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).
# keys are path-path pairs or path-callable pairs which accept d, p as arguments. This method can be used to retrive with default values or aggregate a new dictionary.
d.get("path;leaf", keys: Optional[Union[dict, ndict]] = None, default: Optional[Any] = None)
len(d) # Should return the number of leaf nodes. Note that None leaves also count.

```

```
d.size(ignore_none=True) # Option to ignore None leaves.
d == d1 # equal if they are exactly the same, including the None leaf.
# optional to ignore None leaf. Options to specify what subtree and the
comparing methods (callables) should be compared. Option to ignore missi
ng path in any parties.
d.eq(d1, keys: Optional[Union[dict, ndict]], includes: Optional[list[st
r]] = None, excludes: Optional[list[str]] = None, ignore_none: bool = Tr
ue, ignore_missing=False)
"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(a) # Generate the ndict from the flatten dictionary. Note that
the __getitem__ and __setitem__ method make it possible to initialize nd
ict from flatten dict automatically
```

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

```
d.diff(d1, keys: Optional[Union[dict, ndict]], includes: Optional[list[s
tr]] = None, excludes: Optional[list[str]] = None, ignore_none: bool = T
rue, missing_method: str = "keep")
```

This method will give the difference in a dictionary format.

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 additional features:

1. Type restriction. Configurations should only be basic types: int, float, bool and str. Users may choose to enable this feature or not.
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. 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__": {"path_delimiter": "."}
}
```

Note that users may configure how to seperate paths in the `__naapc__` 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 CLI 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": {"flags": "node2", "type": "float", "help": "The specifications can be arguments of the add_argument method."},
                "__excludes__": ["node1;node3"]
        },
        "__naapc__": {"arg_delimiter": "."}
}
```

Limitations

The major limitations are as follows:

1. Only support str type for key due to the needs to serialize the nested dictionary.
2. **Users should not modify the underline dictionary outsied of ndict class, since this will corrupt the flatten_dict class.**

Detailed Design

ndict

The ndict class maintain a built-in dictionary and a flatten dictionary. This will increase the memory usage but improve the time efficiency. To maintain the flatten_dict, every update of the underling dictionary should be notified. Most functionalities of ndict depends on two methods:

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

```
def depth_stop_condition(
        curr: Any,
        parent_path: str,
        depth: int,
        res: Any,
```

```
):
        ...

def dfs(
    res: Any, node: Any, path: str, depth: int, action: callable, stop_condit
ion: callable
) -> Any:
    """Depth-first traverse.

    Args:
        res (Any): Current results.
        node (Any): Current node (value).
        path (str): Current path (key)
        depth (int): Current depth (0 at root)
        action (callable): Action.
        stop_condition (callable): Whether should stop the traverse process.

    Returns:
        Any: Results
    """
        Apply actions on current node.
        Update self._flatten_dict if the return from actions is true.
        if stop_condition(...): return res
        elif type is dict:
                add current node to parent path.
                iterate to next depth
        else: return res

def traverse(
        self,
        res: Any,
        actions: Union[callable, tuple[callable, dict[str, callable]]],
        depth: Optional[int] = -1,
        stop_condition: Optional[Union[list[callable], callable]] = None,
        alg: callable = dfs,
) -> Any:
        """Go through nodes in the tree and apply actions / collect data.

        Args:
            res (Any): Where the final results will be.
            actions (Union[callable, tuple[callable, dict[str, callable]]]):
action applied on all nodes or A tuple of
                (default action, {path: path actions}). Each callable should
accept: res, node, path, depth 4 arguments.
            depth (Optional[int], optional): Maximum traverse depth. Defaults
to -1, which means traverse all depth.
            stop_condition (Optional[Union[list[callable], callable]], option
al): Callable which returns bool to
```

```
                determine whether the tranverse should be stopped. This calla
ble should accept res, node, path, depth 4
                arguments. It can also be a list of callables. In that case a
ll the callable must return True to stop
                the traverse process. Defaults to None which means no extra c
onditions other depth.
            alg (callable, optional): Algorithm used to traverse the tree. Cu
rrently only support dfs. It should accept
                res, node, path, depth, actions and stop_condition 6 argument
s. Defaults to dfs.

