`fret:` Framework for Reproducible ExperimenTs

A Brief Introduction

Yu Yin

Outline

1. Background
2. Design of `fret`
3. Tutorial
    - 3.1 Basic Usage
    - 3.2 Using Workspace
    - 3.3 Working with Logs, Snapshots and Results
    - 3.4 Running Experiments
    - 3.5 Advanced Module Configuration
4. Examples
    - 4.1 Example Training Process in `PyTorch`
    - 4.2 Example Module Configuration in `TensorFlow`
5. Future Work

Background

Overview: the importance of ***reproducibility***

> *Were the algorithms and resources described to allow for* ***reproducibility****? This includes* ***experimental methodology****,* ***empirical evaluation****, dataset characteristics,* ***code/pseudo-code****, detailed proofs,* ***tuning parameter list and search space****, hardware/software utilized, other useful performance factors, etc.*
>
> *—KDD 2019 Review Criterion*

Background

Code complexity

To achieve reproducibility:

- Requires extra code
- Same sort of code for each new project
- Mess up with main logic. What you actually do becomes unclear

We need of a **general** experimental framework

Background

Challenges

1. Conducting experiments with different setups
2. Configuration management
3. Running process management

Background

Challenges

1. **Conducting experiments with different setups**

   Approaches:

Constants	(don’t do this . . .)
`argparse`	extra code, configuration not recorded
Scripts	too verbose, difficult to modify/extend
Config files	even more code, management issues

   What we want:

   - Running from command line
   - . . . with configuration properly recorded and well organized

Background

Challenges

2. **Configuration management**

Gin project from Google

With one more line (`@gin.configurable`), models will be able to:

- receive setup from configuration file
- configure with full featured referencing, scoping, and nesting

Limitations:

- Writing configuration files being **complicated**
- **Manual** tuning recording
- **Unfriendly** command line interface

Background

Challenges

3. **Running process management**

   Features often needed in reproducible experiments:

   - Save/load model snapshots
   - Stop and resume
   - Logging
   - Saving results
   - and more . . .

Introducing `fret`

Framework for Reproducible ExperimenTs

Features:

- **No** boilerplate code
- **Intuitive** CLI building
- **Easy** configuration and organization
- **Handy** running process utilities

Design

Overview

`fret` achieves its design goals through two main ideas:

- Define CLI command with functions
- The concept of workspace

Design

Define CLI command with functions

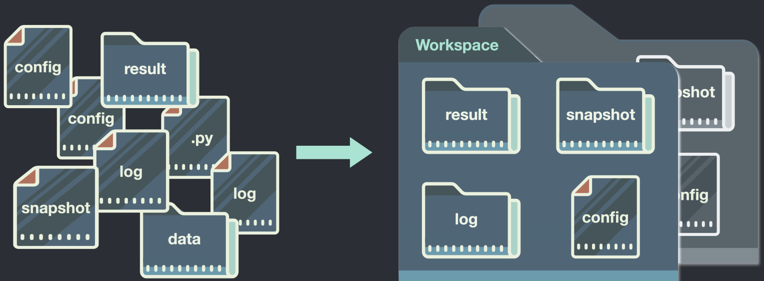
`fret` provides easy CLI building, inspired by Gin and Fire:

```
@fret.command
def run(foo='bar', num=3):
    print(run.config)
```

```
$ fret run -h
usage: fret run [-h] [-foo FOO] [-num NUM]
optional arguments:
  -h, --help         show this help message and exit
  -foo FOO, -f FOO   parameter foo (default: bar)
  -num NUM, -n NUM   parameter num (default: 3)
$ fret run -n 5
foo='bar', num=5
```

Design

The concept of workspace



Tutorial

Installation

Just:

```
$ pip install fret
```

Or visit `https://github.com/yxonic/fret` for latest version

Basic Usage

`@fret.command` and `@fret.configurable`

Everything can be put within one file:

```
# app.py
import fret

@fret.command
def run(ws):
    model = ws.build()
    print(’In [{}]: {}’.format(ws, model))

@fret.configurable
class Model:
    def __init__(self, x=3, y=4):
        ...
```

Basic Usage

`@fret.command` and `@fret.configurable`

Configure and run on command line:

```
$ fret config Model
[ws/_default] configured "main" as "Model" with: x=3, y=4

$ fret run
In [ws/_default]: Model(x=3, y=4)

$ fret config Model -x 5 -y 10
[ws/_default] configured "main" as "Model" with: x=5, y=10

$ fret run
In [ws/_default]: Model(x=5, y=10)
```

Using Workspace

Configuration management

Different configuration in different workspace:

```
$ fret -w ws/model1 config Model
[ws/model1] configured "main" as "Model" with: x=3, y=4

$ fret -w ws/model2 config Model -x 5 -y 10
[ws/model2] configured "main" as "Model" with: x=5, y=10

$ fret -w ws/model1 run
In [ws/model1]: Model(x=3, y=4)

$ fret -w ws/model2 run
In [ws/model2]: Model(x=5, y=10)
```

