NAAPC

I

Justification

Purpose

This package serves two purposes: 1) A nested dictionary class (ndict) which supports easy-to-use and efficient nested dictionary manipulation; 2) A configuration class (nconf) based on the nested dictionary class which supports fundamental functionalities of configuration and argument parsing. This software is initially developed for deep learning experiments and is found to be suitable for many other applications. This package aims to provide the minimum functionality that one may need for any configurations and is supposed to be seamlessly integrated with other tools. To achieve this, I use the most common packages. Many ideas are inspired from GitHub - facebookresearch/hydra: Hydra is a framework for elegantly configuring complex applications. Users are recommended to try it.

Proposed Features

For the ndict (Nested dictionary) class, I propose the following features:

1. The users should use the ndict class as:

```
d = ndict()
d["path;leaf"] = 1 # create a new node if not exist.
print(d["path;leaf"]) # Print 1.
del d["path;leaf"]
for k in d.keys(depth=1): # iterate till depth = 1.
        ...
for v in d.values(depth=2): # iterate till depth = 2.
        ...
for k, v in d.items(): # iterate all depthes
        ...
d.update(d1, ignore_none=True, ignore_missing=True) # d1 can be a ndict or normal dict, or a flatten dictionary (see below).
d.get("path;leaf", default=None)
len(d) # Should return the number of leaf nodes. Note that None leaves also count.
size(d, ignore_none=True) # Option to ignore None leaves.
d == d1 # equal if they are exactly the same, including the None leaf.
d.eq(d1, ignore_none=True, compare_paths=None, ignore_missing=False) # optional to ignore None leaf. Options to specify what subtree should be c
```

```
ompared. Option to compare only the intersection paths.
"path;leaf" in d
if d:   # True if len(d) > 0
        ...
print(d) # print in yaml format (This is because yaml format is more com
pact).
```

Some options are also provided: ignore the None leaves, only compare specified paths, only compare intersected paths, ignore the missed path during updating. Additionally, since the nested dictionary is a tree structure, we also provide options to specify the maximum depth that the iteration can reach. By default, this value is -1 which means till the deepest leaves.

2. The ndict class should be able to be serialized into a normal dictionary as {path:value} and can be deserialized back into a ndict class.

```
a = d.flatten_dict # Get the serialized dictionary. Note that this is co
nsuming since the whole tree needs to be traversed.
d = ndict(flatten_dict=a) # Generate the ndict from the flatten dictiona
ry.
```

3. Other then the eq() and `__equal__` methods, we provide another method to compare two dictionary:

```
d.diff(d1, ignore_none=True, ignore_missing=False)
```

This method will give the difference in a dictionary format.

4. Provide a match() method to determine if the nested dictionary is the wanted one. Users may use path-value pairs or path-function pairs.

5. Provide a transform() method to transform the current ndict to another based on the input path-value pairs and path-function pairs.

```
{
        "new_path;leaf": "old_path;node1;path",
        "new_path;aggregate": ""
}
```

Note that the arguments of the path-function pairs are d (for the current ndict object).

For the nconfig (nested configuration class which is based on ndict class), I propose the following features:

1. Type restriction. Configurations should only be basic types: int, float, bool and str.
2. Composition. User may compose the final configuration through other configurations / configuration paths. The syntax is the same to hydra as follows:

```
{
        "__configs__": [
                ["__config_path__: exps.olrandom"],
                {"data": "__config_path__: imagenet-100"},
                "__curr__"
        ]
}
```

The list-element configurations will be applied globally while the dict-element configurations will be applied to the provided path. The later configurations will overwrite the earlier ones. Note that the `__configs__` keyword must appear at the top of the configuration. Similar to Hydra, we also use a `__curr__` to represent the priority of the configurations of the current file. If not specified, the "__curr__" will be at the bottom of the list. To generate a configuration from multiple files, naapc will generate the configuartions according to the order in the list one by one and the later configurations will overwrite the earlier. Aditionally, users may use `__config_path__` (path relative to the parent of the root configuration file) and `__global_path__` (absolute path or path relative to the current working directory, i.e., the normal path in python scripts.) to get the configs under each configuration entries:

```
{
        "dataset": "__config_path__: datasets.imagenet",
        "model": "__global_path__: configs.resnet.resnet52",
        "__NAAPC_CONFIGS__": {"path_delimiter": "."}
}
```

Note that users may configure how to seperate paths in the configuration. For multi-file configuraiton, the latest will take effect. Therefore, it is recommended to use the consistent seperator. In the future, the priority of the NAAPC configurations will also be considered.

