Theseus

Meta AI, FAIR team

Jul 21, 2023

GETTING STARTED

Theseus is a library for differentiable nonlinear optimization built on PyTorch to support constructing various problems in robotics and vision as end-to-end differentiable architectures.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

1.1.1 Prerequisites

- We *strongly* recommend you install `theseus` in a venv or conda environment with Python 3.8-3.10.
- Theseus requires `torch` installation. To install for your particular CPU/CUDA configuration, follow the instructions in the PyTorch website.
- For GPU support, Theseus requires nvcc to compile custom CUDA operations. Make sure it matches the version used to compile pytorch with `nvcc --version`. If not, install it and ensure its location is on your system's `$PATH` variable.
- **Theseus also requires suitesparse, which you can install via:**
  - `sudo apt-get install libsuitesparse-dev` (Ubuntu).
  - `conda install -c conda-forge suitesparse` (Mac).

1.1.2 Installing

pypi

```
pip install theseus-ai
```

We currently provide wheels with our CUDA extensions compiled using CUDA 11.6 and Python 3.10. For other CUDA versions, consider installing from source or using our build script.

Note that `pypi` installation doesn't include our experimental Theseus Labs. For this, please install from source.

From source

The simplest way to install Theseus from source is by running the following (see further below to also include BaSpaCho)

```
git clone https://github.com/facebookresearch/theseus.git
pip install -e .
python -m pytest tests
```

If you are interested in contributing to `theseus`, instead install using

```
pip install -e ".[dev]"
```

and follow the more detailed instructions in CONTRIBUTING.

Installing BaSpaCho extensions from source By default, installing from source doesn't include our BaSpaCho sparse solver extension. For this, follow these steps:

1. Compile BaSpaCho from source following instructions here. We recommend using flags *-DBLA_STATIC=ON -DBUILD_SHARED_LIBS=OFF*.
2. Run

   ```
   
   ```

   git clone https://github.com/facebookresearch/theseus.git && cd theseus BASPACHO_ROOT_DIR=<path/to/root/baspacho/dir> pip install -e .

   where the BaSpaCho root dir must have binaries in the subdirectory *build*.

Unit tests

With `dev` installation, you can run unit tests via

```
python -m pytest tests
```

By default, unit tests include tests for our CUDA extensions. You can add the option *-m "not cudaext"* to skip them when installing without CUDA support. Additionally, the tests for sparse solver BaSpaCho are automatically skipped when its extlib is not compiled.

1.2 Tutorials

See tutorials and examples to learn about the API and usage.

CHAPTER

TWO

CORE MODULE

2.1 Objective

An objective function to optimize (see *theseus.Objective*).

2.2 Cost Function

A term in the objective function as a function of one or more `Variable` objects.

2.3 Variable

A variable in the optimization problem. `Variable` objects are named wrappers for `torch` tensors.

2.4 Cost Weight

A weight for cost functions.

2.5 Reference

theseus.Objective	An objective function to optimize.
theseus.Objective.add	Adds a cost function to the objective.
theseus.Objective.error	Evaluates the error vector.
theseus.Objective.error_metric	Aggregates all cost function errors into a (batched) scalar objective.
theseus.Objective.update	Updates all variables with the given input tensor dictionary.
theseus.Objective.retract_vars_sequence	Retracts an ordered sequence of variables.
theseus.CostFunction	A cost function in a differentiable optimization problem.
theseus.Variable	A variable in a differentiable optimization problem.

2.5.1 theseus.Objective

class theseus.**Objective**(*dtype: Optional[dtype] = None*, *error_metric_fn: Optional[ErrorMetric] = None*, *__allow_mixed_optim_aux_vars__: bool = False*)

An objective function to optimize.

Defines the structure of an optimization problem in Theseus by aggregating *cost functions* into a single objective. The cost functions that comprise the final objective function are specified via the *add()* method. Cost functions are responsible for registering their optimization and auxiliary variables, which are automatically added to the objective's list of variables when a cost function is added. Importantly, optimization variables must be instances of `Manifold` subclasses, while auxiliary variables can be instances of any *Variable* class.

