ignite Documentation

Release master

Torch Contributors

Jan 18, 2021

NOTES

sharpe is a unified, interactive, general-purpose environment for backtesting or applying machine learning(supervised learning and reinforcement learning) in the context of quantitative trading.

it's designed to allow maximum flexibility and simplicity in the practical research process: raw financial data -> feature engineering -> model training -> trading strategy -> backtesting, come in the reasonable level of abstraction.

core features:

- unified: unify the rule-based and factor-based trading methodology, supervised learning and reinforcement learning in a framework.
- interactive: state(feature) -> action(portfolio-weight) -> reward(profit-and-loss/returns) bar-by-bar, allow maximum workflow control.
- general-purpose: market-independent, instrument-independent, trading-frequency-independent.

CHAPTER

ONE

OUTLINE

- *1 Motivation and concept design*
- *2 Additional Features*
- *3 Install*
- *4 Quick Start*
- *5 Documentation*

CHAPTER

TWO

1 MOTIVATION AND CONCEPT DESIGN

Before we walk through an end-to-end example how to backtest a trading stratey with **sharpe**, let's take a step back and discuss and understand the difficuties encountering when design a backtest engine for quantitative trading, the answer derives from quantitative researchers own different types of trading philosophy, trade different types of instruments in different markets with different trading frequencies.

- **different types of trading philosophy**: rule-based methodology versus factor-based methodology(supervised learning versus reinforcement learning)
- **different types of instruments in different market**: stock, index, ETF, future in different countries and markets.
- **different trading frequencies**: intra-day trading(seconds, minutes, hours) and inter-day trading(daily, weekly, montly)
- **different data structures and dtypes**: cross-sectional data is used for explaining the cross-sectional variation in stock returns, time series data is used for timing strategy development, sequential data is used for sequencial-model, e.g. RNN and it variation algorithm. Besides, supervised learning algorithm and reinforcement learning need different data architecture.

trading decison can be viewed as a special case of sequential decision-making, which can be formalized as follows: at each timestamp,

- a agent sees a observation of the state of the environment, that the agent lives in and interacts with
- and then decides on an action to take, based on a policy(can be also called strategy, a mapping from state to action)
- The agent perceives a reward signal from the environment, a number that tells it how good or bad the current action is
- see the observation of the next state, and iteratively

The goal of the agent is to find a good policy(strategy) to maximize its cumulative reward.

followwing this concept framwork, *sharpe* re-conceptualizes the process of trading and provides research with low-level, common tool to

CHAPTER

THREE

2 ADDITONAL FEATURES

- Support rule-based and factor-based trading strategy backtesting
- Helper functions for data/order management and rl algorithms.
- Various environment wrappers(e.g. data type wrapper, support pandas, numpy, pytorch tensor)
- Logging, visualization, and experiments management
- Unit tested, continuously integrated

