MetaNN

Release 0.1.2

Hanqiao Yu

Jun 13, 2019

CONTENTS:

CHAPTER

ONE

METANN FOR PYTORCH META LEARNING

1.1 1. Introduction

In meta learner scenario, it is common use dependent variables as parameters, and back propagate the gradient of the parameters. However, parameters of PyTorch Module are designed to be leaf nodes and it is forbidden for parameters to have grad_fn. Meta learning coders are therefore forced to rewrite the basic layers to adapt the meta learning requirements.

This module provide an extension of torch.nn.Module, DependentModule that has dependent parameters, allowing the differentiable dependent parameters. It also provide the method to transform nn.Module into DependentModule, and turning all of the parameters of a nn.Module into dependent parameters.

1.2 2. Installation

```
pip install MetaNN
```

1.3 3. Example

```
from metann import DependentModule, Learner
from torch import nn
net = torch.nn.Sequential(
    nn.Linear(10, 100),
    nn.Linear(100, 5))
net = DependentModule(net)
print(net)
```

1.4 4. Documents

MetaNN

This won't build correctly with the heavy dependency PyTorch, so I updated the sphinx built html to GitHub. I hate to use mock to solve This problem, I suggest you to clone the repository and view the html docs yourself.

1.5 5. License

MIT

CHAPTER

TWO

METANN PACKAGE

2.1 Module contents

class metann.**DependentModule**(**args*, ***kwargs*)

Bases: torch.nn.modules.module.Module

This module provides an extension to nn.Module by add a subset to buffers, dependents. They are similar to parameter, but they are registered in buffers, so that they can have grad_fn. This module calls DependentModule.to_dependentmodule when it is created. It turns the module and all of its submodules into sub class of DependentModule

Examples:

```
>>>net = Sequential(Linear(10, 5), Linear(5, 2))
>>>DependentModule(net)
DependentSequential(
  (0): DependentLinear(in_features=10, out_features=5, bias=True)
  (1): DependentLinear(in_features=5, out_features=2, bias=True)
)
```

Note: This class change the origin module when initializing, you might use

>>>DependentModule(deepcopy(net))

if you want the origin model stay unchanged.

---

clear_params(*init=False*, *clear_filter=<function DependentModule.<lambda>>*)

Clear all parameters of self and register them as dependents. :param init: Set the values of dependents to None if set to False, otherwise keep the value of origin parameters. :param clear_filter: Function that return False when those modules you don't want to clear parameters are input

dependents(*recurse=True*)

Parameters **recurse** – traverse only the direct submodules of self if set to False

Returns iterator of dependents of self and sub modules.

named_dependents(*prefix=''*, *recurse=True*)

Parameters

- **prefix** – the prefix of the names
- **recurse** – traverse only the direct submodules of self if set to False

Returns iterator of name, dependent pairs of self and sub modules.

register_dependent(*name*, *tensor*)
register a named tensor to dependents. :param name: name of dependent :param tensor:

Examples:

```
>>>dnet = DependentModule(net)
>>>dnet.register_dependent('some_tensor', torch.randn(3, 3))
>>>dnet.some_tensor
tensor([[ 0.4434,  0.9949, -0.4385],
        [-0.5292,  0.2555,  0.7772],
        [-0.5386,  0.6152, -0.3239]])
```

classmethod stateless(*module: torch.nn.modules.module.Module*, *clear_filter=<function DependentModule.<lambda>>*)
transform input module into a DependentModule whose parameters are cleared. :param module: :param clear_filter:

substitute(*named_params*, *strict=True*)
Substitute self's dependents with the tensors of same name :param named_params: iterator of name, tensor pairs :param strict: forbid named_params and self._dependents mismatch if set to True. default: True

substitute_from_list(*params*)
Substitute from tensor list. :param params: iterator of tensors

classmethod to_dependentmodule(*module: torch.nn.modules.module.Module*, *recurse=True*)

update_shapes()
update the register shape of dependents. Call this method when a dependent is initialize with None and assign to a tensor. **Do not** call this method when you are using built-in methods only. :return:

class metann.**Learner**(*module: torch.nn.modules.module.Module*)
Bases: `torch.nn.modules.module.Module`

This module extends nn.Module by providing functional method. :param module: a nn.Module module

functional(*params*, *training*, **args*, ***kwargs*)

Parameters

- **params** (*iterable*) – input model parameters for functional
- **training** – if the functional set to trainning=True
- **args** – input
- **kwargs** – input

Returns return the output of model

Examples:

```
>>>learner = Learner(net)
>>>outputs = learner.functional(net.parameters(), training=True, x)
```

2.2 Subpackages

2.2.1 metann.utils package

Module contents

class metann.utils.**SubDict**(*super_dict: collections.abc.Mapping*, *keys=[]*, *keep_order=True*)
Bases: collections.abc.MutableMapping

Provide a sub dict **access** to a super dict. Parameters:

param super_dict (Mapping) The super dictionary where you want to take a sub dict

param keys (iterable) An iterable of keys according to which you want to access a sub dict

param keep_order (bool) If set to true the sub dict will keep the iteration order of the super dict when it is iterated. Default: True

Examples:

```
>>>super_dict = collections.OrderedDict({'a': 1, 'b': 2, 'c': 3})
>>>sub_dict = SubDict(super_dict, keys=['a', 'b'])
```

update_keys()
This method update the keys of the sub dict when the super dict is modified.

Note: **Do not** call this method when you use the built-in method only.

Returns

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX