11227.750341
     1661  11226.686075  11227.725535
     1662  11226.721664  11227.658857
     1663  11226.739458  11227.908940
     1664  11223.960840  11225.188709
```

1.6 Components

TensorTrade is built around modular components that together make up a trading strategy. Trading strategies combine reinforcement learning agents with composable trading logic in the form of a `gym` environment. A trading environment is made up of a set of modular components that can be mixed and matched to create highly diverse trading and investment strategies.

Just like electrical components, the purpose of TensorTrade components is to be able to mix and match them as necessary.

1.7 Trading Environment

A trading environment is a reinforcement learning environment that follows OpenAI's `gym.Env` specification. This allows us to leverage many of the existing reinforcement learning models in our trading agent, if we'd like.

Trading environments are fully configurable gym environments with highly composable `Exchange`, `FeaturePipeline`, `ActionScheme`, and `RewardScheme` components.

- The `Exchange` provides observations to the environment and executes the agent's trades.
- The `FeaturePipeline` optionally transforms the exchange output into a more meaningful set of features before it is passed to the agent.

- The `ActionScheme` converts the agent's actions into executable trades.
- The `RewardScheme` calculates the reward for each time step based on the agent's performance.

That's all there is to it, now it's just a matter of composing each of these components into a complete environment.

When the reset method of a `TradingEnvironment` is called, all of the child components will also be reset. The internal state of each exchange, feature pipeline, transformer, action scheme, and reward scheme will be set back to their default values, ready for the next episode.

Let's begin with an example environment. As mentioned before, initializing a `TradingEnvironment` requires an exchange, an action scheme, and a reward scheme, the feature pipeline is optional.

```
from tensortrade.environments import TradingEnvironment

environment = TradingEnvironment(exchange=exchange,
                                 action_scheme=action_scheme,
                                 reward_scheme=reward_scheme,
                                 feature_pipeline=feature_pipeline)
```

1.8 Exchange

Exchanges determine the universe of tradable instruments within a trading environment, return observations to the environment on each time step, and execute trades made within the environment. There are two types of exchanges: live and simulated.

Live exchanges are implementations of `Exchange` backed by live pricing data and a live trade execution engine. For example, `CCXTExchange` is a live exchange, which is capable of returning pricing data and executing trades on hundreds of live cryptocurrency exchanges, such as Binance and Coinbase.

```
import ccxt
from tensortrade.exchanges.live import CCXTExchange

coinbase = ccxt.coinbasepro()
exchange = CCXTExchange(exchange=coinbase, base_instrument='USD')
```

There are also exchanges for stock and ETF trading, such as RobinhoodExchange and InteractiveBrokersExchange, but these are still works in progress.

Simulated exchanges, on the other hand, are implementations of `Exchange` backed by simulated pricing data and trade execution.

For example, `FBMExchange` is a simulated exchange, which generates pricing and volume data using fractional brownian motion (FBM). Since its price is simulated, the trades it executes must be simulated as well. The exchange uses a simple slippage model to simulate price and volume slippage on trades, though like almost everything in TensorTrade, this slippage model can easily be swapped out for something more complex.

```
from tensortrade.exchanges.simulated import FBMExchange

exchange = FBMExchange(base_instrument='BTC', timeframe='1h')
```

Though the `FBMExchange` generates fake price and volume data using a stochastic model, it is simply an implementation of `SimulatedExchange`. Under the hood, `SimulatedExchange` only requires a `data_frame` of price history to generate its simulations. This `data_frame` can either be provided by a coded implementation such as `FBMExchange`, or at runtime.

```python
import pandas as pd
from tensortrade.exchanges.simulated import SimulatedExchange

df = pd.read_csv('./data/btc_ohclv_1h.csv')
exchange = SimulatedExchange(data_frame=df, base_instrument='USD')
```

1.9 Feature Pipeline

Feature pipelines are meant for transforming observations from the environment into meaningful features for an agent to learn from. If a pipeline has been added to a particular exchange, then observations will be passed through the `FeaturePipeline` before being output to the environment.

For example, a feature pipeline could normalize all price values, make a time series stationary, add a moving average column, and remove an unnecessary column, all before the observation is returned to the agent.

Feature pipelines can be initialized with an arbitrary number of comma-separated transformers. Each `FeatureTransformer` needs to be initialized with the set of columns to transform, or if nothing is passed, all input columns will be transformed.

Each feature transformer has a transform method, which will transform a single observation (a `pandas.DataFrame`) from a larger data set, keeping any necessary state in memory to transform the next frame. For this reason, it is often necessary to reset the FeatureTransformer periodically. This is done automatically each time the parent `FeaturePipeline` or `Exchange` is reset.

```python
from tensortrade.features import FeaturePipeline
from tensortrade.features.scalers import MinMaxNormalizer
from tensortrade.features.stationarity import FractionalDifference
from tensortrade.features.indicators import SimpleMovingAverage

price_columns = ["open", "high", "low", "close"]
normalize_price = MinMaxNormalizer(price_columns)
moving_averages = SimpleMovingAverage(price_columns)
difference_all = FractionalDifference(difference_order=0.6)
feature_pipeline = FeaturePipeline(steps=[normalize_price,
                                          moving_averages,
                                          difference_all])

exchange.feature_pipeline = feature_pipeline
```

This feature pipeline normalizes the price values between 0 and 1, before adding some moving average columns and making the entire time series stationary by fractionally differencing consecutive values.

1.10 Action Scheme

Action schemes define the action space of the environment and convert an agent's actions into executable trades.

For example, if we were using a discrete action space of 3 actions (0 = hold, 1 = buy 100 %, 2 = sell 100%), our learning agent does not need to know that returning an action of 1 is equivalent to buying an instrument. Rather, our agent needs to know the reward for returning an action of 1 in specific circumstances, and can leave the implementation details of converting actions to trades to the `ActionScheme`.

Each action scheme has a get_trade method, which will transform the agent's specified action into an executable `Trade`. It is often necessary to store additional state within the scheme, for example to keep track of the currently traded position. This state should be reset each time the action scheme's reset method is called, which is done automatically when the parent `TradingEnvironment` is reset.

```
from tensortrade.actions import DiscreteActions

action_scheme = DiscreteActions(n_actions=20, instrument='BTC')
```

This discrete action scheme uses 20 discrete actions, which equates to 4 discrete amounts for each of the 5 trade types (market buy/sell, limit buy/sell, and hold). E.g. [0,5,10,15]=hold, 1=market buy 25%, 2=market sell 25%, 3=limit buy 25%, 4=limit sell 25%, 6=market buy 50%, 7=market sell 50%, etc…

1.11 Reward Scheme

Reward schemes receive the `Trade` taken at each time step and return a `float`, corresponding to the benefit of that specific action. For example, if the action taken this step was a sell that resulted in positive profits, our `RewardScheme` could return a positive number to encourage more trades like this. On the other hand, if the action was a sell that resulted in a loss, the scheme could return a negative reward to teach the agent not to make similar actions in the future.

A version of this example algorithm is implemented in `SimpleProfit`, however more complex schemes can obviously be used instead.

Each reward scheme has a `get_reward` method, which takes in the trade executed at each time step and returns a `float` corresponding to the value of that action. As with action schemes, it is often necessary to store additional state within a reward scheme for various reasons. This state should be reset each time the reward scheme's reset method is called, which is done automatically when the parent `TradingEnvironment` is reset.

```
from tensortrade.rewards import SimpleProfit

reward_scheme = SimpleProfit()
```

The simple profit scheme returns a reward of -1 for not holding a trade, 1 for holding a trade, 2 for purchasing an instrument, and a value corresponding to the (positive/negative) profit earned by a trade if an instrument was sold.

1.12 Trading Strategy

A `TradingStrategy` consists of a learning agent and one or more trading environments to tune, train, and evaluate on. If only one environment is provided, it will be used for tuning, training, and evaluating. Otherwise, a separate environment may be provided at each step.

```
from stable_baselines import PPO2

from tensortrade.strategies import TensorforceTradingStrategy,
                                   StableBaselinesTradingStrategy

agent_spec = {
    "type": "ppo_agent",
    "step_optimizer": {
        "type": "adam",
        "learning_rate": 1e-4
    },
    "discount": 0.99,
    "likelihood_ratio_clipping": 0.2,
```

(continues on next page)

(continued from previous page)

```
}

network_spec = [
    dict(type='dense', size=64, activation="tanh"),
    dict(type='dense', size=32, activation="tanh")
]

a_strategy = TensorforceTradingStrategy(environment=environment,
                                        agent_spec=agent_spec,
                                        network_spec=network_spec)

b_strategy = StableBaselinesTradingStrategy(environment=environment,
                                            model=PPO2,
                                            policy='MlpLnLSTMPolicy')
```

1.13 TradingEnvironment

A trading environment is a reinforcement learning environment that follows OpenAI's `gym.Env` specification. This allows us to leverage many of the existing reinforcement learning models in our trading agent, if we'd like.

`TradingEnvironment` steps through the various interfaces from the `tensortrade` library in a consistent way, and will likely not change too often as all other parts of `tensortrade` changes. We're going to go through an overview of the Trading environment below.

Trading environments are fully configurable gym environments with highly composable `Exchange`, `FeaturePipeline`, `ActionScheme`, and `RewardScheme` components.

- The `Exchange` provides observations to the environment and executes the agent's trades.
- The `FeaturePipeline` optionally transforms the exchange output into a more meaningful set of features before it is passed to the agent.
- The `ActionScheme` converts the agent's actions into executable trades.
- The `RewardScheme` calculates the reward for each time step based on the agent's performance.

That's all there is to it, now it's just a matter of composing each of these components into a complete environment.

When the reset method of a `TradingEnvironment` is called, all of the child components will also be reset. The internal state of each exchange, feature pipeline, transformer, action scheme, and reward scheme will be set back to their default values, ready for the next episode.

Let's begin with an example environment. As mentioned before, initializing a `TradingEnvironment` requires an exchange, an action scheme, and a reward scheme, the feature pipeline is optional.

1.13.1 OpenAI Gym Primer

Usually the OpenAI gym runs in the following way:

```
# Declare the environment
env = TrainingEnvironment()
# Declare and agent with an action_space, usually declared inside of the environment␣
↪itself
agent = RandomAgent(env.action_space)
reward = 0
```

(continues on next page)

(continued from previous page)

```
done = False

# Reset all of the variables
ob = env.reset() # Gets an observation as a response to resetting the variables
while True:
    # Get an observation, and input the previous reward, and indicator if the episode␣
↪is complete or not (done).
    action = agent.act(ob, reward, done)
    ob, reward, done, _ = env.step(action)
    if done:
        break
```

As such, the TradingEnvironment runs largely like this as well.

```
from tensortrade.environments import TradingEnvironment
from tensortrade.strategies import StableBaselinesTradingStrategy


environment = TradingEnvironment(exchange=exchange,
                                 action_scheme=action_scheme,
                                 reward_scheme=reward_scheme,
                                 feature_pipeline=feature_pipeline)

strategy.environment = environment
test_performance = strategy.run(episodes=1, testing=True)
```

Here you may notice that we don't have the same training code we saw above:

```
while True:
    # Get an observation, and input the previous reward, and indicator if the episode␣
↪is complete or not (done).
    action = agent.act(ob, reward, done)
    ob, reward, done, _ = env.step(action)
    if done:
        break
```

That's because the code to run that exist directly inside of the `TradingStrategy` codebase. The command `run`, has abstractions of that code. Please refer to the Strategies codebase.

1.13.2 Functions:

To better understand what's inside of the `TradingEnvironment`, you should understand the notation. Everything that begins with an underscore _ is a relatively private function. While everything that doesn't have the underscore is a public facing function.

Private

- `_take_action`
  - Determines a specific trade to be taken and executes it within the exchange.
- `_next_observation`
  - Returns the next observation from the exchange.
- `_get_reward`

    - Returns the reward for the current timestep.

- `_done`
    - Returns whether or not the environment is done and should be restarted. The two key conditions to determine if the environment is completed is if either `90% of the funds are lost` or if there are `no more observations left.`
- `_info`
    - Returns any auxiliary, diagnostic, or debugging information for the current timestep.

Public

- `step`
    - Run one timestep within the environment based on the specified action.
- `reset`
    - Resets the state of the environment and returns an initial observation.
- `render`
    - This sends an output of what's occuring in the gym enviornment for the user to keep track of.

Almost 100% of the private functions belong in the step function.

1.14 Exchange

Exchanges determine the universe of tradable instruments within a trading environment, return observations to the environment on each time step, and execute trades made within the environment. There are two types of exchanges: live and simulated.

Live exchanges are implementations of `Exchange` backed by live pricing data and a live trade execution engine. For example, `CCXTExchange` is a live exchange, which is capable of returning pricing data and executing trades on hundreds of live cryptocurrency exchanges, such as Binance and Coinbase.

```
import ccxt
from tensortrade.exchanges.live import CCXTExchange

coinbase = ccxt.coinbasepro()
exchange = CCXTExchange(exchange=coinbase, base_instrument='USD')
```

There are also exchanges for stock and ETF trading, such as RobinhoodExchange and InteractiveBrokersExchange, but these are still works in progress.

Simulated exchanges, on the other hand, are implementations of `Exchange` backed by simulated pricing data and trade execution.

For example, `FBMExchange` is a simulated exchange, which generates pricing and volume data using fractional brownian motion (FBM). Since its price is simulated, the trades it executes must be simulated as well. The exchange uses a simple slippage model to simulate price and volume slippage on trades, though like almost everything in TensorTrade, this slippage model can easily be swapped out for something more complex.

```
from tensortrade.exchanges.simulated import FBMExchange

exchange = FBMExchange(base_instrument='BTC', timeframe='1h')
```

Though the `FBMExchange` generates fake price and volume data using a stochastic model, it is simply an implementation of `SimulatedExchange`. Under the hood, `SimulatedExchange` only requires a `data_frame` of price history to generate its simulations. This `data_frame` can either be provided by a coded implementation such as `FBMExchange`, or at runtime.

```
import pandas as pd
from tensortrade.exchanges.simulated import SimulatedExchange

