confugue

Ondřej Cífka

Sep 13, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

Confugue is a hierarchical configuration framework for Python. It provides a wrapper class for nested configuration dictionaries (usually loaded from YAML files), which can be used to easily configure complicated object hierarchies.

The package is ideal for configuring deep learning experiments. These often have large numbers of hyperparameters, and managing all their values globally can quickly get tedious. Instead, Confugue allows each part of the deep learning model to be automatically supplied with hyperparameters from a configuration file, eliminating the need to pass them around. The structure of the configuration file follows the hierarchy of the model architecture; for example, if the model has multiple encoders consisting of several layers, then each layer will have its section in the configuration file, nested under the corresponding encoder section.

As an example, here is a simplified code snippet from a machine learning project which uses Confugue:

```
@configurable
class Model:

    def __init__(self, vocabulary, use_sampling=False):
        self.embeddings = self._cfg['embedding_layer'].configure(EmbeddingLayer,
                                                                  input_
↪size=len(vocabulary))
        self.decoder = self._cfg['decoder'].configure(RNNDecoder,
                                                      vocabulary=vocabulary,
                                                      embedding_layer=self.embeddings)

@configurable
class RNNDecoder:

    def __init__(self, vocabulary, embedding_layer):
        self.cell = self._cfg['cell'].configure(tf.keras.layers.GRUCell,
                                                units=100,
                                                dtype=tf.float32)
        self.output_projection = self._cfg['output_projection'].configure(
            tf.layers.Dense,
            units=len(vocabulary),
            use_bias=False)
```

The model could then be configured using the following config file, overriding the values specified in the code and filling in the ones that are missing.

```
embedding_layer:
  output_size: 300
decoder:
  cell:
    class: !!python/name:tensorflow.keras.layers.LSTMCell
```

(continues on next page)

(continued from previous page)

```
    units: 1024
use_sampling: True
```

CHAPTER

TWO

CONTENTS

2.1 Getting Started

2.1.1 Getting Started – General Guide

Tip: Deep learning users should check out the *Deep Learning Quick Start Guide* with examples in PyTorch.

First, we need a function or class to configure. Let's start with a simple function like this:

```
def main(foo, bar, baz):
    print(foo, bar, baz)
```

Next, we need to create a *Configuration* object. Typically, we will do this by loading a YAML config file:

```
from confugue import Configuration

config = Configuration.from_yaml_file('config.yaml')
```

`config` now acts as a wrapper for the contents of `config.yaml`, and can be used to configure our `main()` function, like so:

```
config.configure(main, foo=1, bar=2)  # baz needs to be set in config.yaml
```

The code above will call `main()` with the given arguments, plus any arguments defined in the configuration. The values specified in the code are treated as defaults and can be overridden by the configuration. For example, if `config.yaml` looks like this...

```
foo: ham
baz: spam
```

...then the above code will call `main(foo='ham', bar=2, baz='spam')`.

Hierarchical configuration

Although any function or class can be configured as described above, in order to make full use of Confugue, we need to decorate our functions and classes with the *@configurable* decorator. This enables them to access values from their parent *Configuration* object, and use them to further configure other functions or class instances. Decorated functions and classes each behave a bit differently:

- A *@configurable* class automatically obtains a magic `_cfg` property containing the parent configuration object. The property is set immediately upon the creation of the object, so that it can already be used in `__init__`.
- A *@configurable* function (or method) should define a keyword-only parameter `_cfg` (see below for an example of how to do that), which will receive the parent configuration object.

For example:

```
from confugue import configurable

@configurable
def main(foo, bar=456, *, _cfg):
    print('main', foo, bar)
    ham1 = _cfg['ham1'].configure(Ham)
    ham2 = _cfg['ham2'].configure(Ham)

@configurable
class Ham:

    def __init__(self, x):
        print('Ham', x)
        self._egg = self._cfg['egg'].configure(Egg, y=0)

class Egg:

    def __init__(self, y):
        print('Egg', y)

config = Configuration.from_yaml_file('config2.yaml')
config.configure(main)
```

Now, given the following `config2.yaml`...

```
foo: 123
ham1:
  x: 1
  egg:
    y: 2
ham2:
  x: 3
```

...we will get this output:

```
main 123 456
Ham 1
Egg 2
Ham 3
Egg 0
```

How does it work?

When we call `config.configure(main)`, the following happens:

- The `foo` value defined in the config file gets passed as an argument to `main()`. The values `ham1` and `ham2`, however, do not get passed as arguments since the function does not accept them, and instead become available via `_cfg`.
- `_cfg['ham1']` retrieves the `ham1` config dictionary and wraps it in a new *`Configuration`* object, ready to configure a new instance of `Ham`.
- Similarly, inside `Ham`'s constructor, the value under `ham1 -> egg` is retrieved and used to configure an `Egg` instance.

Notice how `self._cfg['egg'].configure(Egg, y=0)` works even though there is no `ham2 -> egg` key in the config file. This is because `self._cfg['egg']` returns an empty *`Configuration`* object, which will happily instantiate `Egg` as long as a default value for `y` is provided in the code.

Keep in mind

- When calling a configurable, Confugue looks at its function signature and matches the configuration keys against it. Only the matching keys are passed as arguments (unless the signature contains a `**kwargs` argument, in which case all keys will be used). This behavior can be changed by passing a list of configurable parameters as the `params` argument of the *`@configurable`* decorator.
- A configurable can still be called normally (rather than using *`configure`*). `_cfg` will be automatically set to a default configuration object, which will behave as if the configuration file was empty.
- The *`@configurable`* decorator is necessary only if the function or class needs to access its configuration (`_cfg`).
- Instead of loading a YAML file, one can use any other configuration dictionary by directly calling `Configuration(cfg_dict)`.

See also:

Advanced features are described in *More Features*.

2.1.2 Deep Learning Quick Start Guide

This section is intended as a quick start guide for deep learning users. It is based on PyTorch examples, but it should be easy to follow even for people working with other frameworks like TensorFlow.

Not into deep learning?

Confugue is absolutely not limited to machine learning applications. Python users unfamiliar with deep learning should check out the *General Guide*.

Tip: This guide is available as a Jupyter notebook.

Basic PyTorch example

We are going to start with a basic PyTorch model, adapted from the CIFAR-10 tutorial. First, let's see what the model looks like *without* using Confugue:

```
from torch import nn

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(400, 120)
        self.fc2 = nn.Linear(120, 10)
        self.act = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.act(self.conv1(x)))
        x = self.pool(self.act(self.conv2(x)))
        x = x.flatten(start_dim=1)
        x = self.act(self.fc1(x))
        x = self.fc2(x)
        return x
```

Making it configurable

Instead of hard-coding all the hyperparameters like above, we want to be able to specify them in a configuration file. To do so, we are going to decorate our class with the *`@configurable`* decorator. This provides it with a magic `_cfg` property, giving it access to the configuration. We can then rewrite our `__init__` as follows:

```
from confugue import configurable, Configuration

@configurable
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = self._cfg['conv1'].configure(nn.Conv2d, in_channels=3)
        self.conv2 = self._cfg['conv2'].configure(nn.Conv2d)
        self.pool = self._cfg['pool'].configure(nn.MaxPool2d)
        self.fc1 = self._cfg['fc1'].configure(nn.Linear)
        self.fc2 = self._cfg['fc2'].configure(nn.Linear, out_features=10)
        self.act = self._cfg['act'].configure(nn.ReLU)

    def forward(self, x):
        x = self.pool(self.act(self.conv1(x)))
        x = self.pool(self.act(self.conv2(x)))
        x = x.flatten(start_dim=1)
        x = self.act(self.fc1(x))
        x = self.fc2(x)
        return x
```

Instead of creating each layer directly, we configure it with values from the corresponding section of the configuration file (which we will see in a moment). Notice that we can still specify arguments in the code (e.g. `in_channels=3` for the `conv1` layer), but these are treated as defaults and can be overridden in the configuration file if needed.