Parameters

- **dtype** (*optional[torch.dtype]*) – the data type to use for all variables. If `None` is passed, then uses `torch.get_default_dtype()`.
- **error_metric_fn** (*optional[callable]*) – a reference to a Python function used to aggregate cost functions into a single objective. Defaults to using the sum of squared costs. If given, it must receive a single tensor as input. The objective will use it to pass the batched concatenated error vector, will all cost function errors concatenated.

__init__(*dtype: Optional[dtype] = None*, *error_metric_fn: Optional[ErrorMetric] = None*, *__allow_mixed_optim_aux_vars__: bool = False*)

Methods

add(cost_function)	Adds a cost function to the objective.
copy()	Creates a new copy of this objective.
dim()	Returns the dimension of the error vector.
erase(name)	Removes a cost function from the objective given its name
error([input_tensors, also_update])	Evaluates the error vector.
error_metric([input_tensors, also_update])	Aggregates all cost function errors into a (batched) scalar objective.
get_aux_var(name)	Returns a reference to the auxiliary variable with the given name.
get_cost_function(name)	Returns a reference to the cost function with the given name.
get_functions_connected_to_aux_var(aux_var)	Gets a list of functions that depend on a given auxiliary variable.
get_functions_connected_to_optim_var(variable)	Gets a list of functions that depend on a given optimization variable.
get_optim_var(name)	Returns a reference to the optimization variable with the given name.
has_aux_var(name)	Checks if an auxiliary variable is used in the objective.
has_cost_function(name)	Checks if a cost function with the given name is in the objective.
has_optim_var(name)	Checks if an optimization variable is used in the objective.
retract_vars_sequence(delta, ordering[, ...])	Retracts an ordered sequence of variables.
size()	Returns the number of cost functions and variables in the objective.
size_aux_vars()	Returns the number of auxiliary variables in the objective.
size_cost_functions()	Returns the number of cost functions in the objective.
size_variables()	Returns the number of optimization variables in the objective.
to(*args, **kwargs)	Applies torch.Tensor.to() to all cost functions in the objective.
update([input_tensors, batch_ignore_mask, ...])	Updates all variables with the given input tensor dictionary.

2.5.2 theseus.Objective.add

Objective.**add**(*cost_function:* CostFunction)

Adds a cost function to the objective.

When a cost function is added, this method goes over its list of registered optimization and auxiliary variables, and adds any of them to the objective's list of variables, as long as a variable with th4 same name hasn't been added before. If any of the cost function's variables has the same as that of a variable previously added to the objective, the method checks that they are referring to the same *theseus.Variable*. If this is not the case, an error will be triggered. In other words, the objective expects to have a unique mapping between variable names and objects.

The same procedure is followed for the cost function's weight.

Parameters
cost_function (*theseus.CostFunction*) – the cost function to be added to the objective.

Warning: If a cost weight registers optimization variables that are not used in any *theseus.CostFunction* objects, these will **NOT** be added to the set of the objective's optimization variables; they will be kept in a separate container. The *update()* method will check for this, and throw a warning whenever this happens. Also note that Theseus always considers cost weights as constants, even if their value depends on variables declared as optimization variables.

2.5.3 theseus.Objective.error

Objective.**error**(*input_tensors: Optional[Dict[str, Tensor]] = None*, *also_update: bool = False*) → Tensor

Evaluates the error vector.

Parameters

- **input_tensors** (*Dict[str, torch.Tensor], optional*) – if given, it must be a dictionary mapping variable names to tensors; if a variable with the given name is registered in the objective, its tensor will be replaced with the one in the dictionary (possibly permanently, depending on the value of `also_update`). Defaults to `None`, in which case the error is evaluated using the current tensors stored in all registered variables.
- **also_update** (*bool, optional*) – if `True`, and `input_tensors` is given, the modified variables are permanently updated with the given tensors. Defaults to `False`, in which case the variables are reverted to the previous tensors after the error is evaluated.

Returns

a tensor of shape (batch_size x error_dim), with the
concatenation of all cost functions error vectors. The order corresponds to the order in which cost functions were added to the objective.

Return type
torch.Tensor

2.5.4 theseus.Objective.error_metric

Objective.**error_metric**(*input_tensors: Optional[Dict[str, Tensor]] = None*, *also_update: bool = False*) → Tensor

Aggregates all cost function errors into a (batched) scalar objective.