df = pd.read_csv('./data/btc_ohclv_1h.csv')
exchange = SimulatedExchange(data_frame=df, base_instrument='USD')
```

1.14.1 Purpose of `Exchange`?

Inside of the README, you'll have seen the reason why we have the higher level abstract. It reiterate, `Exchange` is the highest level abstract for all other exchanges. It has functions used through all others.

1.14.2 Class Parameters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.

1.14.3 Properties and Setters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.
- `base_precision`
  - The floating point precision of the base instrument.
- `instrument_precision`
  - The floating point precision of the instrument to be traded.
- `initial_balance`
  - The initial balance of the base symbol on the exchange.
- `balance`
  - The current balance of the base symbol on the exchange.
- `portfolio`

  - The current balance of each symbol on the exchange (non-positive balances excluded).
- `trades`
  - A list of trades made on the exchange since the last reset.
- `performance`
  - The performance of the active account on the exchange since the last reset.
- `generated_space`
  - The initial shape of the observations generated by the exchange, before feature transformations.
- `generated_columns`
  - The list of column names of the observation data frame generated by the exchange, before feature transformations.
- `observation_space`
  - The final shape of the observations generated by the exchange, after feature transformations.
- `net_worth`
  - Calculate the net worth of the active account on the exchange.
- `profit_loss_percent`
  - Calculate the percentage change in net worth since the last reset.
- `has_next_observation`
  - If `False`, the exchange's data source has run out of observations.
  - Resetting the exchange may be necessary to continue generating observations.

1.14.4 Functions

Below are the functions that the `Exchange` uses to effectively operate.

Private

- `_create_observation_generator`

Public

- `has_next_observation`
  - Return the reward corresponding to the selected risk-adjusted return metric.
- `next_observation`
  - Generate the next observation from the exchange.
- `instrument_balance`
  - The current balance of the specified symbol on the exchange, denoted in the base instrument.
- `current_price`
  - The current price of an instrument on the exchange, denoted in the base instrument.
- `execute_trade`

  - Execute a trade on the exchange, accounting for slippage.
- `reset`
  - Reset the feature pipeline, initial balance, trades, performance, and any other temporary stateful data.

1.14.5 Use Cases

...

1.15 SimulatedExchange

An exchange, in which the price history is based off the supplied data frame and trade execution is largely decided by the designated slippage model. If the `data_frame` parameter is not supplied upon initialization, it must be set before the exchange can be used within a trading environment.

The reason we create simulated exchanges is so we create more robust models quickly. We train using either sampled or fake information. Usually the data sets are stochastic in nature.

1.15.1 Class Parameters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.
- `kwargs`
  - Go through each of them here

1.15.2 Properties and Setters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.
- `base_precision`
  - The floating point precision of the base instrument.
- `instrument_precision`
  - The floating point precision of the instrument to be traded.
- `initial_balance`

  - The initial balance of the base symbol on the exchange.
- `balance`
  - The current balance of the base symbol on the exchange.
- `portfolio`
  - The current balance of each symbol on the exchange (non-positive balances excluded).
- `trades`
  - A list of trades made on the exchange since the last reset.
- `performance`
  - The performance of the active account on the exchange since the last reset.
- `generated_space`
  - The initial shape of the observations generated by the exchange, before feature transformations.
- `generated_columns`
  - The list of column names of the observation data frame generated by the exchange, before feature transformations.
- `observation_space`
  - The final shape of the observations generated by the exchange, after feature transformations.
- `net_worth`
  - Calculate the net worth of the active account on the exchange.
- `profit_loss_percent`
  - Calculate the percentage change in net worth since the last reset.
- `has_next_observation`
  - If `False`, the exchange's data source has run out of observations.
  - Resetting the exchange may be necessary to continue generating observations.

1.15.3 Functions

Below are the functions that the `SimulatedExchange` uses to effectively operate.

Private

- `_create_observation_generator`

Public

- `has_next_observation`
  - Return the reward corresponding to the selected risk-adjusted return metric.
- `next_observation`
  - Generate the next observation from the exchange.
- `instrument_balance`

  - The current balance of the specified symbol on the exchange, denoted in the base instrument.

- `current_price`
  - The current price of an instrument on the exchange, denoted in the base instrument.
- `execute_trade`
  - Execute a trade on the exchange, accounting for slippage.
- `reset`
  - Reset the feature pipeline, initial balance, trades, performance, and any other temporary stateful data.

1.15.4 Use Cases

```
import pandas as pd
from tensortrade.exchanges.simulated import SimulatedExchange

df = pd.read_csv('./data/btc_ohclv_1h.csv')
exchange = SimulatedExchange(data_frame=df, base_instrument='USD')
```

1.15.5 Use Cases

Working on it.

1.16 FBMExchange

Fractal Brownian Motion

A simulated exchange, in which the price history is based off a fractional brownian motion model with supplied parameters.

1.16.1 What is Fractal Brownian Motion?

Fractial Brownian Motion is apart of a class of differential equations called stochastic processes.

`Stochastic processes` are an accumulation of random variables that help us describe the emergence of a system over time. The power of them is that they can be used to describe all of the world around us. In fact, during early 1900 one of the first active uses of Stochastic Processes was with valuing stock options. It was called a Brownian Motion, developed by a French mathematician named Louis Bachelier. It can also be used for looking at random interactions of molecules over time. We use this method to solve reinforcement learning's sample inefficiency problem.

.

We generate prices, and train on that. Simple.

1.16.2 Class Parameters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.

- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.

1.16.3 Properties and Setters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.
- `base_precision`
  - The floating point precision of the base instrument.
- `instrument_precision`
  - The floating point precision of the instrument to be traded.
- `initial_balance`
  - The initial balance of the base symbol on the exchange.
- `balance`
  - The current balance of the base symbol on the exchange.
- `portfolio`
  - The current balance of each symbol on the exchange (non-positive balances excluded).
- `trades`
  - A list of trades made on the exchange since the last reset.
- `performance`
  - The performance of the active account on the exchange since the last reset.
- `generated_space`
  - The initial shape of the observations generated by the exchange, before feature transformations.
- `generated_columns`
  - The list of column names of the observation data frame generated by the exchange, before feature transformations.
- `observation_space`
  - The final shape of the observations generated by the exchange, after feature transformations.
- `net_worth`
  - Calculate the net worth of the active account on the exchange.
- `profit_loss_percent`

  - Calculate the percentage change in net worth since the last reset.
- `has_next_observation`
  - If `False`, the exchange's data source has run out of observations.
  - Resetting the exchange may be necessary to continue generating observations.

1.16.4 Functions

Below are the functions that the `FBMExchange` uses to effectively operate.

Private

- `_create_observation_generator`

Public

- `has_next_observation`
  - Return the reward corresponding to the selected risk-adjusted return metric.
- `next_observation`
  - Generate the next observation from the exchange.
- `instrument_balance`
  - The current balance of the specified symbol on the exchange, denoted in the base instrument.
- `current_price`
  - The current price of an instrument on the exchange, denoted in the base instrument.
- `execute_trade`
  - Execute a trade on the exchange, accounting for slippage.
- `reset`
  - Reset the feature pipeline, initial balance, trades, performance, and any other temporary stateful data.

1.16.5 Use Cases

Use Case #1: Generate Price History for Exchange

We generate the price history when

```python
from tensortrade.exchanges.simulated import FBMExchange

exchange = FBMExchange(base_instrument='BTC', timeframe='1h')
```

1.17 CCXTExchange

An exchange for trading on CCXT-supported cryptocurrency exchanges.

1.17.1 Class Parameters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.

1.17.2 Properties and Setters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.
- `base_precision`
  - The floating point precision of the base instrument.
- `instrument_precision`
  - The floating point precision of the instrument to be traded.
- `initial_balance`
  - The initial balance of the base symbol on the exchange.
- `balance`
  - The current balance of the base symbol on the exchange.
- `portfolio`
  - The current balance of each symbol on the exchange (non-positive balances excluded).
- `trades`
  - A list of trades made on the exchange since the last reset.
- `performance`
  - The performance of the active account on the exchange since the last reset.
- `generated_space`
  - The initial shape of the observations generated by the exchange, before feature transformations.
- `generated_columns`
  - The list of column names of the observation data frame generated by the exchange, before feature transformations.
- `observation_space`
  - The final shape of the observations generated by the exchange, after feature transformations.

- `net_worth`
  - Calculate the net worth of the active account on the exchange.
- `profit_loss_percent`
  - Calculate the percentage change in net worth since the last reset.
- `has_next_observation`
  - If `False`, the exchange's data source has run out of observations.
  - Resetting the exchange may be necessary to continue generating observations.

1.17.3 Functions

Below are the functions that the `Exchange` uses to effectively operate.

Private

- `_create_observation_generator`

Public

- `has_next_observation`
  - Return the reward corresponding to the selected risk-adjusted return metric.
- `next_observation`
  - Generate the next observation from the exchange.
- `instrument_balance`
  - The current balance of the specified symbol on the exchange, denoted in the base instrument.
- `current_price`
  - The current price of an instrument on the exchange, denoted in the base instrument.
- `execute_trade`
  - Execute a trade on the exchange, accounting for slippage.
- `reset`
  - Reset the feature pipeline, initial balance, trades, performance, and any other temporary stateful data.

1.17.4 Use Cases

Use Case #1: Initiating CCXTExchnage

```python
import ccxt
from tensortrade.exchanges.live import CCXTExchange

coinbase = ccxt.coinbasepro()
exchange = CCXTExchange(exchange=coinbase, base_instrument='USD')
```

1.18 InteractiveBrokersExchange

An exchange for trading using the Interactive Brokers API.

1.18.1 Class Parameters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.

1.18.2 Properties and Setters

- `base_instrument`
  - The exchange symbol of the instrument to store/measure value in.
- `dtype`
  - A type or str corresponding to the dtype of the `observation_space`.
- `feature_pipeline`
  - A pipeline of feature transformations for transforming observations.
- `base_precision`
  - The floating point precision of the base instrument.
- `instrument_precision`
  - The floating point precision of the instrument to be traded.
- `initial_balance`
  - The initial balance of the base symbol on the exchange.
- `balance`
  - The current balance of the base symbol on the exchange.
- `portfolio`
  - The current balance of each symbol on the exchange (non-positive balances excluded).
- `trades`
  - A list of trades made on the exchange since the last reset.
- `performance`
  - The performance of the active account on the exchange since the last reset.
- `generated_space`
  - The initial shape of the observations generated by the exchange, before feature transformations.
- `generated_columns`

  - The list of column names of the observation data frame generated by the exchange, before feature transformations.

- `observation_space`
  - The final shape of the observations generated by the exchange, after feature transformations.
- `net_worth`
  - Calculate the net worth of the active account on the exchange.
- `profit_loss_percent`
  - Calculate the percentage change in net worth since the last reset.
- `has_next_observation`
  - If `False`, the exchange's data source has run out of observations.
  - Resetting the exchange may be necessary to continue generating observations.

1.18.3 Functions

Below are the functions that the `Exchange` uses to effectively operate.

Private

- `_create_observation_generator`
  - ...

Public

- `has_next_observation`
  - Return the reward corresponding to the selected risk-adjusted return metric.
- `next_observation`
  - Generate the next observation from the exchange.
- `instrument_balance`
  - The current balance of the specified symbol on the exchange, denoted in the base instrument.
- `current_price`
  - The current price of an instrument on the exchange, denoted in the base instrument.
- `execute_trade`
  - Execute a trade on the exchange, accounting for slippage.
- `reset`
  - Reset the feature pipeline, initial balance, trades, performance, and any other temporary stateful data.

1.18.4 Use Cases

Use Case #1: Initiating Interactive Broker

We generate the price history when

```
from tensortrade.exchanges.live import InteractiveBrokersExchange

exchange = InteractiveBrokersExchange(base_instrument='BTC', timeframe='1h')
```

1.19 FeaturePipeline

Feature pipelines are meant for transforming observations from the environment into meaningful features for an agent to learn from. If a pipeline has been added to a particular exchange, then observations will be passed through the `FeaturePipeline` before being output to the environment.

For example, a feature pipeline could normalize all price values, make a time series stationary, add a moving average column, and remove an unnecessary column, all before the observation is returned to the agent.

Feature pipelines can be initialized with an arbitrary number of comma-separated transformers. Each `FeatureTransformer` needs to be initialized with the set of columns to transform, or if nothing is passed, all input columns will be transformed.

Each feature transformer has a transform method, which will transform a single observation (a `pandas.DataFrame`) from a larger data set, keeping any necessary state in memory to transform the next frame. For this reason, it is often necessary to reset the FeatureTransformer periodically. This is done automatically each time the parent `FeaturePipeline` or `Exchange` is reset.

1.19.1 How It Operates

The `FeaturePipeline` has a setup that resembles the `keras` library. The concept is simple:

1. We take in an observation of data (price information), usually in the form of a pandas dataframe.
2. We take the observation and effectively run it through all declared ways of transforming that data inside of the FeaturePipeline and turn the result as a `gym.space`.

Just like kera's `Sequential` module, it accepts a list inside of its constructor and iterates through each piece on call. To draw on parallels, look at `keras`:

```
model = Sequential([
    Dense(32, input_shape=(500,)),
    Dense(32)
])
```

1.19.2 Class Parameters

- `steps`
  - A list of feature transformations to apply to observations.
- `dtype`
  - The `dtype` elements in the pipeline should be cast to.

1.19.3 Properties and Setters

- `steps`
  - A list of feature transformations to apply to observations.
- `dtype`
  - The `dtype` that elements in the pipeline should be input and output as.
- `reset`
  - Reset all transformers within the feature pipeline.

1.19.4 Functions

Below are the functions that the `FeaturePipeline` uses to effectively operate.

Private

- `_transform`
  - Utility method for transforming observations via a list of *make changes here* `FeatureTransformer` objects.
  - In other words, it runs through all of the `steps` in a for loop, and casts the response.

The code from the transform function: As you see, it iterates through every step and adds the observation to the dataframe.

```
for transformer in self._steps:
    observations = transformer.transform(observations)
```

At the end the observations are converted into a ndarray so that they can be interpreted by the agent.

Public

- `reset`
  - Reset all transformers within the feature pipeline.
- `transform`
  - Apply the pipeline of feature transformations to an observation frame.

1.19.5 Use Cases

Use Case #1: Initiate Pipeline

```
from tensortrade.features import FeaturePipeline
from tensortrade.features.scalers import MinMaxNormalizer
from tensortrade.features.stationarity import FractionalDifference
from tensortrade.features.indicators import SimpleMovingAverage

price_columns = ["open", "high", "low", "close"]
normalize_price = MinMaxNormalizer(price_columns)
moving_averages = SimpleMovingAverage(price_columns)
```

(continues on next page)

(continued from previous page)

```
difference_all = FractionalDifference(difference_order=0.6)
feature_pipeline = FeaturePipeline(steps=[normalize_price,
                                          moving_averages,
                                          difference_all])

exchange.feature_pipeline = feature_pipeline
```

1.20 FeatureTransformer

As stated before in the overview, We use an `ABCMeta` abstract hierarchy to handle the transformation calls of each asset. The `FeatureTransformer` is an abstract of all other price transformers available inside of the `tensortrade` library. As such, it has a set of common functions that are called on almost every transformer.

1.20.1 Properties and Setters

- `columns`
  - A list of column names to normalize

1.20.2 Functions

Below are the functions that the `FeatureTransformer` uses to effectively operate.

Private

`None`

Public

- `reset`
  - Optionally implementable method for resetting stateful transformers.
- `transform`
  - Transform the data set and return a new data frame.

1.21 MinMaxNormalizer

A transformer for normalizing values within a feature pipeline by the column-wise extrema.

1.21.1 Class Parameters

- `columns`
  - A list of column names to normalize.
- `feature_min`

  - The minimum value in the range to scale to.
- feature_max
  - The maximum value in the range to scale to.
- inplace
  - If False, a new column will be added to the output for each input column.

1.21.2 Properties and Setters

None

1.21.3 Functions

Below are the functions that the MinMaxNormalizer uses to effectively operate.

Private

None

Public

- transform
  - Apply the pipeline of feature transformations to an observation frame.

1.21.4 Use Cases:

```
from tensortrade.features import FeaturePipeline
from tensortrade.features.scalers import MinMaxNormalizer
from tensortrade.features.stationarity import FractionalDifference
from tensortrade.features.indicators import SimpleMovingAverage
price_columns = ["open", "high", "low", "close"]
normalize_price = MinMaxNormalizer(price_columns)
moving_averages = SimpleMovingAverage(price_columns)
difference_all = FractionalDifference(difference_order=0.6)
feature_pipeline = FeaturePipeline(steps=[normalize_price,
                                          moving_averages,
                                          difference_all])
exchange.feature_pipeline = feature_pipeline
```

1.22 StandardNormalizer

A transformer for normalizing values within a feature pipeline by removing the mean and scaling to unit variance.

1.22.1 Class Parameters

- `columns`
  - A list of column names to normalize.
- `feature_min`
  - The minimum value in the range to scale to.
- `feature_max`
  - The maximum value in the range to scale to.
- `inplace`
  - If `False`, a new column will be added to the output for each input column.

1.22.2 Properties and Setters

- None

1.22.3 Functions

Below are the functions that the `StandardNormalizer` uses to effectively operate.

Private

None

Public

- `transform`
  - Apply the pipeline of feature transformations to an observation frame.
- `reset`
  - Resets the history of the standard scaler.

1.22.4 Use Cases:

Use Case #1: Different Input Spaces

This `StandardNormalizer` operates differently depending on if we pretransform the observation to an ndarray or keep it as a pandas dataframe.