        Returns:
            Any: The results
        """

        Generate stop condition function.
        Generate action function.

        return alg(...)

    def _generate_depth_stop_condition(self, max_depth: int = -1) -> callabl
e:
        def depth_stop_condition(res: Any, node: Any, path: str, depth) -> bo
ol:
            return max_depth != -1 and depth > max_depth

        return depth_stop_condition

    def _generate_stop_condition_pipeline(
        self, conditions: Optional[Union[list[callable], callable]], max_dept
h: int = -1
    ) -> callable:
        conditions = [conditions] if isinstance(conditions, callable) else co
nditions
        conditions = [] if conditions is None else conditions
        conditions.append(self._generate_depth_stop_condition(max_depth))

        def stop_condition_pipeline(res: Any, node: Any, path: str, depth: in
t) -> bool:
            for cond in conditions:
                if cond(res, node, path, depth):
                    return True
            return False

        return stop_condition_pipeline

    def _generate_action_pipeline(
        self, actions: Union[callable, tuple[callable, dict[str, callable]]]
```

```
    ) -> callable:
        if isinstance(actions, tuple):
            default_action = actions[0]
            if len(actions) == 2:
                specific_actions = actions[1]
        else:
            default_action = actions
            specific_actions = {}

        def action_pipeline(res: Any, node: Any, path: str, depth: int) -> No
ne:
            if path in specific_actions:
                specific_actions[path](res, node, path, depth)
            else:
                default_action(res, node, path, depth)

        return action_pipeline
```

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. **Note that deleting nodes by the traverse method is prohibited.**

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

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

This method should not be called publicly due to the risk of modify the underling dictionary from outside.

Additionally, many methods accept path-value or path-callable pairs. We call these pairs "query". Many also accept a missing_method argument, which can be "ignore", "exception", "keep".

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 and is able to provide the basic functionalitiesz:

```
def __init__(self, d: Optional[Union[dict, ndict]] = None, delimiter: str =
";"):
        If None: d = {}.
        if is ndict, dictionary.
```

```
        Assign attributes.
        Generate & assign flatten_dict.
```

To get the flatten_dict from the input dictionary, we provide the flatten() method using the traverse method:

```
def flatten(self, d: dict) -> dict[str, Any]:
        def _flatten_action(curr: Any, parent_path: str, depth: int, res: dict) -> dict:
                if curr is not dict:
                        flatten_dict[path] = curr
                return {}
        res = {}
        self.traverse(res, _flatten_action)
```

For the dict-like user experience, The magic methods are of vital importance. `__setitem__`, `__getitem__` and `__delitem__` are implemented based on _travel() method. `__setitem__` method only needs to ignore the last node. Note that **all setting items are achieved by __setitem__ method.** Additionally, `__setitem__` and `__delitem__` method should also update the flatten_dict:

```
def __getitem__(self, path: Union[str, list[str]]) -> Any:
        # Call _travel()

def __setitem__(self, path: Union[str, list[str]], value: Any) -> None:
        # Set value.
        # Update flatten_dict.

def __delitem__(self, path: Union[str, list[str]]) -> None:
        # Recursively delete the nodes.
        # Delete the paths from the flatten_dict.
```

For iterating, they are implemented using the traverse method as follows:

```
def keys(self, depth: int = -1) -> list[str]:
        max_depth = depth
        def _key_action(curr: Any, parent_path: str, depth: int, res: list[str]) -> dict:
                if depth == max_depth or curr is not dict:
                        res.append(path)
```

```
                return {}
        res = []
        self.traverse(res, _key_action, depth=max_depth)
        return res

def values(self, depth: int = -1) -> list:
        max_depth = depth
        def _value_action(curr: Any, parent_path: str, depth: int, res: list
[str]) -> dict:
                if depth == max_depth or curr is not dict:
                        res.append(curr)
                return {}
        res = []
        self.traverse(res, _value_action, depth=max_depth)
        return res

# It's not recommended to directly change the item returned by this method.
def items(self, depth: int = -1) -> tuple[list[str], list[Any]]:
        max_depth = depth
        def _items_action(curr: Any, parent_path: str, depth: int, res: tuple
[list[str], list[Any]]) -> dict:
                if depth == max_depth or curr is not dict:
                        res[0].append(path)
                        res[1].append(curr)
                return {}
        res = ([], [])
        self.traverse(res, _value_action, depth=max_depth)
        res = zip(*res)
        return res
```

To update from another dict, we need the update method:

```
def update(self, d: Union[ndict, dict] ignore_none: bool = True, missing_meth
od: str = "ignore") -> None:
        # d can be a normal path-value pair or a path-callable pair. Callable
s should accept: d (the target) and path (the target path)
        d = ndict(d)
        iterate flatten dict to update the underling dict and flatten_dict by
__setitem__.
```

We also provide the get method to get/collect/generate data:

```
def get(path: Optional[Union[str, list[str]]] = None, keys: Optional[list[Union[str, Union[dict, ndict]]]] = None, default: Optional[Any] = None) -> None:
        # keys: list of path or path-function pairs or path.
        if path is not None:
                # To use the same code to serve the same functionality.
                keys.append(path)

        Convert list keys into dictionary keys: {Path: path or function}

        res = {}
        for p, q in keys.items():
                check if in
                res[p] = q(self, p) if callable(q) else self[p]

        if path and not keys: return res[path]
        else:
                return res
```

The length of the dictionary is defined as the length of the flatten_dict and size can choose to do filtering before count:

```
def __len__(self) -> int:
        return self.size(ignore_none=False)

def size(self, ignore_none: bool = True) -> int:
        d = {v for v in self.values() if v is not None} if ignore_none else self._flatten_dict
        return len(d)
```

To compare different ndicts, we provide two approaches: equality test and difference computing:

```
# Equality
def __eq__(self, other: Union[ndict, dict]) -> bool:
        return self.eq(other, ignore_none=False, keys=None, includes=None, excludes=None, ignore_missing=False)

def _get_compare_keys(keys: Optional[Union[dict, ndict]], includes: Optional[list[str]] = None, excludes: Optional[list[str]] = None, ignore_none: bool = True, missing_method: str = "ignore") -> dict:
        Generate path-value/callable pairs from flatten_dicts or keys and filtered by includes / excludes.
```

```
        filtered by None.
        filtered by missing.

def _get_compare_value(d: ndict, p: Union[str, callable]) -> Any:
        return self.get(d, default=None) if not isinstance(p) else call(p)

def eq(d: Union[ndict, dict], keys: Optional[Union[dict, ndict]], includes: O
ptional[list[str]] = None, excludes: Optional[list[str]] = None, ignore_none:
bool = True, missing_method: str = "ignore") -> bool:
        # Keys can be path-value pairs | path-callable | callable-callable pa
irs.
        keys = self._get_compare_keys()
        d = ndict(d)
        for p, q in keys.items():
                if self._get_compare_value(self, p) != self._get_compare_valu
e(d, q):
                        return False
        return True

# Search difference
def diff(d: Union[ndict, dict], keys: Optional[Union[dict, ndict]], includes:
Optional[list[str]] = None, excludes: Optional[list[str]] = None, ignore_non
e: bool = True, missing_method: str = "keep") -> dict:
        keys = self._get_compare_keys()
        d = ndict(d)
        res = []
        for p, q in keys.items():
                pv =
                qv =
                if not self._compare():
                        res.append(({p: self.get(p, default=None)}, {}))
        return True
```

To test if a path is in the ndcit:

```
@property
def paths(self) -> list[str]:
        return list[self._flatten_dict.keys()]

def __contain__(self, path: str) -> bool:
        return path in self.paths
```

For print:

```
def __str__(self) -> str:
        return yaml.dumps(self.d)

def __repr__(self) -> str:
        return f"<ndict of {len(self)} leaves.>"
```

nconfig

To restrict the type, we overwrite the `__setitem__` method:

```
def _get_check_type_func(self, enable: bool = True):
        return check_type

def __setitem__(self, path: str, value: Any) -> None:
        # Check type
        super()
```

We also check the leaves type immediately after initialization.

For nconfig class, we allow the user to intialize the class using config path and from cli arguments. If users use a dictionary to initialize the class, the config_path should be provided. Otherwise, the class will automatically determine the config_path following: 1) in ["__naapc__;config_path"]; 2) work_directory/config; 3) work_directory/configs; 4) work_directory/conf; 5) work_directory/confs; 6) work_directory. For composition, we generate the configurations and naapc configurations according to their orders in the `__configs__` entry independently. Therefore, the final naapc configuration is from the main configurations. The exception is the `__config_path__` which is only determined in the main configurations. It shouldn't appear in the sub-configurations and won't work any way. For the composition, we generate the configurations recursively.

```
class nconfig(ndict)
        def __init__(self, path: Optional[Union[Path, str]] = None, config: O
ptional[Union[dict, ndcit]] = None, config_path: Optional[Union[Path, str]] =
None, delimiter: str = ";", type_restriction: bool = True) -> None:
                # Check if path or config
                self._GLOBAL_PAH, self._CONFIG_PATH = slef._get_paths(path, c
onfig, config_path)
                configs = self._compose_configs(path, config, delimiter)
                super()
                self._check_type_func = self._get...

```

```
                check type

        def _compose_configs(d: ndict, delimiter) -> ndict:
                Get naapc configurations or use the default ones.
                Get arg configurations.
                Iterate through the configuration dictionary.
                        if is a path:
                                read the dictionary as ndict.
                                tmp_arg_specs, configs[""] = go to composisio
n again.
                                update_arg_specs.
                return arg_specs, configurations.
```

Note that we also support to initialize an empty configuration object.

Users may use update to update the configurations from namespace.

```
def update(d: Union[Union[ndict, dict], namespace], ...):
        if ndict, dict:
                super()
        iterate through the namespace and update accordingly.
```

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. Lazy evaluation of tree size, flattened dictionary, paths should be implemented.
6. A way to make the traverse algorithms readonly (may improve the efficiency).
7. Consider if needed to maintain a state_dict() or meta().
8. ndict should be a subclass of python dict.
9. Raise exception if the arg_specs contain specifications of configurations leaves not in the current configuration.