Parameters

- **input_tensors** (*Dict[str, torch.Tensor], optional*) – if given, it must be a dictionary mapping variable names to tensors; if a variable with the given name is registered in the objective, its tensor will be replaced with the one in the dictionary (possibly permanently, depending on the value of `also_update`). Defaults to `None`, in which case the error is evaluated using the current tensors stored in all registered variables.
- **also_update** (*bool, optional*) – if `True`, and `input_tensors` is given, the modified variables are permanently updated with the given tensors. Defaults to `False`, in which case the variables are reverted to the previous tensors after the error is evaluated.

Returns

a tensor of shape (batch_size,) with the scalar value of
the objective function.

Return type
torch.Tensor

2.5.5 theseus.Objective.update

`Objective.update(input_tensors: Optional[Dict[str, Tensor]] = None, batch_ignore_mask: Optional[Tensor] = None, _update_vectorization: bool = True)`

Updates all variables with the given input tensor dictionary.

The behavior of this method can be summarized by the following pseudocode:

```
for name, tensor in input_tensors.items():
    var = self.get_var_with_name(name).update(tensor)
check_batch_size_consistency(self.all_variables)
```

Any variables not included in the input tensors dictionary will retain their current tensors.

After updating, the objective will modify its batch size property according to the resulting tensors. Therefore, all variable tensors must have a consistent batch size (either 1 or the same value as the others), after the update is completed. Note that this includes variables not referenced in the `input_tensors` dictionary.

Parameters

- **input_tensors** (*Dict[str, torch.Tensor], optional*) – if given, it must be a dictionary mapping variable names to tensors; if a variable with the given name is registered in the objective, its tensor will be replaced with the one in the dictionary (possibly permanently, depending on the value of `also_update`). Defaults to `None`, in which case nothing will be updated. In both cases, the objective will resolve the batch size with whatever tensors are stored after updating.
- **batch_ignore_mask** (*torch.Tensor, optional*) – an optional tensor of shape (batch_size,) of boolean type. Any `True` values indicate that this batch index should remain unchanged in all variables. Defaults to `None`.

Raises
ValueError – if tensors with inconsistent batch dimension are given.

2.5.6 theseus.Objective.retract_vars_sequence

`Objective.retract_vars_sequence(delta: Tensor, ordering: Iterable[Manifold], ignore_mask: Optional[Tensor] = None, force_update: bool = False)`

Retracts an ordered sequence of variables.

The behavior of this method can be summarized by the following pseudocode:

```
for var in ordering:
    var.retract(delta[var_idx])
```

This function assumes that `delta` is constructed as follows:

```
delta = torch.cat([delta_v1, delta_v2, ..., delta_vn], dim=-1)
```

For an ordering `[v1 v2 ... vn]`, and where `delta_vi.shape = (batch_size, vi.dof())`

Parameters

- **delta** (*torch.Tensor*) – the tensor to use for retract operation.
- **ordering** (*Iterable[Manifold]*) – an ordered iterator of variables to retract. The order must be consistent with `delta` as explained above.
- **ignore_mask** (*torch.Tensor, optional*) – An ignore mask for batch indices as in *update()*. Defaults to `None`.
- **force_update** (*bool, optional*) – if `True`, disregards the `ignore_mask`. Defaults to `False`.

2.5.7 theseus.CostFunction

class theseus.**CostFunction**(*cost_weight: CostWeight*, *name: Optional[str] = None*)

A cost function in a differentiable optimization problem.

__init__(*cost_weight: CostWeight*, *name: Optional[str] = None*)

2.5.8 theseus.Variable

class theseus.**Variable**(*tensor: Tensor*, *name: Optional[str] = None*)

A variable in a differentiable optimization problem.

__init__(*tensor: Tensor*, *name: Optional[str] = None*)

CHAPTER

THREE

EMBODIED MODULE

CHAPTER

FOUR

GEOMETRY MODULE

CHAPTER

FIVE

OPTIMIZER MODULE

CHAPTER

SIX

UTILITIES MODULE

INDEX

Symbols

A

C

E

O

R

U

V