```
from tensortrade.features import FeaturePipeline
from tensortrade.features.scalers import StandardNormalizer
from tensortrade.features.stationarity import FractionalDifference
from tensortrade.features.indicators import SimpleMovingAverage
price_columns = ["open", "high", "low", "close"]
normalize_price = MinMaxNormalizer(price_columns)
moving_averages = SimpleMovingAverage(price_columns)
difference_all = FractionalDifference(difference_order=0.6)
```

(continues on next page)

(continued from previous page)

```
feature_pipeline = FeaturePipeline(steps=[normalize_price,
                                          moving_averages,
                                          difference_all])
exchange.feature_pipeline = feature_pipeline
```

1.23 FractionalDifference

A transformer for differencing values within a feature pipeline by a fractional order. It removes the stationarity of the dataset available in realtime. To learn more about why non-stationarity should be converted to stationary information, please look at the blog here.

1.23.1 Class Parameters

- `columns`
  - A list of column names to difference.
- `difference_order`
  - The fractional difference order. Defaults to 0.5.
- `difference_threshold`
  - A type or str corresponding to the dtype of the `observation_space`.
- `inplace`
  - If `False`, a new column will be added to the output for each input column.

1.23.2 Functions

Below are the functions that the `FractionalDifference` uses to effectively operate.

Private

- `_difference_weights`
  - Gets the weights for ...
- `_fractional_difference`
  - Computes fractionally differenced series, with an increasing window width.

Public

- `transform`
  - Apply the pipeline of feature transformations to an observation frame.
- `reset`
  - Resets the history of the standard scaler.

1.23.3 Use Cases:

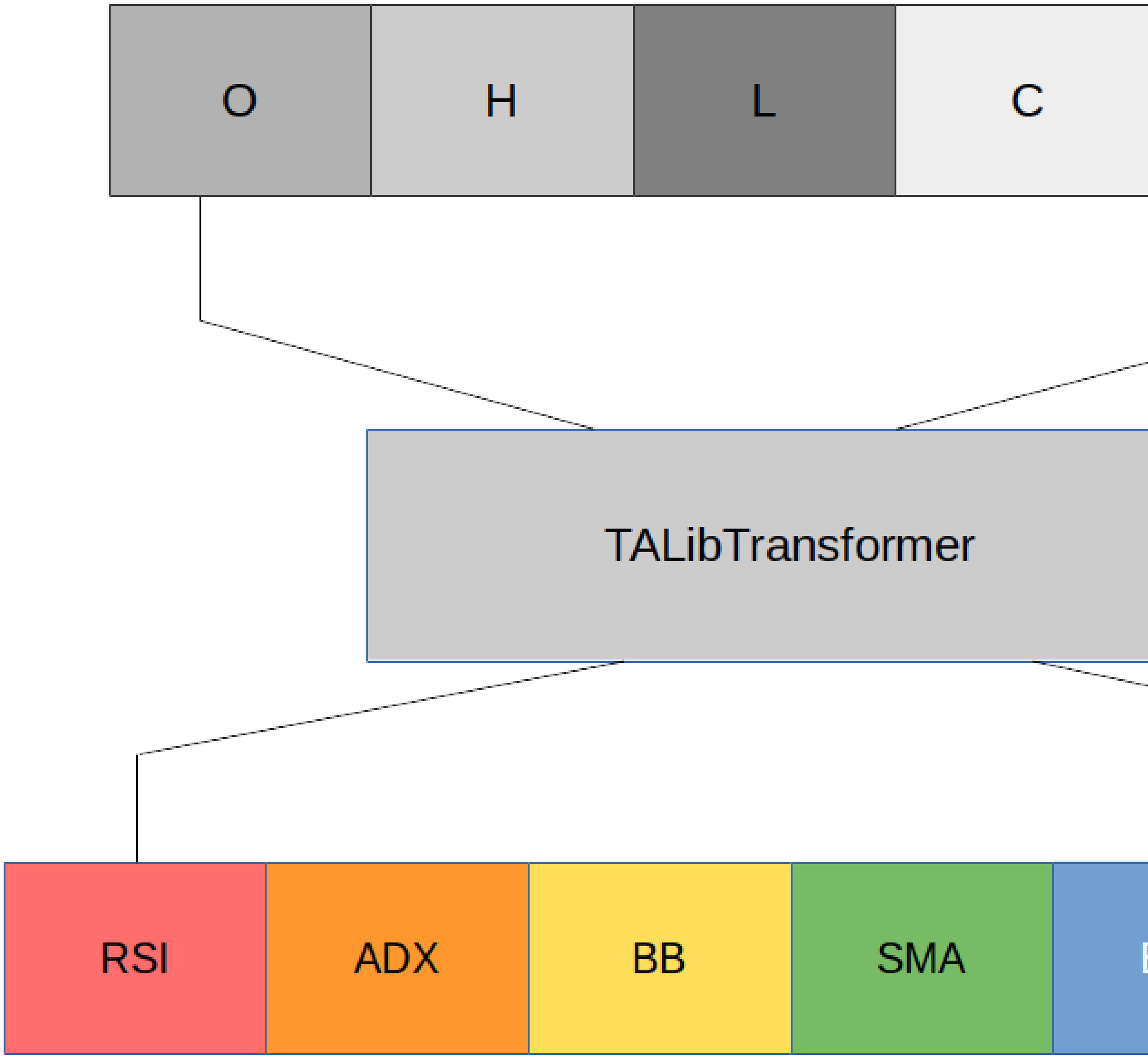
Use Case #1: Different Input Spaces

This `FeatureTransformer` operates differently depending on if we pretransform the observation to an ndarray or keep it as a pandas dataframe.

```
from tensortrade.features import FeaturePipeline
from tensortrade.features.stationarity import FractionalDifference
price_columns = ["open", "high", "low", "close"]
difference_all = FractionalDifference(difference_order=0.6) # fractional difference␣
↪is seen here
feature_pipeline = FeaturePipeline(steps=[difference_all])
exchange.feature_pipeline = feature_pipeline
```

1.24 TAlibIndicator

Adds one or more TAlib indicators to a data frame, based on existing open, high, low, and close column values.


O
H
L
C
TALibTransformer
RSI
ADX
BB
SMA

1.24.1 Class Parameters

- `indicators`
  - A list of indicators you want to transform the price information to.
- `lows`
  - The lower end of the observation space. See `spaces.Box` to best understand.
- `highs`
  - The lower end of the observation space. See `spaces.Box` to best understand.

1.24.2 Properties and Setters

- **NONE**

1.24.3 Functions

Below are the functions that the `TAlibIndicator` uses to effectively operate.

Private

- `_str_to_indicator` - Converts the name of an indicator to an actual instance of the indicator. For a list of indicators see list here.

Public

- `transform`
  - Transform the data set and return a new data frame.

1.24.4 Use Cases:

1.24.5 Use Cases

Use Case #1: Selecting Indicators

The key advantage the `TAlibIndicator` has is that it allows us to dynamically set indicators according what's inside of a list. For instance, if we're trying to get the RSI and EMA together, we would run the following parameters inside ofthe indicator.

```
talib_indicator = TAlibIndicator(["rsi", "ema"])
```

This runs through the indicators in the list, at runtime and matches them to what is seen inside of TA-Lib. The features are then flattened into the `output_space`, both into the `high` and `low` segment of `space.Box`.