Working with Logs, Snapshots and Results

Paths, logger, and save/load utilities

Toy model: define states to be saved/loaded:

```
@fret.configurable(states=[’weight’])  # here
class Model:
    def __init__(self):
        self.weight = 0.

    def train(self):
        self.weight = random.random()

    def test(self):
        return self.weight ** 0.5
```

Working with Logs, Snapshots and Results

Paths, logger, and save/load utilities

Toy train/test example with model save/load:

```
@fret.command
def train(ws):
    logger = ws.logger(’train’)  # log to screen *and* train.log
    model = ws.build()

    model.train()
    logger.info(’trained with weight as %.3f’, model.weight)

    ws.save(model, ’trained’)  # save model with tag ‘trained‘
```

Working with Logs, Snapshots and Results

Paths, logger, and save/load utilities

Toy train/test example with model save/load:

```
@fret.command
def test(ws):
    logger = ws.logger(’test’)
    model = ws.load(’trained’)  # load trained model
    logger.info(’Loaded weight: %.3f’, model.weight)

    result = model.test()
    with ws.result(’test_result.txt’).open(’w’) as of:
        # save test result into a file
        print(result, file=of)
```

Working with Logs, Snapshots and Results

Paths, logger, and save/load utilities

Running:

```
$ fret config Model
[ws/_default] configured "main" as "Model"

$ fret train
INFO trained with weight as 0.605

$ fret test
INFO Loaded weight: 0.605

$ cat ws/_default/result/test_result.txt
0.7776197887329115
```

Running Experiments

ws.run() and fret.nonbreak()

```
@fret.command
def resumable(ws):
    with ws.run(’exp-1’) as run:
        sum = run.acc()
        for i in fret.nonbreak(run.range(1, 6)):
            time.sleep(0.5)
            sum += i
            print(’current i: %d, sum: %d’ % (i, sum))
```

Running Experiments

`ws.run()` and `fret.nonbreak()`

```
$ fret resumable
current i: 1, sum: 1
current i: 2, sum: 3
^CW SIGINT received. Delaying KeyboardInterrupt.
current i: 3, sum: 6
Traceback (most recent call last):
  ...
KeyboardInterrupt
W cancelled by user

$ fret resumable
current i: 4, sum: 10
current i: 5, sum: 15
```

Advanced Module Configuration

Submodules

```
@fret.configurable
class A:
    def __init__(self, foo=’bar’):
        ...

@fret.configurable(submodules=[’sub’])
class B:
    def __init__(self, sub, bar=3):
        self.sub = sub
```

Advanced Module Configuration

Submodules

```
$ fret config sub A
[ws/_default] configured "sub" as "A"
$ fret config B
[ws/_default] configured "main" as "B" with: sub=’sub’, bar=3
$ fret run
In [ws/_default]: B(bar=3, sub=A(foo=’bar’))
```

Advanced Module Configuration

Inheritance

```
@fret.configurable
class A:
    def __init__(self, foo=’bar’):
        ...

@fret.configurable
class B(A):
    def __init__(self, num=3, **others):
        super().__init__(**others)
        ...
```

Advanced Module Configuration

Inheritance

```
$ fret config B -foo baz -num 0
[ws/_default] configured "main" as "B" with: foo=’baz’, num=0
$ fret run
In [ws/_default]: B(num=0, foo=’baz’)
```

Examples

Example training process in PyTorch

Here demonstrates a training process that is:

1. resumable
2. with separate thread for data prefetching

```
with ws.run(’train’) as run:
    run.register(model)
    run.register(optimizer)
    for i in run.brange(n_epochs):
        train_iter = run.iter(
            ’train_iter’, train_data.data, train_data.targets,
            prefetch=True, batch_size=batch_size)
        for batch in fret.nonbreak(tqdm(train_iter,
                                            initial=train_iter.pos)):
            ...
```

Examples

Example module configuration in `TensorFlow`

Here shows sequence module configuration code in TF:

```
@fret.configurable
class RNN(tf.keras.Model):
    def __init__(self, batch_size, vocab_size,
                 emb_size=128, rnn_size=256):
        super().__init__()
        ...
```

Examples

Example module configuration in `TensorFlow`

```
@fret.configurable
class LSTM(RNN):
    def __init__(self, rnn_size=128, **others):
        super().__init__(rnn_size=rnn_size, **others)
        ...

@fret.configurable
class GRU(RNN):
    def __init__(self, **others):
        super().__init__(**others)
        ...
```

Future Work

- Result collection and table generation
- Configurable functions
- Better TensorFlow support

Reference

About `fret`:

- GitHub: `https://github.com/yxonic/fret`
- PyPI: `https://pypi.org/project/fret/`

Related projects:

- Gin configuration framework: `https://github.com/google/gin-config`
- A RL experimental framework: `https://github.com/google/dopamine/`
- Automatic CLI generation: `https://github.com/google/python-fire`
- Another CLI toolkit: `https://github.com/pallets/click/`

Q&A