Loading configuration from a YAML file

Calling `Net()` directly would result in an error, since we haven't specified defaults for all the required parameters of each layer. We therefore need to create a configuration file `config.yaml` to supply them:

```
conv1:
  out_channels: 6
  kernel_size: 5
conv2:
  in_channels: 6
  out_channels: 16
  kernel_size: 5
pool:
  kernel_size: 2
  stride: 2
fc1:
  in_features: 400
  out_features: 120
fc2:
  in_features: 120
```

Note: We do not need to include the activation function (`act`), since it does not have any required parameters. We could, however, *override the type* of the activation function itself.

We are now ready to load the file into a Configuration object and use it to configure our network:

```
>>> cfg = Configuration.from_yaml_file('config.yaml')
>>> cfg
Configuration({'conv1': {'out_channels': 6, 'kernel_size': 5}, 'conv2': {'in_channels
↪': 6, 'out_channels': 16, 'kernel_size': 5}, 'pool': {'kernel_size': 2, 'stride': 2}
↪, 'fc1': {'in_features': 400, 'out_features': 120}, 'fc2': {'in_features': 120}})
>>> cfg.configure(Net)
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=10, bias=True)
  (act): ReLU()
)
```

Tip: Instead of loading a YAML file, one can use any configuration dictionary by directly calling `Configuration(cfg_dict)`.

Nested configurables

One of the most useful features of Confugue is that *@configurable* classes and functions can use other configurables, and the structure of the configuration file will naturally follow this hierarchy. To see this in action, we are going to write a configurable `main` function which trains our simple model on the CIFAR-10 dataset.

```
import torchvision
from torchvision import transforms

@configurable
def main(num_epochs=1, log_period=2000, *, _cfg):
    net = _cfg['net'].configure(Net)
    criterion = _cfg['loss'].configure(nn.CrossEntropyLoss)
    optimizer = _cfg['optimizer'].configure(torch.optim.SGD, params=net.parameters(),
                                            lr=0.001)

    transform = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                              download=True, transform=transform)
    train_loader = _cfg['data_loader'].configure(torch.utils.data.DataLoader,
                                                 dataset=train_data, batch_size=4,
                                                 shuffle=True, num_workers=2)

    for epoch in range(num_epochs):
        for i, batch in enumerate(train_loader):
            inputs, labels = batch
            optimizer.zero_grad()
            loss = criterion(net(inputs), labels)
            loss.backward()
            optimizer.step()

            if (i + 1) % log_period == 0:
                print(i + 1, loss.item())
```

Our `config.yaml` might then look like this:

```
net:
  conv1:
    out_channels: 6
    kernel_size: 5
  conv2:
    in_channels: 6
    out_channels: 16
    kernel_size: 5
  pool:
    kernel_size: 2
    stride: 2
  fc1:
    in_features: 400
    out_features: 120
  fc2:
    in_features: 120

optimizer:
  class: !!python/name:torch.optim.Adam
data_loader:
```

(continues on next page)

(continued from previous page)

```
  batch_size: 8
num_epochs: 2
log_period: 1000
```

To create and train our model:

```
cfg = Configuration.from_yaml_file('config.yaml')
cfg.configure(main)
```

Configuring lists

The *configure_list* method allows us to configure a list of objects, with the parameters for each supplied from the configuration file. We are going to use this, in conjunction with `nn.Sequential`, to fully specify the model in the configuration file, so we won't need our `Net` class anymore.

```
layers:
  - class: !!python/name:torch.nn.Conv2d
    in_channels: 3
    out_channels: 6
    kernel_size: 5
  - class: !!python/name:torch.nn.ReLU
  - class: !!python/name:torch.nn.MaxPool2d
    kernel_size: 2
    stride: 2
  - class: !!python/name:torch.nn.Conv2d
    in_channels: 6
    out_channels: 16
    kernel_size: 5
  - class: !!python/name:torch.nn.ReLU
  - class: !!python/name:torch.nn.MaxPool2d
    kernel_size: 2
    stride: 2
  - class: !!python/name:torch.nn.Flatten
  - class: !!python/name:torch.nn.Linear
    in_features: 400
    out_features: 120
  - class: !!python/name:torch.nn.ReLU
  - class: !!python/name:torch.nn.Linear
    in_features: 120
    out_features: 10
```