```
for i in range(len(self._indicators)):
    output_space.low = np.append(output_space.low, self._lows[i])
    output_space.high = np.append(output_space.high, self._highs[i])
```

Actual Use

```python
from tensortrade.features import FeaturePipeline
from tensortrade.features.scalers import MinMaxNormalizer
from tensortrade.features.stationarity import FractionalDifference
from tensortrade.features.indicators import TAlibIndicator
price_columns = ["open", "high", "low", "close"]
normalize_price = MinMaxNormalizer(price_columns)
moving_averages = TAlibIndicator(["EMA", "RSI", "BB"])
difference_all = FractionalDifference(difference_order=0.6)
feature_pipeline = FeaturePipeline(steps=[normalize_price,
                                          moving_averages,
                                          difference_all])
exchange.feature_pipeline = feature_pipeline
```

1.25 ActionScheme

Action schemes define the action space of the environment and convert an agent's actions into executable trades.

For example, if we were using a discrete action space of 3 actions (0 = hold, 1 = buy 100 %, 2 = sell 100%), our learning agent does not need to know that returning an action of 1 is equivalent to buying an instrument. Rather, our agent needs to know the reward for returning an action of 1 in specific circumstances, and can leave the implementation details of converting actions to trades to the `ActionScheme`.

Each action scheme has a get_trade method, which will transform the agent's specified action into an executable `Trade`. It is often necessary to store additional state within the scheme, for example to keep track of the currently traded position. This state should be reset each time the action scheme's reset method is called, which is done automatically when the parent `TradingEnvironment` is reset.

1.25.1 What is an Action?

This is a review of what was mentioned inside of the overview section. It explains how a RL operates. You'll better understand what an action is in context of an observation space and reward. At the same time, hopefully this will be a proper refresher.

An action is a predefined value of how the machine should move inside of the world. To better summarize, its a *command that a player would give inside of a video game in respose to a stimuli*. The commands usually come in the form of an `action_space`. An `action_space` is something that represents how to make the user move inside of an environment. While it might not be easily interpretable by humans, it can easily be interpreted by a machine.

Let's look at a good example. Lets say we're trying to balance a cart with a pole on it (cartpole). We can choose to move the cart left and right. This is a `Discrete(2)` action type.

- 0 - Push cart to the left
- 1 - Push cart to the right

When we get the action from the RL agent, the environment will see that number instead of a name. We can create lists, tuples, and a box.

1.25.2 Setters & Properties

Each property and property setter.

- `dtype`

  - A type or str corresponding to the dtype of the `action_space`.
- `exchange`
  - The exchange being used by the current trading environment.
  - This will be set by the trading environment upon initialization. Setting the exchange causes the scheme to reset.
- `action_space`
  - The shape of the actions produced by the scheme. This takes in a `gym.space` and is different for each given scheme.

1.25.3 Functions

- reset
  - Optionally implementable method for resetting stateful schemes.
- get_trade
  - Get the trade to be executed on the exchange based on the action provided.
  - Usually this is the way we distill the information generated from the `action_space`.

1.26 ContinuousActions

Simple continuous scheme, which calculates the trade amount as a fraction of the total balance.

1.26.1 Key Variables

- `max_allowed_slippage`
  - The exchange symbols of the instruments being traded.
- `instrument`
  - The number of bins to divide the total balance by. Defaults to 20 (i.e. 1/20, 2/20, . . . , 20/20).
- `instrument`
  - The maximum amount above the current price the scheme will pay for an instrument. Defaults to 1.0 (i.e. 1%).

1.26.2 Setters & Properties

Each property and property setter.

- `dtype`
  - A type or str corresponding to the dtype of the `action_space`.
- `exchange`
  - The exchange being used by the current trading environment.
  - This will be set by the trading environment upon initialization. Setting the exchange causes the scheme to reset.

- `action_space`
  - The shape of the actions produced by the scheme. This takes in a `gym.space` and is different for each given scheme.

1.26.3 Functions

- `reset`
  - Optionally implementable method for resetting stateful schemes.
- `get_trade`
  - Get the trade to be executed on the exchange based on the action provided.
  - Usually this is the way we distill the information generated from the `action_space`.

1.26.4 Use Cases

TODO: Place Use Case Here

1.27 DiscreteActions

Simple discrete scheme, which calculates the trade amount as a fraction of the total balance.

1.27.1 Key Variables

- `instruments`
  - The exchange symbols of the instruments being traded.
- `actions_per_instrument`
  - The number of bins to divide the total balance by. Defaults to 20 (i.e. 1/20, 2/20, . . . , 20/20).
- `max_allowed_slippage_percent`
  - The maximum amount above the current price the scheme will pay for an instrument. Defaults to 1.0 (i.e. 1%).

1.27.2 Setters & Properties

Each property and property setter.

- `dtype`
  - A type or str corresponding to the dtype of the `action_space`.
- `exchange`
  - The exchange being used by the current trading environment.
  - This will be set by the trading environment upon initialization. Setting the exchange causes the scheme to reset.
- `action_space`

  - The shape of the actions produced by the scheme. This takes in a `gym.space` and is different for each given scheme.

1.27.3 Functions

- `reset`
  - Optionally implementable method for resetting stateful schemes.
- `get_trade`
  - Get the trade to be executed on the exchange based on the action provided.
  - Usually this is the way we distill the information generated from the `action_space`.

1.27.4 Use Cases

```
from tensortrade.actions import DiscreteActions

action_scheme = DiscreteActions(n_actions=20, instrument='BTC')
```

This discrete action scheme uses 20 discrete actions, which equates to 4 discrete amounts for each of the 5 trade types (market buy/sell, limit buy/sell, and hold). E.g. [0,5,10,15]=hold, 1=market buy 25%, 2=market sell 25%, 3=limit buy 25%, 4=limit sell 25%, 6=market buy 50%, 7=market sell 50%, etc…

1.28 MultiDiscreteActions

Discrete scheme, which calculates the trade amount as a fraction of the total balance for each instrument provided.

The trade type is determined by `action % len(TradeType)`, and the trade amount is determined by the multiplicity of the action. For example, `0 = HOLD`, `1 = LIMIT_BUY|0.25`, `2 = MARKET_BUY|0.25`, `5 = HOLD`, `6 = LIMIT_BUY|0.5`, `7 = MARKET_BUY|0.5`, etc.

1.28.1 Key Variables

- `_instruments`
  - The exchange symbols of the instruments being traded.
- `_actions_per_instrument`
  - The number of bins to divide the total balance by. Defaults to 20 (i.e. 1/20, 2/20, . . . , 20/20).
- `_max_allowed_slippage_percent`
  - The maximum amount above the current price the scheme will pay for an instrument. Defaults to 1.0 (i.e. 1%).

1.28.2 Setters & Properties

Each property and property setter.

- `dtype`
  - A type or str corresponding to the dtype of the `action_space`.

- `exchange`
  - The exchange being used by the current trading environment.
  - This will be set by the trading environment upon initialization. Setting the exchange causes the scheme to reset.
- `action_space`
  - The shape of the actions produced by the scheme. This takes in a `gym.space` and is different for each given scheme.

1.28.3 Functions

- `reset`
  - Optionally implementable method for resetting stateful schemes.
- `get_trade`
  - Get the trade to be executed on the exchange based on the action provided.
  - Usually this is the way we distill the information generated from the `action_space`.

1.28.4 Use Cases