CHAPTER

FOUR

3 INSTALL

```
$ git clone https://github.com/StateOfTheArt-quant/sharpe
$ cd sharpe
$ python setup.py install
```

CHAPTER

FIVE

4 QUICK START

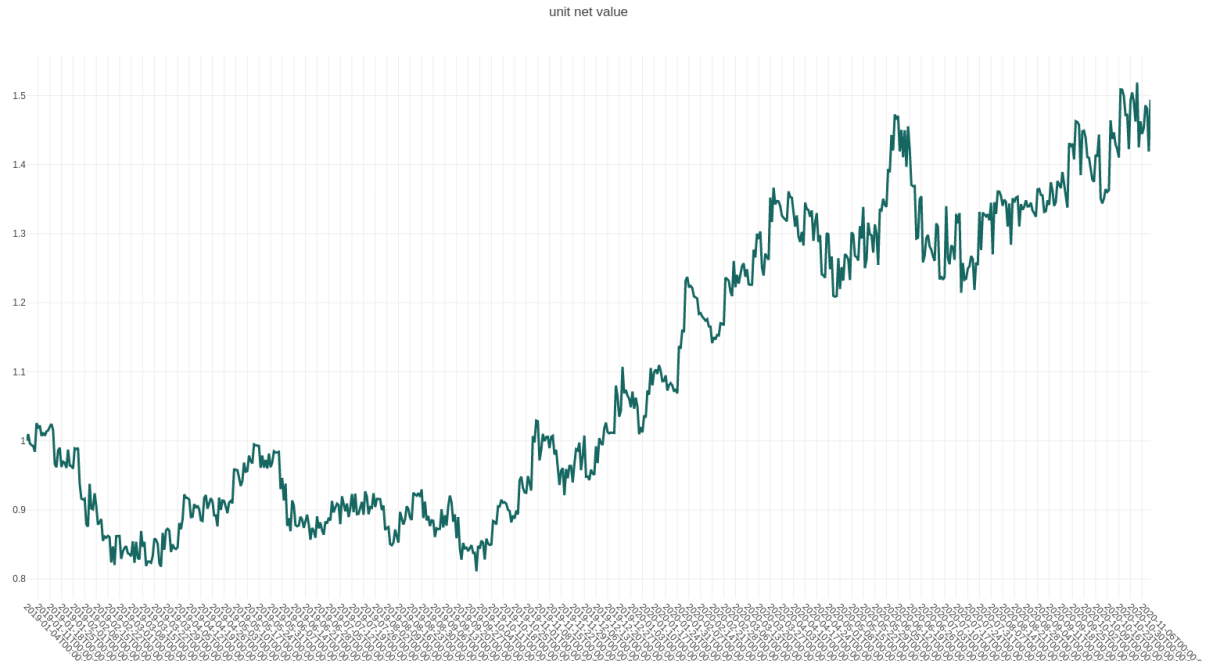
The following snippet showcases the whole workflow of trading strategy development in *sharpe*.

```
from sharpe.utils.mock_data import create_toy_feature
from sharpe.data.data_source import DataSource
from sharpe.environment import TradingEnv
from sharpe.mod.sys_account.api import order_target_weights
import random
random.seed(111)

feature_df, price_s = create_toy_feature(order_book_ids_number=2, feature_number=3,␣
↪start="2020-01-01", end="2020-01-11", random_seed=111)
data_source = DataSource(feature_df=feature_df, price_s=price_s)

env= TradingEnv(data_source=data_source, look_backward_window=4, mode="rl")
print('--------------------------------------------------------------------------')

company_id = "000001.XSHE"


def your_strategy(state):
     """
     here is a random strategy, only trade the first stock with a random target␣
↪percent
     """

     target_percent_of_position =  round(random.random(),2)
     target_position_dict = {company_id : target_percent_of_position}
     print("the target portfolio is to be: {}".format(target_position_dict))
     # call trade API
     action = order_target_weights(target_position_dict)
     return action


state = env.reset()

while True:
     print("the current trading_dt is: {}".format(env.trading_dt))
     action = your_strategy(state)

     next_state, reward, done, info = env.step(action)
     print("the reward of this action: {}".format(reward))
     print("the next state is \n {}".format(next_state))
     if done:
          break
```

(continues on next page)

(continued from previous page)

```
        else:
            state = next_state
env.render()
```



CHAPTER

SIX

5 DOCUMENTATION

To get started, please, read *Quick start* and *Concepts*.

6.1 Quick start

Welcome to **sharpe** quick start guide that just covers the essentials of getting a project up and walking through the code.

In several lines of this given code, you can backtest your first rule-based trading strategy as shown below:

6.1.1 Code

```
from sharpe.utils.mock_data import create_toy_feature
from sharpe.data.data_source import DataSource
from sharpe.environment import TradingEnv
from sharpe.mod.sys_account.api import order_target_weights
import random
random.seed(111)

feature_df, price_s = create_toy_feature(order_book_ids_number=2, feature_number=3,␣
↪start="2020-01-01", end="2020-01-11", random_seed=111)
data_source = DataSource(feature_df=feature_df, price_s=price_s)

env= TradingEnv(data_source=data_source, look_backward_window=4)
print('--------------------------------------------------------------------------')

company_id = "000001.XSHE"


def your_strategy(state):
    """
    here is a random strategy, only trade the first stock with a random target␣
↪percent
    """

    target_percent_of_position =  round(random.random(),2)
    target_position_dict = {company_id : target_percent_of_position}
    print("the target portfolio is to be: {}".format(target_position_dict))
    # call trade API
    action = order_target_weights(target_position_dict)
    return action
```

(continues on next page)

(continued from previous page)

```
 state = env.reset()

 while True:
      print("the current trading_dt is: {}".format(env.trading_dt))
      action = your_strategy(state)

      next_state, reward, done, info = env.step(action)
      print("the reward of this action: {}".format(reward))
      print("the next state is \n {}".format(next_state))
      if done:
           break
      else:
           state = next_state
env.render()
```

6.1.2 Explanation

Now let's break up the code and review it in details. the first step is to create your custom data source.

the input of DataSource involves in one pd.DataFrame(called feature_df) and one pd.Series(called price_s) with multiindex.

```
feature_df, price_s = create_toy_feature(order_book_ids_number=2, feature_number=3,␣
↪start="2020-01-01", end="2020-01-11", random_seed=111)
data_source = DataSource(feature_df=feature_df, price_s=price_s)
```

the feature_df is a multindex dataframe, the first index is instrument name, the second index is timestamp, the data containw your instruments' features(like the raw feature, open, high, low, close ,volume or features(factors) after feature engineering), that is the states of the market you care about.

```
                          feature_1  feature_2  feature_3
order_book_id datetime
000001.XSHE   2020-01-01  -1.133838   0.384319   1.496554
              2020-01-02  -0.355382  -0.787534  -0.459439
              2020-01-03  -0.059169  -0.354174  -0.735523
              2020-01-04  -1.183940   0.238894  -0.589920
              2020-01-05  -1.440585   0.773703  -1.027967
              2020-01-06  -0.090986   0.492003   0.424672
              2020-01-07   1.283049   0.315986  -0.408082
              2020-01-08  -0.067948  -0.952427  -0.110677
              2020-01-09   0.570594   0.915420  -1.669341
              2020-01-10   0.482714  -0.310473   2.394690
              2020-01-11   1.550931  -0.646465  -0.928937
000002.XSHE   2020-01-01  -1.654976   0.350193  -0.141757
              2020-01-02   0.521082  -0.020901  -1.743844
              2020-01-03  -0.799159  -1.303570   0.178105
              2020-01-04  -0.334402  -0.306027  -0.332406
              2020-01-05   1.962947   0.719242   1.142887
              2020-01-06   2.082877  -1.284648   0.538128
              2020-01-07  -0.044539   2.597164  -0.058266
              2020-01-08  -0.945287   0.541172  -0.055009
              2020-01-09   1.120021  -0.191643  -0.610138
```

(continues on next page)

(continued from previous page)

```
               2020-01-10  -0.444579  -2.204009  -0.430670
               2020-01-11  -0.425093   0.147292   0.424924
```

the price_s is a multiindex pd.Series, contraining the price of the instrument at different timestamp. this is an important component of the DataSource, which is used to backtest.

```
order_book_id  datetime
000001.XSHE    2020-01-01    42.31
               2020-01-02    43.61
               2020-01-03    40.40
               2020-01-04    43.11
               2020-01-05    46.29
               2020-01-06    43.01
               2020-01-07    38.62
               2020-01-08    46.05
               2020-01-09    45.38
               2020-01-10    39.80
               2020-01-11    42.19
000002.XSHE    2020-01-01    10.59
               2020-01-02    13.08
               2020-01-03    12.07
               2020-01-04    19.72
               2020-01-05    19.09
               2020-01-06    16.76
               2020-01-07    11.15
               2020-01-08    19.58
               2020-01-09    10.92
               2020-01-10    16.30
               2020-01-11    19.03
Name: price, dtype: float64
```

Next we define the environment. the input of TradingEnv is the data_source we have created and a int parameter, called *look_backward_window*, which tells the enviontment, at each timestamp. we can observe the past feature with window size==4, which consist the state of the environment at that timestamp. that means the shape of the state of environment is (instrument_numbers, look_backward_window, feature_numbers)

```
env= TradingEnv(data_source=data_source, look_backward_window=4)
```

The most interesting part of the code snippet is define your trading strategy which is a mapping from state to investment action(all instrument investment weight). it actually involve in two steps. the first step is to determine the investment percent(weight) of instruments based on the current state, which is your core logic of trading strategy. the output of this step is a dict {instrument_1_name: percent1, instrument_2_name: percent2},. the next step is call builtin trade API *order_target_portfolio* to create action(a order list)

```
def your_strategy(state):
    """
    here is a random strategy, only trade the first stock with a random target␣
↪percent
    """
    #step1
    target_percent_of_postion =  round(random.random(),2)
    target_pososition_dict = {company_id : target_percent_of_postion}
    print("the target portfolio is to be: {}".format(target_pososition_dict))
    #step2: call trade API
    action = order_target_portfolio(target_pososition_dict)
    return action
```

the last step is backtest your strategy, iterate all available timestamps

```
 state = env.reset() #the initial state of environment
 while True:
      print("the current trading_dt is: {}".format(env.trading_dt))
      action = your_strategy(state)

      next_state, reward, done, info = env.step(action)
      print("the reward of this action: {}".format(reward))
      print("the next state is \n {}".format(next_state))
      if done:
           break
      else:
           state = next_state
env.render()
```

6.2 Concepts

6.2.1 DataSource

6.3 Examples

- risk parity strategy

6.4 FAQ

In this section we grouped answers on frequently asked questions and some best practices of using *sharpe*.

6.5 About us

6.5.1 Authors

The following people are currently core contributors to sharpe's development and maintenance:

- Yu Jiang @walkacross

6.5.2 Join Core Team

We are looking for motivated contributors to become collaborators and help out with the project. We can start considering a candidate after several successfully merged Github pull requests. If you are interested, for more details, please, contact yu jiang (@walkacross) via email *yujiangallen* at *126.com*.

6.5.3 Citing sharpe

If you use sharpe in a scientific publication, we would appreciate citations to the project.

```
@misc{sharpe,
  author = {Yu Jiang},
  title = {Sharpe: a unified, interactive, general-purpose environment in the context␣
↪of quantitative trading},
  year = {2021},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {\url{https://github.com/StateOfTheArt-quant/sharpe}},
}
```

6.5.4 Acknowledgements

sharpe derived from our initial project trading_gym, which now is a event-driven(or observer) design pattern, the code highly inspired by RQALPHA

This library is named *sharpe* to respect William F. Sharpe