Note that the ".json" suffix is omited which is also the Hydra conventions.

3. Parsing input argument according to the configurations. Users may specify the types of the configuration. Null configurations will be omitted unless specified in purpose. Configurations are as follows:

```
{
        "__ARG_SPECS__": [
                {"node1;node2": {"type": "float", "help": "The specifications can be arguments of the add_argument method."}}
        ]
}
```

Limitations

The major limitations are as follows:

1. Only support str type for key due to the needs to serialize the nested dictionary.

Detailed Design

ndict

The ndict class maintain a built-in dictionary. Most functionalities of ndict depends on two methods:

1. traverse() which traversese the nested dictionary and apply certain actions and return the results:

```
def dfs(
        curr: dict,
        path: str,
        depth: int,
        res: Any,
        actions: tuple[callable, dict[str, callable]],
        stop_condition: callable,
        includes: Optional[list[str]] = None,
        excludes: Optional[list[str]] = None
) -> Any:
        depth++
        Apply actions on current node.

```

```
        if stop_condition(...): return res
        else: iterate to next depth or return res

def traverse(
        self,
        res: Any,
        actions: Union[callable, tuple[callable, dict[str, callable]]],
        depth: Optional[int] = -1,
        stop_condition: callable,
        alg: callable = dfs,
        includes: Optional[list[str]] = None,
        excludes: Optional[list[str]] = None
) -> Any:
        # Actions should be a tuple of (callable, {path:callable}). The first is the default action. Users may also only
        # input the default action. If users don't have default actions, they are advised to use __getitem__ or __setitem__ istead.
        # For all callables, they must accept 4 arguments: node: Any, path: str, depth: int, node: Any.
        def _depth_stop_condition(...):
                # The default stop_condition

        Generate stop condition function.
        Generate action function.

        return alg(...)
```

Note that one may use the traverse method to iteratively change the values of the nested dictionary. However, the users should be careful of deleting/add elements. Users may use `includes` and `excludes` to specify the subtrees or exclude subtrees.

2. _travel() which reduce the path and reach the final destination:

```
def _travel(self, path: list[str]) -> Any:
        return reduce(getitem, path, self._d)
```

Note that since the `__getitem__` and `__setitem__` methods mainly use the _travel method, users are recommended to use them as instead.

We now focus on how to implement each features of the ndict class.

For initialization, we should allow the class to be initialized as an empty dictionary:

```
def __init__(self, d: Optional[dict] = None, delimiter: str = ";"):
        If None: d = {}.
        Assign attributes.
```

Flatten the Tree Structure

Decisions

1. The two class names are chosen to be lowercase in order to make it easier to use.
2. JSON is faster and it does not support object serialization. Therefore, it is more suitable for naapc.
3. The built-in ArgumentParser is used for maximum combatibility.
4. Currently, only dfs is implemented. In the future, the users may choose different traverse algorithms or implement their own traverse algorithm.
5. If the return type is dict, ndict won't automatically convert it into ndict. This feature may be implemented in the future with an option.
6. The visualization should be in yaml format since yaml is more compact.
7. The ndict should be able to be serialized and deserialized since the deserialized format is useful for some situations, e.g. specify the a few nodes and their values.
8. We don't provide a paths property, since the users may use keys() to get all paths.

Future Plan

1. Use C++ to implement the traverse and travle methods and measure the performance boost.
2. The priority of the NAAPC configurations.
3. Yield generator.s
4. If the returned type is dictionary, we may provide an option to automatically convert it into ndict.
5. Users may change `__str__` to be in json format.