```
from tensortrade.actions import MultiDiscreteActions

action_scheme = MultiDiscreteActions(n_actions=20, instrument='BTC')
```

This discrete action scheme uses 20 discrete actions, which equates to 4 discrete amounts for each of the 5 trade types (market buy/sell, limit buy/sell, and hold). E.g. [0,5,10,15]=hold, 1=market buy 25%, 2=market sell 25%, 3=limit buy 25%, 4=limit sell 25%, 6=market buy 50%, 7=market sell 50%, etc…

1.29 Reward Scheme

Reward schemes receive the `Trade` taken at each time step and return a `float`, corresponding to the benefit of that specific action. For example, if the action taken this step was a sell that resulted in positive profits, our `RewardScheme` could return a positive number to encourage more trades like this. On the other hand, if the action was a sell that resulted in a loss, the scheme could return a negative reward to teach the agent not to make similar actions in the future.

A version of this example algorithm is implemented in `SimpleProfit`, however more complex schemes can obviously be used instead.

Each reward scheme has a `get_reward` method, which takes in the trade executed at each time step and returns a `float` corresponding to the value of that action. As with action schemes, it is often necessary to store additional state within a reward scheme for various reasons. This state should be reset each time the reward scheme's reset method is called, which is done automatically when the parent `TradingEnvironment` is reset.

Ultimately the agent creates a sequence of actions to maximize its total reward over a given time. The `RewardScheme` is an abstract class that encapsulates how to tell the trading bot in `tensortrade` if it's trading positively or negatively over time. The same methods will be called each time for each step, and we can directly swap out schemes.

1.29.1 Properties and Setters

- `exchange`
  - The central exchange for the scheme.
  - The exchange being used by the current trading environment. Setting the exchange causes the scheme to reset.

1.29.2 Methods

- `get_reward`
  - Gets the reward for the RL agent.
  - Returns a float corresponding to the benefit earned by the action taken this timestep.
- `reset`
  - Resets the current state if the reward has a state.
  - Optionally implementable method for resetting stateful schemes.

1.30 Risk Adjusted Returns

A reward scheme that rewards the agent for increasing its net worth, while penalizing more volatile strategies.

1.30.1 What are risk adjusted models?

When trading you often are not just looking at the overall returns of your model. You're also looking at the overall volatility of your trading strategy over time compared to other metrics. The two major strategies here are the sharpe and sortino ratio.

The **sharpe ratio** looks at the overall movements of the portfolio and generates a penalty for massive movements through a lower score. This includes major movements towards the upside and downside.

$$S = \left(\frac{R_p - R_f}{\sigma_p} \right)$$

The **sortino ratio** takes the same idea, though it focuses more on penalizing only the upside. That means it'll give a huge score for moments when the price moves upward, and will only give a negative score when the price drops heavily. This is a great direction for the RL algorithm. Seeing that we don't want to incur heavy downsides, yet want to take on large upsides, using this metric alone gives us lots of progress to mititgate downsides and increase upsides.

$$S = \frac{(R - MAR)}{\text{downside deviation}}$$

1.30.2 Class Parameters

- `return_algorithm`
  - The risk-adjusted return metric to use. Options are ‘sharpe’ and ‘sortino’. Defaults to ‘sharpe’.
- `risk_free_rate`
  - The risk free rate of returns to use for calculating metrics. Defaults to 0.
- `target_returns`
  - The target returns per period for use in calculating the sortino ratio. Default to 0.

1.30.3 Functions

Below are the functions that the `RiskAdjustedReturns` uses to effectively operate.

Private

- `_return_algorithm_from_str`
  - Allows us to dynamically choose an algorithm for the reward within a given selection. We can choose between either sharpe or sortino ratios. Each are volatility models we’ve discussed above.
- `_sharpe_ratio`
  - Return the sharpe ratio for a given series of a returns.
- `_sortino_ratio`
  - Return the sortino ratio for a given series of a returns.

Public

- `get_reward`
  - Return the reward corresponding to the selected risk-adjusted return metric.

1.30.4 Use Cases

...

1.31 Simple Profit

A reward scheme that rewards the agent for profitable trades and prioritizes trading over not trading.

1.31.1 Class Parameters

None

1.31.2 Functions

Below are the functions that the `SimpleProfit` uses to effectively operate.

1.31.3 Private

None

1.31.4 Public

- `reset` - Reset variables
  - Necessary to reset the last purchase price and state of open positions
  - Variables it resets
    - `_purchase_price`
    - The price the bot purchased the asset
    - `_is_holding_instrument` - A boolean that shares with the get_reward function if we're currently holding onto a trade.
- `get_reward`
  - Returns the reward for the given action
  - The `5^(log_10(profit))` function simply slows the growth of the reward as trades get large.

1.31.5 Use Cases

The simple profit scheme needs to keep a history of profit over time. The way it does this is through looking at the portfolio as a means of keeping track of how the portfolio moves. It also keeps track to see if it's holding onto a trade as well. This is seen inside of the get_reward function.

Use Case #1: Buying

When the bot says buy, it sets the variable `_is_holding_instrument` to `True`, and sets the current price to the price of the trade.

We see that inside of this line of code here. This allows us to check to see if we've made a profit later.

```
elif trade.is_buy and trade.amount > 0:
    self._purchase_price = trade.price
    self._is_holding_instrument = True
```

Use Case #2: Selling

We then sell afterward using the original trade price as a reference. Which is suggested in the lines below:

```python
if trade.is_sell and trade.amount > 0:
    self._is_holding_instrument = False
    profit_per_instrument = trade.price - self._purchase_price
    profit = trade.amount * profit_per_instrument
    profit_sign = np.sign(profit)

    return profit_sign * (1 + (5 ** np.log10(abs(profit))))
```

1.32 Learning Agents

This is where the "deep" part of the deep reinforcement learning framework come in. Learning agents are where the math (read: magic) happens.

At each time step, the agent takes the observation from the environment as input, runs it through its underlying model (a neural network most of the time), and outputs the action to take. For example, the observation might be the previous `open`, `high`, `low`, and `close` price from the exchange. The learning model would take these values as input and output a value corresponding to the action to take, such as `buy`, `sell`, or `hold`.

It is important to remember the learning model has no intuition of the prices or trades being represented by these values. Rather, the model is simply learning which values to output for specific input values or sequences of input values, to earn the highest reward.

1.33 Stable Baselines

In this example, we will be using the Stable Baselines library to provide learning agents to our trading scheme, however, the TensorTrade framework is compatible with many reinforcement learning libraries such as Tensorforce, Ray's RLLib, OpenAI's Baselines, Intel's Coach, or anything from the TensorFlow line such as TF Agents.

It is possible that custom TensorTrade learning agents will be added to this framework in the future, though it will always be a goal of the framework to be interoperable with as many existing reinforcement learning libraries as possible, since there is so much concurrent growth in the space. But for now, Stable Baselines is simple and powerful enough for our needs.

```python
from stable_baselines.common.policies import MlpLnLstmPolicy
from stable_baselines import PPO2

model = PPO2
policy = MlpLnLstmPolicy
params = { "learning_rate": 1e-5 }

agent = model(policy, environment, model_kwargs=params)
```

Note: Stable Baselines is not required to use TensorTrade though it is required for this tutorial. This example uses a GPU-enabled Proximal Policy Optimization model with a layer-normalized LSTM perceptron network. If you would like to know more about Stable Baselines, you can view the Documentation.

1.34 Tensorforce

I will also quickly cover the Tensorforce library to show how simple it is to switch between reinforcement learning frameworks.

```
from tensorforce.agents import Agent

agent_spec = {
    "type": "ppo_agent",
    "step_optimizer": {
        "type": "adam",
        "learning_rate": 1e-4
    },
    "discount": 0.99,
    "likelihood_ratio_clipping": 0.2,
}

network_spec = [
    dict(type='dense', size=64, activation="tanh"),
    dict(type='dense', size=32, activation="tanh")
]

agent = Agent.from_spec(spec=agent_spec,
                        kwargs=dict(network=network_spec,
                                    states=environment.states,
                                    actions=environment.actions))
```

If you would like to know more about Tensorforce agents, you can view the Documentation.

1.35 TradingStrategy

A `TradingStrategy` consists of a learning agent and one or more trading environments to tune, train, and evaluate on. If only one environment is provided, it will be used for tuning, training, and evaluating. Otherwise, a separate environment may be provided at each step.

1.36 StableBaselinesTradingStrategy

A trading strategy capable of self tuning, training, and evaluating with stable-baselines.

1.36.1 Class Parameters

- `environment`
  - A `TradingEnvironment` instance for the agent to trade within.
- `agent`
  - A `Tensorforce` agent or agent specification.
- `model`
  - The runner will automatically save the best agent
- `policy`

    - The RL policy to train the agent's model with. Defaults to 'MlpPolicy'.

- `_model_kwargs`
    - ...

1.36.2 Properties and Setters

- `agent`
    - A `TradingEnvironment` instance for the agent to trade within.
- `max_episode_timesteps`
    - The maximum timesteps per episode.

1.36.3 Functions

- `restore_agent`
    - Deserialize the strategy's learning agent from a file.
    - **parameters**:
        * `path`
            · The `str` path of the file the agent specification is stored in.
- `save_agent`
    - Serialize the learning agent to a file for restoring later.
    - **parameters**:
        * `path`
            · The `str` path of the file to store the agent specification in.
- `tune`
    - Function `NotImplemented`
- `run`
    - Runs all of the episodes specified.
    - **parameters**:
        * steps
        * episodes
        * episode_callback
        * evaluation

1.36.4 Use Cases

Use Case #1: Run a Strategy

```
from stable_baselines import PPO2
from tensortrade.strategies import StableBaselinesTradingStrategy
from tensortrade.environments import TradingEnvironment

environment = TradingEnvironment(exchange=exchange,
                                 action_strategy=action_strategy,
                                 reward_strategy=reward_strategy)
b_strategy = StableBaselinesTradingStrategy(model=PPO2)
strategy.environment = environment
stategy.run(episodes=10)
```

Use Case #2: Run a Live Strategy

```
import ccxt
from tensortrade.environments import TradingEnvironment
from tensortrade.strategies import StableBaselinesTradingStrategy
from tensortrade.exchanges.live import CCXTExchange
coinbase = ccxt.coinbasepro(...) # your credentials go here in dictionary form
exchange = CCXTExchange(exchange=coinbase,
                        timeframe='1h',
                        base_instrument='USD',
                        feature_pipeline=feature_pipeline)
environment = TradingEnvironment(exchange=exchange,
                                 action_strategy=action_strategy,
                                 reward_strategy=reward_strategy)
strategy.environment = environment
strategy.restore_agent(path="../agents/ppo_btc/1h")
live_performance = strategy.run(steps=0, trade_callback=episode_cb)
```

1.37 TensorforceTradingStrategy

A trading strategy capable of self tuning, training, and evaluating with Tensorforce.

1.37.1 Class Parameters

- `environment`
  - A `TradingEnvironment` instance for the agent to trade within.
- `agent`
  - A `Tensorforce` agent or agent specification.
- `model`
  - The runner will automatically save the best agent
- `policy`
  - The RL policy to train the agent's model with. Defaults to 'MlpPolicy'.
- `_model_kwargs`

1.37.2 Properties and Setters

- `environment`

  - A `TradingEnvironment` instance for the agent to trade within.

1.37.3 Functions

- `restore_agent`
  - Deserialize the strategy's learning agent from a file.
  - **parameters**:
    * `path`
      · The `str` path of the file the agent specification is stored in.
- `save_agent`
  - Serialize the learning agent to a file for restoring later.
  - **parameters**:
    * `path`
      · The `str` path of the file to store the agent specification in.
- `tune`
  - Function `NotImplemented`
- `run`
  - Runs all of the episodes specified.
  - **parameters**:
    * steps
    * episodes
    * episode_callback

1.37.4 Use Cases

Use Case #1: Run a Strategy

```
from stable_baselines import PPO2
from tensortrade.strategies import TensorforceTradingStrategy


agent_spec = {
    "type": "ppo_agent",
    "step_optimizer": {
        "type": "adam",
        "learning_rate": 1e-4
    },
    "discount": 0.99,
    "likelihood_ratio_clipping": 0.2,
}

network_spec = [
    dict(type='dense', size=64, activation="tanh"),
    dict(type='dense', size=32, activation="tanh")
]
```

(continues on next page)

(continued from previous page)

```
a_strategy = TensorforceTradingStrategy(environment=environment,
                                        agent_spec=agent_spec,
                                        network_spec=network_spec)
a_strategy.run(episodes=10)
```

Use Case #2: Run a Live Strategy

```
import ccxt
from tensortrade.environments import TradingEnvironment
from tensortrade.strategies import StableBaselinesTradingStrategy
from tensortrade.exchanges.live import CCXTExchange
coinbase = ccxt.coinbasepro(...) # your credentials go here in dictionary form
exchange = CCXTExchange(exchange=coinbase,
                        timeframe='1h',
                        base_instrument='USD',
                        feature_pipeline=feature_pipeline)

environment = TradingEnvironment(exchange=exchange,
                                 action_strategy=action_strategy,
                                 reward_strategy=reward_strategy)

agent_spec = {
    "type": "ppo_agent",
    "step_optimizer": {
        "type": "adam",
        "learning_rate": 1e-4
    },
    "discount": 0.99,
    "likelihood_ratio_clipping": 0.2,
}

network_spec = [
    dict(type='dense', size=64, activation="tanh"),
    dict(type='dense', size=32, activation="tanh")
]

a_strategy = TensorforceTradingStrategy(environment=environment,
                                        agent_spec=agent_spec,
                                        network_spec=network_spec)

strategy.environment = environment
strategy.restore_agent(path="../agents/ppo_btc/1h")
live_performance = strategy.run(steps=0, trade_callback=episode_cb)
```

1.38 tensortrade

1.38.1 tensortrade package

Subpackages

tensortrade.actions package

Submodules

tensortrade.actions.action_scheme module

tensortrade.actions.continuous_actions module

tensortrade.actions.discrete_actions module

tensortrade.actions.multi_discrete_actions module

tensortrade.base package

Submodules

tensortrade.base.component module

tensortrade.base.context module

tensortrade.base.registry module

tensortrade.environments package

Subpackages

tensortrade.environments.render package

Submodules

tensortrade.environments.render.matplotlib_trading_chart module

Submodules

tensortrade.environments.trading_environment module

tensortrade.exchanges package

Subpackages

tensortrade.exchanges.live package

Submodules

tensortrade.exchanges.live.ccxt_exchange module

tensortrade.exchanges.live.interactive_brokers_exchange module

tensortrade.exchanges.live.robinhood_exchange module

tensortrade.exchanges.simulated package

Submodules

tensortrade.exchanges.simulated.fbm_exchange module

tensortrade.exchanges.simulated.gan_exchange module

tensortrade.exchanges.simulated.simulated_exchange module

Submodules

tensortrade.exchanges.exchange module

tensortrade.features package

Subpackages

tensortrade.features.indicators package

Submodules

tensortrade.features.indicators.simple_moving_average module

tensortrade.features.indicators.ta_indicator module

tensortrade.features.indicators.talib_indicator module

tensortrade.features.scalers package

Submodules

tensortrade.features.scalers.min_max_normalizer module

tensortrade.features.scalers.standard_normalizer module

tensortrade.features.stationarity package

Submodules

tensortrade.features.stationarity.fractional_difference module

Submodules

tensortrade.features.feature_pipeline module

tensortrade.features.feature_transformer module

tensortrade.rewards package

Submodules

tensortrade.rewards.reward_scheme module

tensortrade.rewards.risk_adjusted_returns module

tensortrade.rewards.simple_profit module

tensortrade.slippage package

Submodules

tensortrade.slippage.random_slippage_model module

tensortrade.slippage.slippage_model module

tensortrade.strategies package

Submodules

tensortrade.strategies.stable_baselines_strategy module

tensortrade.strategies.tensorforce_trading_strategy module

tensortrade.strategies.trading_strategy module

tensortrade.trades package

Submodules

tensortrade.trades.trade module

tensortrade.trades.trade_type module

Submodules

tensortrade.version module