MetaNN

Release 0.2.10

Dec 16, 2022

Contents

CHAPTER 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.

CHAPTER 2

2. Installation

```
pip install MetaNN
```

CHAPTER 3

3. Example

PyTorch suggest all parameters of a module to be independent variables. Using DependentModule arbitrary torch.nn.module can be transformed into dependent module.

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

Higher-level api such as MAML class are more recommended to use.

```
from metann.meta import MAML, default_evaluator_classification as evaluator
from torch import nn
net = torch.nn.Sequential(
    nn.Linear(10, 100),
    nn.Linear(100, 5))
)
maml = MAML(net, steps_train=5, steps_eval=10, lr=0.01)
output = maml(data_train)
loss = evaluator(output, data_test)
loss.backward()
```

CHAPTER 4

4. Documents

The documents are available at ReadTheDocs. MetaNN

CHAPTER 5

5. License

MIT

CHAPTER 6

Contents

6.1 metann package

6.1.1 Subpackages

metann.utils package

Submodules

metann.utils.containers module

class metann.utils.containers.**DefaultList**(*factory=<function _none_fun>*, *fill=None*)
Bases: object

fill(*data: collections.abc.Iterable*)

class metann.utils.containers.**MultipleList**(*lst*)
Bases: object

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

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

Parameters

- **super_dict** (*Mapping*) – The super dictionary where you want to take a sub dict
- **keys** (*iterable*) – An iterable of keys according to which you want to access a sub dict
- **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

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

- **super_dict** (*Mapping*) – The super dictionary where you want to take a sub dict
- **keys** (*iterable*) – An iterable of keys according to which you want to access a sub dict
- **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

6.1.2 Submodules

6.1.3 metann.dependentmodule module

class metann.dependentmodule.**DependentModule**(**args*, ***kwargs*)
Bases: torch.nn.modules.module.Module

The PyTorch sugggest all parameters of a module to be independent variables, and forbid a parameter to have a grad_fn. This module provides an extension to nn.Module by register a subset of buffers as **dependents**, which indicates the dependent parameters. This enables the parameters of a DependentModule to be the dependent variables, which is useful in meta learning. This module

calls DependentModule.to_dependentmodule when it is created. It turns the module and all of its submodules into sub class of DependentModule. Then you might use clear_params to transform all parameters to dependents.

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.

Parameters

- **init** (*bool*) – Set the values of dependents to None if set to False, otherwise keep the value of origin parameters.
- **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.

Return type Iterative

named_dependents (*prefix: str = ''*, *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.

Return type Iterative

register_dependent (*name: str*, *tensor: torch.Tensor*) → None

register a named tensor to dependents.

Parameters

- **name** – name of dependent tensor
- **tensor** (*torch.Tensor*) – dependent 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.

Parameters

- **module** –
- **clear_filter** – Function that return False when those modules you don't want to clear parameters are input

substitute(*named_params*, *strict=True*)
Substitute self's dependents with the tensors of same name

Parameters

- **named_params** – iterator of name, tensor pairs
- **strict** (*bool*) – forbid named_params and self._dependents mismatch if set to True. default: True

substitute_from_list(*params*)
Substitute from tensor list.

Parameters **params** – iterator of tensors

classmethod to_dependentmodule(*module: torch.nn.modules.module.Module*, *recurse=True*)
Transform a module and all its submodule into dependent module.

Parameters

- **module** –
- **recurse** – if set to be True all submodules will be transformed into dependent module recursively.

Returns a dependent module

Return type *DependentModule*

update_actives()

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.