Creating the model then becomes a matter of two lines of code:

```
>>> cfg = Configuration.from_yaml_file('config.yaml')
>>> nn.Sequential(*cfg['layers'].configure_list())
Sequential(
  (0): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (4): ReLU()
  (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (6): Flatten()
  (7): Linear(in_features=400, out_features=120, bias=True)
  (8): ReLU()
```

(continues on next page)

(continued from previous page)

```
  (9): Linear(in_features=120, out_features=10, bias=True)
)
```

This offers a lot of flexibility, but it should be used with care. If your configuration file is longer than your code, you might be overusing it.

See also:

Advanced features are described in *More Features*.

Two guides for getting started with Confugue are available:

- *General guide*: A guide to the basics of Confugue for general Python users.
- *Deep learning guide*: A quick start guide for deep learning users with extensive examples in PyTorch.

2.2 More Features

2.2.1 Overriding the callable

In addition to overriding the arguments of a callable (function or class), the configuaration file may also replace the callable itself using the `class` key. In a YAML file, the value needs to be specified using the `!!python/name` tag (see the PyYAML documentation for more information):

```
ham:
  class: !!python/name:spam.Spam
```

Note that this is potentially unsafe, as it allows the configuration file to execute arbitrary code. To load YAML files safely (which will disable this feature), pass `loader=yaml.SafeLoader` to *`from_yaml()`* or *`from_yaml_file()`*.

2.2.2 Accessing raw values

The raw content of a configuration object can be obtained by calling its *`get()`* method. To access the value of a given key, use `_cfg.get('key')` (equivalent to `_cfg['key'].get()`).

2.2.3 Configuration lists

Sometimes we might want to create a list of objects of the same type, with arguments for each item supplied in the configuration file. This can be useful for example when creating a deep neural network with layers of different sizes. In this situation, we can use the *`configure_list()`* method, like so:

```
_cfg['dense_layers'].configure_list(tf.keras.layers.Dense, activation='relu')
```

The configuration file might then look like this:

```
dense_layers:
  - units: 100
  - units: 150
  - units: 2
    activation: None
```

2.2.4 Argument binding

It is sometimes useful to pre-configure a callable without actually calling it, for example if we intend to call it multiple times or want to supply additional arguments later. This can be achieved using the *bind()* method, e.g.:

```
Dense = _cfg['dense_layer'].bind(tf.keras.layers.Dense, activation='relu')
dense1 = Dense()  # parameters supplied by configuration
dense2 = Dense(use_bias=False, activation=None)  # overrides configuration
```

2.2.5 Maybe configure

We have seen that we can omit parts of the configuration file as long as defaults for all the required parameters are defined in the code. However, we might sometimes want to skip creating an object if the corresponding key is omitted from the configuration. This functionality is provided by the *maybe_configure()* method, which returns `None` if the configuration value is missing.

There is also *maybe_bind()*, which works analogously (see *Argument binding* above).

2.2.6 Required parameters

Instead of providing a default value, it is possbile to explicitly mark a parameter as required:

```
_cfg['dense_layer'].configure(tf.keras.layers.Dense, activation=_cfg.REQUIRED)
```

Not providing a value for this parameter in the configuration will result in an exception.

2.3 API Reference

`@confugue.`**`configurable`**(*, *params=ALL*, *cfg_property='_cfg'*, *cfg_param='_cfg'*)

A decorator that makes a function or a class configurable.

The decorator may be used with or without parentheses (i.e. both `@configurable` and `@configurable()` is valid).

If the decorated callable is a function or method, it needs to define a keyword-only argument `_cfg`, which will be automatically filled with an instance of *Configuration* when the function is called. If the decorated callable is a class, a `_cfg` property will be created holding the *Configuration* instance.

The decorated function/class can be called/instantiated normally (without passing the *_cfg* argument), or via *Configuration.configure()*.

Parameters

- **`params`** – A list of configuration keys to pass as keyword arguments. The default behavior is to include all keys matching the function's signature, or all keys if the signature contains a ** parameter.
- **`cfg_property`** – The name of the property that will hold the *Configuration* object in the case where a class is beging decorated.
- **`cfg_param`** – The name of the parameter that will receive the *Configuration* object in the case where a function is being decorated. This needs to be a keyword-only parameter.

class confugue.**Configuration**(*value: Any = MISSING_VALUE*, *name: str = '<root>'*)

Wrapper for nested configuration dictionaries or lists.

The core functionality is provided by the *configure()* method, which calls a given callable with the arguments from the wrapped dictionary.

If the wrapped value is a dictionary or a list, basic operations such as indexing and iteration are supported, with the values recursively wrapped in *Configuration* objects. If the user tries to access a key or index which is missing, an "empty" configuration object is returned; this can still be used normally and behaves more or less as if it contained an empty dictionary.

The wrapped value may be of any other type, but in this case, most of the methods will raise an exception. To retrieve the raw wrapped value (whatever the type), use the *get()* method with no arguments.

configure(*constructor: Optional[Callable] = None*, */*, ***kwargs*) → Any

Configure a callable using this configuration.

Calls *constructor* with the keyword arguments specified in this configuration object or passed to this method. Note that the constructor is called even if this configuration object corresponds to a missing key. *constructor* may be overridden in by a *class* configuration key (if the *constructor* parameter is not given, then the *class* key is required).

Any keyword arguments passed to this method are treated as defaults and can be overridden by the configuration. A special `Configuration.REQUIRED` value can be used to mark a given key as required.

Returns The return value of *constructor*, or *None* if the wrapped value is *None*.

Raises ***ConfigurationError*** – If the wrapped value is not a dict, if required arguments are missing, or if any exception occurs while calling *constructor*.

maybe_configure(*constructor: Optional[Callable] = None*, */*, ***kwargs*) → Any

Configure a callable only if the configuration is present.

Like *configure()*, but returns *None* if the configuration is missing.

Returns The return value of *constructor*, or *None* if the wrapped value is missing or *None*.

Raises ***ConfigurationError*** – If the wrapped value is not a dict, if required arguments are missing, or if any exception occurs while calling *constructor*.

configure_list(*constructor: Optional[Callable] = None*, */*, ***kwargs*) → Optional[List]

Configure a list of objects.

This method should be used if the configuration is expected to be a list. Every item of this list will then be used to configure a new object, as if *configure()* was called on it. Any defaults supplied to this method will be used for all the items.

Returns A list containing the values obtained by configuring *constructor*, in turn, using all the dicts in the wrapped list; *None* if the wrapped value is *None*.

Raises ***ConfigurationError*** – If the wrapped value is not a list of dicts, if required arguments are missing, or if any exception occurs while calling *constructor*.

bind(*constructor: Optional[Callable] = None*, */*, ***kwargs*) → Optional[Callable]

Configure a callable without calling it.

Like *configure()*, but instead of calling *constructor* directly, it returns a new function that calls *constructor* with parameters bound to the supplied values. The function may still accept other parameters.

Returns A function, or *None* if the wrapped value is *None*.

Raises ***ConfigurationError*** – If the wrapped value is not a dict, or if required arguments are missing.

maybe_bind(*constructor: Optional[Callable] = None*, */*, ***kwargs*) → Optional[Callable]
Configure a callable without calling it, but only if the configuration is present.

Like *bind()*, but returns *None* if the configuration is missing.

Returns A function, or *None* if the wrapped value is missing or *None*.

Raises ***ConfigurationError*** – If the wrapped value is not a dict, or if required arguments are missing.

get(*key: Hashable = None*, *default: Any = NO_DEFAULT*) → Any
Return an item from this configuration object (assuming the wrapped value is indexable).

Returns If *key* is given, the corresponding item from the wrapped object. Otherwise, the entire wrapped value. If the value is missing, *default* is returned instead (if given).

Raises

- **KeyError** – If the value is missing and no default was given.
- **IndexError** – If the value is missing and no default was given.
- **TypeError** – If the wrapped object does not support indexing.

get_unused_keys(*warn: bool = False*) → List[Hashable]
Recursively find keys that were never accessed.

Parameters **warn** – If *True*, a warning will be issued if unused keys are found.

Returns A list of unused keys.

classmethod from_yaml(*stream: str | bytes | TextIO | BinaryIO*, *loader=yaml.Loader*) → Configuration
Create a configuration from YAML.

The configuration is loaded using PyYAML's (potentially unsafe) `Loader` by default. If you wish to load configuration files from untrusted sources, you should pass `loader=yaml.SafeLoader`.

Parameters

- **stream** – A YAML string or an open file object.
- **loader** – One of PyYAML's loader classes.

Returns A *Configuration* instance wrapping the loaded configuration.

classmethod from_yaml_file(*stream: str | TextIO | BinaryIO*, *loader=yaml.Loader*) → Configuration
Create a configuration from a YAML file.

The configuration is loaded using PyYAML's (potentially unsafe) `Loader` by default. If you wish to load configuration files from untrusted sources, you should pass `loader=yaml.SafeLoader`.

Parameters

- **stream** – A path to a YAML file, or an open file object.
- **loader** – One of PyYAML's loader classes.

Returns A *Configuration* instance wrapping the loaded configuration.

class confugue.**interactive**(*mode: str = 'all'*)
A context manager that enables or disables the interactive editing mode.

Parameters **mode** – *'all'* to edit all values, *'missing'* to edit only missing values, or *'none'* to disable the interactive mode.

class confugue.**ConfigurationError**

class confugue.**ConfigurationWarning**

2.4 Comparison to Other Frameworks

2.4.1 Gin

Confugue is somewhat similar to Gin, but is much more minimalistic yet, in some ways, more powerful. Some advantages of Confugue over Gin are:

- It is straightforward to configure many objects of the same type with different parameters for each; with Gin, this is possible, but it requires using scopes.
- Any function or class can be configured without having been explicitly registered.
- Config files may override the type of an object (or the function being called) while preserving the default parameters provided by the caller.
- It is possible to access (and even manipulate) configuration values explicitly instead of (or in addition to) having them supplied as parameters.
- The structure of the config file is nested – typically following the call hierarchy – compared to Gin's linear structure.

On the other hand, Confugue doesn't have some of the advanced features of Gin, such as config file inclusion or 'operative configuration' logging. It also doesn't support macros, but a similar effect can be achieved using PyYAML's aliases.

Some other differences (which may be viewed as advantages or disadvantages in different situations) are:

- Gin config files specify *default* values for function parameters, which can be overridden by the caller. In Confugue, on the other hand, the config file has the final say.
- Gin will seamlessly load defaults from the configuration file every time a configurable function or class is called. Confugue is more explicit in that the caller first has to ask for a specific key from the configuration file.

2.4.2 Sacred

Sacred also offers configuration functionality, but its goals are much broader, focusing on experiment management (including keeping track of metrics and other information). Confugue, on the other hand, is not specifically targeted to scientific experimentation (even though it is particularly well suited for machine learning experiments). As for the configuration mechanism itself, Sacred has so-called 'captured functions' which resemble configurable functions in Confugue or Gin, but does not offer the same ability to configure arbitrary objects in a hierarchical way.

INDEX