6.1.4 metann.meta module

class metann.meta.**GDLearner**(*steps*, *lr*, *create_graph=True*, *evaluator=<function default_evaluator_classification>*)
Bases: *metann.meta.Learner*

forward(*model*, *data*, *inplace=False*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

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

forward(**args*, *inplace=False*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

forward_inplace(*model*, *data*)

forward_pure(*model*, *data*)

class metann.meta.**MAML**(*model*, *steps_train*, *steps_eval*, *lr*, *evaluator=<function default_evaluator_classification>*, *first_order=False*)
Bases: `torch.nn.modules.module.Module`

forward(*data*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class metann.meta.**MAMLpp**(*model*, *steps_train*, *steps_eval*, *lr*, *evaluator=<function default_evaluator_classification>*, *first_order=False*)
Bases: `torch.nn.modules.module.Module`

forward(*data*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class metann.meta.**MultiModel**(*model: metann.proto.ProtoModule*, *fast_weight_lst*)
Bases: `torch.nn.modules.module.Module`

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class metann.meta.**RMSPropLearner**(*lr=0.01*, *alpha=0.99*, *eps=1e-08*, *centered=False*, *create_graph=True*, *evaluator=<function default_evaluator_classification>*, *steps=None*)
Bases: *metann.meta.Learner*

forward_inplace(*model*, *data*, *evaluator=None*)

forward_pure(*model*, *data*, *evaluator=None*)

class metann.meta.**SequentialGDLearner**(*lr*, *momentum=0.5*, *create_graph=True*, *evaluator=<function default_evaluator_classification>*)
Bases: *metann.meta.Learner*

forward_inplace(*model*, *data*, *evaluator=None*)

forward_pure(*model*, *data*, *evaluator=None*, *mimo=False*)

metann.meta.**active_indices**(*lst*)

metann.meta.**default_evaluator_classification**(*model*, *data*, *criterion=CrossEntropyLoss()*)

metann.meta.**mamlpp_evaluator**(*mimo: metann.meta.MultiModel*, *data*, *steps: int*, *evaluator*, *gamma=0.6*)

6.1.5 metann.proto module

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

This module extends nn.Module by providing functional method. It is a stateful module, but allows you to call its stateless functional.

Parameters **module** – a nn.Module module

forward(**args*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

functional(*params*, *training=None*)

Parameters

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

Returns return the output of model

Examples

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

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

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

metann.proto.**tensor_copy**(*tensor_lst*)

6.1.6 Module contents

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

Bases: torch.nn.modules.module.Module

The PyTorch sugggest all parameters of a module to be independent variables, and forbid a parameter to have a grad_fn. This module provides an extension to nn.Module by register a subset of buffers as **dependents**, which indicates the dependent parameters. This enables the parameters of a DependentModule to be the dependent variables, which is useful in meta learning. This module calls DependentModule.to_dependentmodule when it is created. It turns the module and all of its submodules into sub class of DependentModule. Then you might use clear_params to transform all parameters to dependents.

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.

Parameters

- **init** (*bool*) – Set the values of dependents to None if set to False, otherwise keep the value of origin parameters.
- **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.

Return type Iterative

named_dependents(*prefix: str = ''*, *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.

Return type Iterative

register_dependent(*name: str*, *tensor: torch.Tensor*) → None
register a named tensor to dependents.

Parameters

- **name** – name of dependent tensor
- **tensor** (*torch.Tensor*) – dependent 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.

Parameters

- **module** –
- **clear_filter** – Function that return False when those modules you don't want to clear parameters are input

substitute(*named_params*, *strict=True*)
Substitute self's dependents with the tensors of same name

Parameters

- **named_params** – iterator of name, tensor pairs
- **strict** (*bool*) – forbid named_params and self._dependents mismatch if set to True. default: True

substitute_from_list(*params*)
Substitute from tensor list.

Parameters **params** – iterator of tensors

classmethod to_dependentmodule(*module: torch.nn.modules.module.Module*, *recurse=True*)
Transform a module and all its submodule into dependent module.

Parameters

- **module** –
- **recurse** – if set to be True all submodules will be transformed into dependent module recursively.

Returns a dependent module

Return type *DependentModule*

update_actives()

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.

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

This module extends nn.Module by providing functional method. It is a stateful module, but allows you to call its stateless functional.

Parameters **module** – a nn.Module module

forward(**args*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

functional(*params*, *training=None*)

Parameters

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

Returns return the output of model

Examples

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

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

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

P

R

S

T

U