TorchOpt

Release 0.6.1.dev1+gb15eb1f

TorchOpt Contributors

Dec 07, 2022

GETTING STARTED

TorchOpt is a high-performance optimizer library built upon PyTorch for easy implementation of functional optimization and gradient-based meta-learning. It consists of two main features:

- TorchOpt provides functional optimizer which enables JAX-like composable functional optimizer for PyTorch. With TorchOpt, one can easily conduct neural network optimization in PyTorch with functional style optimizer, similar to Optax in JAX.
- With the design of functional programming, TorchOpt provides efficient, flexible, and easy-to-implement differentiable optimizer for gradient-based meta-learning research. It largely reduces the efforts required to implement sophisticated meta-learning algorithms.

CHAPTER

ONE

INSTALLATION

Requirements:

- PyTorch
- (Optional) Graphviz

Please follow the instructions at https://pytorch.org to install PyTorch in your Python environment first. Then run the following command to install TorchOpt from PyPI:

```
pip install torchopt
```

You can also build shared libraries from source, use:

```
git clone https://github.com/metaopt/torchopt.git
cd torchopt
pip3 install .
```

We provide a conda environment recipe to install the build toolchain such as *cmake*, *g++*, and *nvcc*:

```
git clone https://github.com/metaopt/torchopt.git
cd torchopt

# You may need `CONDA_OVERRIDE_CUDA` if conda fails to detect the NVIDIA driver (e.g. in␣
↪docker or WSL2)
CONDA_OVERRIDE_CUDA=11.7 conda env create --file conda-recipe-minimal.yaml

conda activate torchopt
```

1.1 Get Started with Jupyter Notebook

In this tutorial, we will use Google Colab notebooks to show you the most basic usages of TorchOpt.

- 1: Functional Optimizer
- 2: Visualization
- 3: Meta-Optimizer
- 4: Stop Gradient
- 5: Implicit Differentiation
- 6: Zero-order Differentiation

1.2 Model-Agnostic Meta-Learning

Meta-reinforcement learning has achieved significant successes in various applications. **Model-Agnostic Meta-Learning** (MAML) [FAL17] is the pioneer one. In this tutorial, we will show how to train MAML on few-shot Omniglot classification with TorchOpt step by step. The full script is at examples/few-shot/maml_omniglot.py.

Contrary to existing differentiable optimizer libraries such as higher, which follows the PyTorch designing which leads to inflexible API, TorchOpt provides an easy way of construction through the code-level.

1.2.1 Overview

There are six steps to finish MAML training pipeline:

1. Load Dataset: load Omniglot dataset;
2. Build the Network: build the neural network architecture of model;
3. Train: meta-train;
4. Test: meta-test;
5. Plot: plot the results;
6. Pipeline: combine step 3-5 together;

In the following sections, we will set up Load Dataset, build the neural network, train-test, and plot to successfully run the MAML training and evaluation pipeline. Here is the overall procedure:

1.2.2 Load Dataset

In your Python code, simply import torch and load the dataset, the full script is at examples/few-shot/support/omniglot_loaders.py:

```
from .support.omniglot_loaders import OmniglotNShot
import torch

device = torch.device('cuda:0')
db = OmniglotNShot(
    '/tmp/omniglot-data',
    batchsz=args.task_num,
    n_way=args.n_way,
    k_shot=args.k_spt,
    k_query=args.k_qry,
    imgsz=28,
    rng=rng,
    device=device,
)
```

The goal is to train a model for few-shot Omniglot classification.

1.2.3 Build the Network

TorchOpt supports any user-defined PyTorch networks. Here is an example:

```
import torch, numpy as np
from torch import nn
import torch.optim as optim

net = nn.Sequential(
    nn.Conv2d(1, 64, 3),
    nn.BatchNorm2d(64, momentum=1.0, affine=True),
    nn.ReLU(inplace=False),
    nn.MaxPool2d(2, 2),
    nn.Conv2d(64, 64, 3),
    nn.BatchNorm2d(64, momentum=1.0, affine=True),
    nn.ReLU(inplace=False),
    nn.MaxPool2d(2, 2),
    nn.Conv2d(64, 64, 3),
    nn.BatchNorm2d(64, momentum=1.0, affine=True),
    nn.ReLU(inplace=False),
    nn.MaxPool2d(2, 2),
    nn.Flatten(),
    nn.Linear(64, args.n_way),
).to(device)

# We will use Adam to (meta-)optimize the initial parameters
# to be adapted.
meta_opt = optim.Adam(net.parameters(), lr=1e-3)
```

1.2.4 Train

Define the `train` function:

```
def train(db, net, meta_opt, epoch, log):
    net.train()
    n_train_iter = db.x_train.shape[0] // db.batchsz
    inner_opt = torchopt.MetaSGD(net, lr=1e-1)

    for batch_idx in range(n_train_iter):
        start_time = time.time()
        # Sample a batch of support and query images and labels.
        x_spt, y_spt, x_qry, y_qry = db.next()

        task_num = x_spt.size(0)

        # TODO: Maybe pull this out into a separate module so it
        # doesn't have to be duplicated between `train` and `test`?

        # Initialize the inner optimizer to adapt the parameters to
        # the support set.
        n_inner_iter = 5
```

(continues on next page)

(continued from previous page)

```
        qry_losses = []
        qry_accs = []
        meta_opt.zero_grad()

        net_state_dict = torchopt.extract_state_dict(net)
        optim_state_dict = torchopt.extract_state_dict(inner_opt)
        for i in range(task_num):
            # Optimize the likelihood of the support set by taking
            # gradient steps w.r.t. the model's parameters.
            # This adapts the model's meta-parameters to the task.
            for _ in range(n_inner_iter):
                spt_logits = net(x_spt[i])
                spt_loss = F.cross_entropy(spt_logits, y_spt[i])
                inner_opt.step(spt_loss)

            # The final set of adapted parameters will induce some
            # final loss and accuracy on the query dataset.
            # These will be used to update the model's meta-parameters.
            qry_logits = net(x_qry[i])
            qry_loss = F.cross_entropy(qry_logits, y_qry[i])
            qry_acc = (qry_logits.argmax(dim=1) == y_qry[i]).float().mean()
            qry_losses.append(qry_loss)
            qry_accs.append(qry_acc.item())

            torchopt.recover_state_dict(net, net_state_dict)
            torchopt.recover_state_dict(inner_opt, optim_state_dict)

        qry_losses = torch.mean(torch.stack(qry_losses))
        qry_losses.backward()
        meta_opt.step()
        qry_losses = qry_losses.item()
        qry_accs = 100.0 * np.mean(qry_accs)
        i = epoch + float(batch_idx) / n_train_iter
        iter_time = time.time() - start_time

        print(
            f'[Epoch {i:.2f}] Train Loss: {qry_losses:.2f} | Acc: {qry_accs:.2f} | Time:
↪{iter_time:.2f}'
        )
        log.append(
            {
                'epoch': i,
                'loss': qry_losses,
                'acc': qry_accs,
                'mode': 'train',
                'time': time.time(),
            }
        )
```

1.2.5 Test

Define the `test` function:

```
def test(db, net, epoch, log):
    # Crucially in our testing procedure here, we do *not* fine-tune
    # the model during testing for simplicity.
    # Most research papers using MAML for this task do an extra
    # stage of fine-tuning here that should be added if you are
    # adapting this code for research.
    net.train()
    n_test_iter = db.x_test.shape[0] // db.batchsz
    inner_opt = torchopt.MetaSGD(net, lr=1e-1)

    qry_losses = []
    qry_accs = []

    for batch_idx in range(n_test_iter):
        x_spt, y_spt, x_qry, y_qry = db.next('test')

        task_num = x_spt.size(0)

        # TODO: Maybe pull this out into a separate module so it
        # doesn't have to be duplicated between `train` and `test`?
        n_inner_iter = 5

        net_state_dict = torchopt.extract_state_dict(net)
        optim_state_dict = torchopt.extract_state_dict(inner_opt)
        for i in range(task_num):
            # Optimize the likelihood of the support set by taking
            # gradient steps w.r.t. the model's parameters.
            # This adapts the model's meta-parameters to the task.
            for _ in range(n_inner_iter):
                spt_logits = net(x_spt[i])
                spt_loss = F.cross_entropy(spt_logits, y_spt[i])
            inner_opt.step(spt_loss)

            # The query loss and acc induced by these parameters.
            qry_logits = net(x_qry[i]).detach()
            qry_loss = F.cross_entropy(qry_logits, y_qry[i])
            qry_acc = (qry_logits.argmax(dim=1) == y_qry[i]).float().mean()
            qry_losses.append(qry_loss.item())
            qry_accs.append(qry_acc.item())

            torchopt.recover_state_dict(net, net_state_dict)
            torchopt.recover_state_dict(inner_opt, optim_state_dict)

    qry_losses = np.mean(qry_losses)
    qry_accs = 100.0 * np.mean(qry_accs)

    print(f'[Epoch {epoch+1:.2f}] Test Loss: {qry_losses:.2f} | Acc: {qry_accs:.2f}')
    log.append(
        {
```

(continues on next page)

(continued from previous page)

```
            'epoch': epoch + 1,
            'loss': qry_losses,
            'acc': qry_accs,
            'mode': 'test',
            'time': time.time(),
        }
    )
```

1.2.6 Plot

TorchOpt supports any user-defined PyTorch networks and optimizers. Yet, of course, the inputs and outputs must comply with TorchOpt's API. Here is an example:

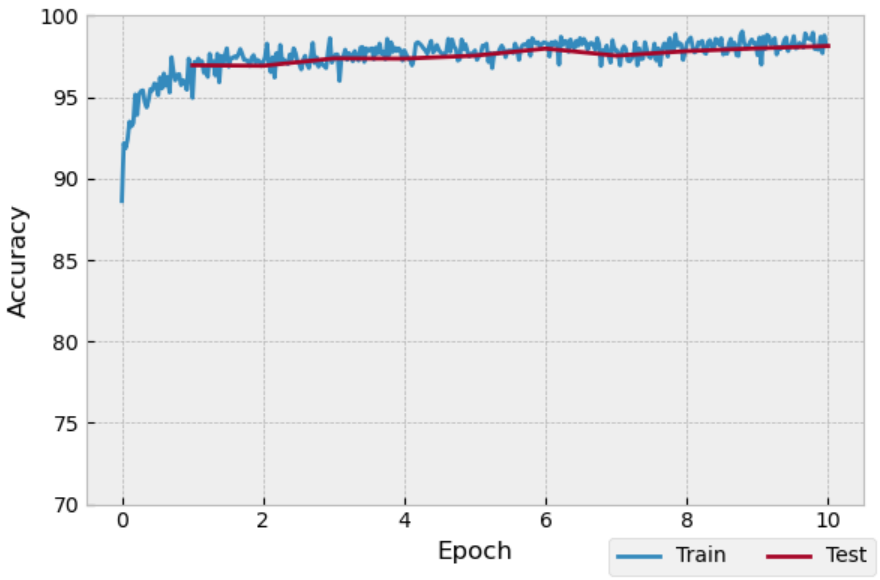
```
def plot(log):
    # Generally you should pull your plotting code out of your training
    # script but we are doing it here for brevity.
    df = pd.DataFrame(log)

    fig, ax = plt.subplots(figsize=(6, 4))
    train_df = df[df['mode'] == 'train']
    test_df = df[df['mode'] == 'test']
    ax.plot(train_df['epoch'], train_df['acc'], label='Train')
    ax.plot(test_df['epoch'], test_df['acc'], label='Test')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Accuracy')
    ax.set_ylim(70, 100)
    fig.legend(ncol=2, loc='lower right')
    fig.tight_layout()
    fname = 'maml-accs.png'
    print(f'--- Plotting accuracy to {fname}')
    fig.savefig(fname)
    plt.close(fig)
```

1.2.7 Pipeline

We can now combine all the components together, and plot the results.

```
log = []
for epoch in range(10):
    train(db, net, meta_opt, epoch, log)
    test(db, net, epoch, log)
    plot(log)
```



References

1.3 Contributing to TorchOpt

Before contributing to TorchOpt, please follow the instructions below to setup.

1. Fork TorchOpt (fork) on GitHub and clone the repository.

```
git clone git@github.com:<your username>/torchopt.git  # use the SSH protocol
cd torchopt

git remote add upstream git@github.com:metaopt/torchopt.git
```

2. Setup a development environment via conda:

```
# You may need `CONDA_OVERRIDE_CUDA` if conda fails to detect the NVIDIA driver (e.g. in␣
↪docker or WSL2)
CONDA_OVERRIDE_CUDA=11.7 conda env create --file conda-recipe.yaml

conda activate torchopt
```

3. Setup the pre-commit hooks:

```
pre-commit install --install-hooks
```

Then you are ready to rock. Thanks for contributing to TorchOpt!

1.3.1 Install Develop Version

To install TorchOpt in an "editable" mode, run:

```
make install-editable  # or run `pip3 install --no-build-isolation --editable .`
```

in the main directory. This installation is removable by:

```
make uninstall
```

1.3.2 Lint Check

We use several tools to secure code quality, including:

- PEP8 code style: `black`, `isort`, `pylint`, `flake8`
- Type hint check: `mypy`
- C++ Google-style: `cpplint`, `clang-format`
- License: `addlicense`
- Documentation: `pydocstyle`, `doc8`

To make things easier, we create several shortcuts as follows.

To automatically format the code, run:

```
make format
```

To check if everything conforms to the specification, run:

```
make lint
```

1.3.3 Test Locally

This command will run automatic tests in the main directory:

```
make test
```

1.3.4 Build Wheels

To build compatible **manylinux2014** (**PEP 599**) wheels for distribution, you can use `cibuildwheel`. You will need to install `docker` first. Then run the following command:

```
pip3 install --upgrade cibuildwheel

export TEST_TORCH_SPECS="cpu cu116"  # `torch` builds for testing
export CUDA_VERSION="11.7"           # version of `nvcc` for compilation
python3 -m cibuildwheel --platform=linux --output-dir=wheelhouse --config-file=pyproject.
↪toml
```

It will install the CUDA compiler with `CUDA_VERSION` in the build container. Then build wheel binaries for all supported CPython versions. The outputs will be placed in the `wheelhouse` directory.

To build a wheel for a specific CPython version, you can use the `CIBW_BUILD` environment variable. For example, the following command will build a wheel for Python 3.7:

```
CIBW_BUILD="cp37*manylinux*" python3 -m cibuildwheel --platform=linux --output-
↪dir=wheelhouse --config-file=pyproject.toml
```

You can change `cp37*` to `cp310*` to build for Python 3.10. See https://cibuildwheel.readthedocs.io/en/stable/options for more options.

1.3.5 Documentation

Documentations are written under the docs/source directory as ReStructuredText (`.rst`) files. `index.rst` is the main page. A Tutorial on ReStructuredText can be found here.

API References are automatically generated by Sphinx according to the outlines under directory docs/source/api and should be modified when any code changes.

To compile documentation into webpage, run

```
make docs
```

The generated webpage locates under directory `docs/build` and will open the browser after building documentation.

Detailed documentation is hosted online at https://torchopt.readthedocs.io.

1.4 Contributor

We always welcome contributions to help make TorchOpt better. Below is an incomplete list of our contributors (find more on this page).

- Yao Fu (future-xy)
- Vincent Moens (vmoens)

1.5 TorchOpt Optimizer

Optimizer(params, impl)	A base class for classic optimizers that similar to `torch.optim.Optimizer`.
MetaOptimizer(module, impl)	The base class for high-level differentiable optimizers.

1.5.1 Optimizer

class torchopt.**Optimizer**(*params*, *impl*)

Bases: object

A base class for classic optimizers that similar to `torch.optim.Optimizer`.

The `init()` function.

Parameters

- **params** (*iterable of torch.Tensor*) – An iterable of `torch.Tensor`s. Specifies what tensors should be optimized.
- **impl** (*GradientTransformation*) – A low level optimizer function, it could be a optimizer function provided by `alias.py` or a customized `chain` provided by `combine.py`. Note that using `Optimizer(sgd())` or `Optimizer(chain(sgd()))` is equivalent to *torchopt.SGD*.

__init__(*params*, *impl*)

The `init()` function.

Parameters

- **params** (*iterable of torch.Tensor*) – An iterable of `torch.Tensor`s. Specifies what tensors should be optimized.
- **impl** (*GradientTransformation*) – A low level optimizer function, it could be a optimizer function provided by `alias.py` or a customized `chain` provided by `combine.py`. Note that using `Optimizer(sgd())` or `Optimizer(chain(sgd()))` is equivalent to *torchopt.SGD*.

zero_grad(*set_to_none=False*)

Sets the gradients of all optimized `torch.Tensor`s to zero.

The behavior is similar to `torch.optim.Optimizer.zero_grad()`.

Parameters

set_to_none (*bool*) – Instead of setting to zero, set the `grads` to `None`.

Return type

None

state_dict()

Returns the state of the optimizer.

Return type

Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...]

load_state_dict(*state_dict*)

Loads the optimizer state.

Parameters

state_dict (Sequence[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]]) – Optimizer state. Should be an object returned from a call to *state_dict()*.

Return type

None

step(*closure=None*)

Performs a single optimization step.

The behavior is similar to `torch.optim.Optimizer.step()`.

Parameters

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

Return type

Optional[Tensor]

add_param_group(*params*)

Add a param group to the optimizer's `param_groups`.

Return type

None

1.5.2 MetaOptimizer

class torchopt.**MetaOptimizer**(*module*, *impl*)

Bases: object

The base class for high-level differentiable optimizers.

The `init()` function.

Parameters

- **module** (Module) – (nn.Module) A network whose parameters should be optimized.
- **impl** (GradientTransformation) – (GradientTransformation) A low level optimizer function, it could be a optimizer function provided by `alias.py` or a customized `chain` provided by `combine.py`. Note that using `MetaOptimizer(sgd(moment_requires_grad=True))` or `MetaOptimizer(chain(sgd(moment_requires_grad=True)))` is equivalent to *torchopt.MetaSGD*.

__init__(*module*, *impl*)

The `init()` function.

Parameters

- **module** (Module) – (nn.Module) A network whose parameters should be optimized.
- **impl** (GradientTransformation) – (GradientTransformation) A low level optimizer function, it could be a optimizer function provided by `alias.py` or a customized `chain` provided by `combine.py`. Note that using `MetaOptimizer(sgd(moment_requires_grad=True))` or `MetaOptimizer(chain(sgd(moment_requires_grad=True)))` is equivalent to *torchopt.MetaSGD*.

step(*loss*)

Compute the gradients of the loss to the network parameters and update network parameters.

Graph of the derivative will be constructed, allowing to compute higher order derivative products. We use the differentiable optimizer (pass argument `inplace=False`) to scale the gradients and update the network parameters without modifying tensors in-place.

Parameters
loss (Tensor) – (torch.Tensor) The loss that is used to compute the gradients to the network parameters.

Return type
None

add_param_group(*module*)

Add a param group to the optimizer's state_groups.

Return type
None

state_dict()

Extract the references of the optimizer states.

Note that the states are references, so any in-place operations will change the states inside *MetaOptimizer* at the same time.

Return type
Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...]

load_state_dict(*state_dict*)

Load the references of the optimizer states.

Return type
None

1.6 Functional Optimizers

FuncOptimizer(impl, *[, inplace])	A wrapper class to hold the functional optimizer.
adam([lr, betas, eps, weight_decay, ...])	The functional Adam optimizer.
sgd(lr[, momentum, dampening, weight_decay, ...])	The functional version of the canonical Stochastic Gradient Descent optimizer.
rmsprop([lr, alpha, eps, weight_decay, ...])	The functional version of the RMSProp optimizer.
adamw([lr, betas, eps, weight_decay, ...])	Adam with weight decay regularization.

1.6.1 Wrapper for Function Optimizer

class torchopt.**FuncOptimizer**(*impl*, *, *inplace=False*)

Bases: object

A wrapper class to hold the functional optimizer.

This wrapper makes it easier to maintain the optimizer states. The optimizer states are held by the wrapper internally. The wrapper provides a *step()* function to compute the gradients and update the parameters.

See also:

- The functional Adam optimizer: *torchopt.adam()*.
- The functional AdamW optimizer: *torchopt.adamw()*.

- The functional RMSprop optimizer: *torchopt.rmsprop()*.
- The functional SGD optimizer: *torchopt.sgd()*.

The `init()` function.

Parameters

- **impl** (*GradientTransformation*) – A low level optimizer function, it could be a optimizer function provided by *alias.py* or a customized *chain* provided by *combine.py*.
- **inplace** (*optional*) – (default: `False`) The default value of `inplace` for each optimization update.

__init__(*impl*, *, *inplace=False*)

The `init()` function.

Parameters

- **impl** (*GradientTransformation*) – A low level optimizer function, it could be a optimizer function provided by *alias.py* or a customized *chain* provided by *combine.py*.
- **inplace** (*optional*) – (default: `False`) The default value of `inplace` for each optimization update.

step(*loss*, *params*, *inplace=None*)

Compute the gradients of loss to the network parameters and update network parameters.

Graph of the derivative will be constructed, allowing to compute higher order derivative products. We use the differentiable optimizer (pass argument inplace=False) to scale the gradients and update the network parameters without modifying tensors in-place.

Parameters

- **loss** (`Tensor`) – (torch.Tensor) loss that is used to compute the gradients to network parameters.
- **params** (`Union`[`Tensor`, `Tuple`[TensorTree, `...`], `List`[TensorTree], `Dict`[`Any`, TensorTree], `Deque`[TensorTree], `CustomTreeNode`[TensorTree]]) – (tree of torch.Tensor) An tree of `torch.Tensors`. Specifies what tensors should be optimized.
- **inplace** (*optional*) – (default: `None`) Whether to update the parameters in-place. If `None`, use the default value specified in the constructor.

Return type

`Union`[`Tensor`, `Tuple`[TensorTree, `...`], `List`[TensorTree], `Dict`[`Any`, TensorTree], `Deque`[TensorTree], `CustomTreeNode`[TensorTree]]

state_dict()

Extract the references of the optimizer states.

Note that the states are references, so any in-place operations will change the states inside *FuncOptimizer* at the same time.

Return type

`Union`[`Tensor`, `Tuple`[TensorTree, `...`], `List`[TensorTree], `Dict`[`Any`, TensorTree], `Deque`[TensorTree], `CustomTreeNode`[TensorTree]]

load_state_dict(*state_dict*)

Load the references of the optimizer states.

Return type

`None`

1.6.2 Functional Adam Optimizer

torchopt.**adam**(*lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.0*, ***, *eps_root=0.0*, *moment_requires_grad=False*, *maximize=False*, *use_accelerated_op=False*)

The functional Adam optimizer.

Adam is an SGD variant with learning rate adaptation. The *learning rate* used for each weight is computed from estimates of first- and second-order moments of the gradients (using suitable exponential moving averages).

References

- Kingma et al, 2014: https://arxiv.org/abs/1412.6980

Parameters

- **lr** (Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]) – (default: 1e-3) This is a fixed global scaling factor.
- **betas** (Tuple[float, float]) – (default: (0.9, 0.999)) Coefficients used for computing running averages of gradient and its square.
- **eps** (float) – (default: 1e-8) A small constant applied to denominator outside of the square root (as in the Adam paper) to avoid dividing by zero when rescaling.
- **weight_decay** (float) – (default: 0.0) Weight decay, add L2 penalty to parameters.
- **eps_root** (float) – (default: 0.0) A small constant applied to denominator inside the square root (as in RMSProp), to avoid dividing by zero when rescaling. This is needed for example when computing (meta-)gradients through Adam.
- **moment_requires_grad** (bool) – (default: False) If True the momentums will be created with flag requires_grad=True, this flag is often used in Meta-Learning algorithms.
- **maximize** (bool) – (default: False) Maximize the params based on the objective, instead of minimizing.
- **use_accelerated_op** (bool) – (default: False) If True use our implemented fused operator.

Return type
GradientTransformation

Returns
The corresponding GradientTransformation instance.

See also:

The functional optimizer wrapper *torchopt.FuncOptimizer*.

1.6.3 Functional AdamW Optimizer

torchopt.**adamw**(*lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.01*, ***, *eps_root=0.0*, *mask=None*, *moment_requires_grad=False*, *maximize=False*, *use_accelerated_op=False*)

Adam with weight decay regularization.

AdamW uses weight decay to regularize learning towards small weights, as this leads to better generalization. In SGD you can also use L2 regularization to implement this as an additive loss term, however L2 regularization does not behave as intended for adaptive gradient algorithms such as Adam.

References

- Loshchilov et al, 2019: https://arxiv.org/abs/1711.05101

Parameters

- **lr** (Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]) – (default: 1e-3) This is a fixed global scaling factor.
- **betas** (Tuple[float, float]) – (default: (0.9, 0.999)) Coefficients used for computing running averages of gradient and its square.
- **eps** (float) – (default: 1e-8) A small constant applied to denominator outside of the square root (as in the Adam paper) to avoid dividing by zero when rescaling.
- **weight_decay** (float) – (default: 1e-2) Strength of the weight decay regularization. Note that this weight decay is multiplied with the learning rate. This is consistent with other frameworks such as PyTorch, but different from (Loshchilov et al, 2019) where the weight decay is only multiplied with the "schedule multiplier", but not the base learning rate.
- **eps_root** (float) – (default: 0.0) A small constant applied to denominator inside the square root (as in RMSProp), to avoid dividing by zero when rescaling. This is needed for example when computing (meta-)gradients through Adam.
- **mask** (Optional[Union[Any, Callable[[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Any]]) – (default: None) A tree with same structure as (or a prefix of) the params PyTree, or a Callable that returns such a pytree given the params/updates. The leaves should be booleans, True for leaves/subtrees you want to apply the weight decay to, and False for those you want to skip. Note that the Adam gradient transformations are applied to all parameters.
- **moment_requires_grad** (bool) – (default: False) If True the momentums will be created with flag requires_grad=True, this flag is often used in Meta-Learning algorithms.
- **maximize** (bool) – (default: False) Maximize the params based on the objective, instead of minimizing.
- **use_accelerated_op** (bool) – (default: False) If True use our implemented fused operator.

Return type
GradientTransformation

Returns
The corresponding GradientTransformation instance.

See also:

The functional optimizer wrapper *torchopt.FuncOptimizer*.

1.6.4 Functional SGD Optimizer

torchopt.**sgd**(*lr*, *momentum=0.0*, *dampening=0.0*, *weight_decay=0.0*, *nesterov=False*, ***, *moment_requires_grad=False*, *maximize=False*)

The functional version of the canonical Stochastic Gradient Descent optimizer.

This implements stochastic gradient descent. It also includes support for momentum, and nesterov acceleration, as these are standard practice when using stochastic gradient descent to train deep neural networks.

References

- Sutskever et al, 2013: http://proceedings.mlr.press/v28/sutskever13.pdf

Parameters

- **lr** (Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]) – This is a fixed global scaling factor.
- **momentum** (float) – (default: 0.0) The decay rate used by the momentum term. The momentum is not used when it is set to 0.0.
- **weight_decay** (float) – (default: 0.0) Weight decay, add L2 penalty to parameters.
- **dampening** (float) – (default: 0.0) Dampening for momentum.
- **nesterov** (bool) – (default: False) Whether to use Nesterov momentum.
- **moment_requires_grad** (bool) – (default: False) If True the momentums will be created with flag requires_grad=True, this flag is often used in Meta-Learning algorithms.
- **maximize** (bool) – (default: False) Maximize the params based on the objective, instead of minimizing.

Return type
GradientTransformation

Returns
The corresponding GradientTransformation instance.

See also:

The functional optimizer wrapper *torchopt.FuncOptimizer*.

1.6.5 Functional RMSProp Optimizer

torchopt.**rmsprop**(*lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0.0*, *momentum=0.0*, *centered=False*, ***, *initial_scale=0.0*, *nesterov=False*, *maximize=False*)

The functional version of the RMSProp optimizer.

RMSProp is an SGD variant with learning rate adaptation. The *learning rate* used for each weight is scaled by a suitable estimate of the magnitude of the gradients on previous steps. Several variants of RMSProp can be found in the literature. This alias provides an easy to configure RMSProp optimizer that can be used to switch between several of these variants.

References

- Tieleman and Hinton, 2012: http://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf
- Graves, 2013: https://arxiv.org/abs/1308.0850

Parameters

- **lr** (Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]) – (default: 1e-2) This is a fixed global scaling factor.
- **alpha** (float) – (default: 0.99) Smoothing constant, the decay used to track the magnitude of previous gradients.
- **eps** (float) – (default: 1e-8) A small numerical constant to avoid dividing by zero when rescaling.
- **weight_decay** (float) – (default: 0.0) Weight decay, add L2 penalty to parameters.
- **momentum** (float) – (default: 0.0) The decay rate used by the momentum term. The momentum is not used when it is set to 0.0.
- **centered** (bool) – (default: False) If True, use the variance of the past gradients to rescale the latest gradients.
- **initial_scale** (float) – (default: 0.0) Initialization of accumulators tracking the magnitude of previous updates. PyTorch uses 0.0, TensorFlow 1.x uses 1.0. When reproducing results from a paper, verify the value used by the authors.
- **nesterov** (bool) – (default: False) Whether to use Nesterov momentum.
- **maximize** (bool) – (default: False) Maximize the params based on the objective, instead of minimizing.

Return type
GradientTransformation

Returns
The corresponding GradientTransformation instance.

See also:

The functional optimizer wrapper *torchopt.FuncOptimizer*.

1.7 Classic Optimizers

Adam(params, lr[, betas, eps, weight_decay, ...])	The classic Adam optimizer.
SGD(params, lr[, momentum, weight_decay, ...])	The classic SGD optimizer.
RMSProp(params[, lr, alpha, eps, ...])	The classic RMSProp optimizer.
AdamW(params[, lr, betas, eps, ...])	The classic AdamW optimizer.

1.7.1 Classic Adam Optimizer

class torchopt.**Adam**(*params*, *lr*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.0*, ***, *eps_root=0.0*, *maximize=False*, *use_accelerated_op=False*)

Bases: *Optimizer*

The classic Adam optimizer.

See also:

- The functional Adam optimizer: *torchopt.adam()*.
- The differentiable meta-Adam optimizer: *torchopt.MetaAdam*.

The `init()` function.

Parameters

- **params** (Iterable[Tensor]) – (iterable of torch.Tensor) An iterable of `torch.Tensor`s. Specifies what tensors should be optimized.
- **lr** (Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]) – (default: `1e-3`) This is a fixed global scaling factor.
- **betas** (Tuple[float, float]) – (default: `(0.9, 0.999)`) Coefficients used for computing running averages of gradient and its square.
- **eps** (float) – (default: `1e-8`) A small constant applied to denominator outside of the square root (as in the Adam paper) to avoid dividing by zero when rescaling.
- **weight_decay** (float) – (default: `0.0`) Weight decay, add L2 penalty to parameters.
- **eps_root** (float) – (default: `0.0`) A small constant applied to denominator inside the square root (as in RMSProp), to avoid dividing by zero when rescaling. This is needed for example when computing (meta-)gradients through Adam.
- **maximize** (bool) – (default: `False`) Maximize the params based on the objective, instead of minimizing.
- **use_accelerated_op** (bool) – (default: `False`) If `True` use our implemented fused operator.

1.7.2 Classic AdamW Optimizer

class torchopt.**AdamW**(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.01*, ***, *eps_root=0.0*, *mask=None*, *maximize=False*, *use_accelerated_op=False*)

Bases: *Optimizer*

The classic AdamW optimizer.

See also:

- The functional AdamW optimizer: *torchopt.adamw()*.
- The differentiable meta-AdamW optimizer: *torchopt.MetaAdamW*.

The `init()` function.

Parameters

- **params** (Iterable[Tensor]) – (iterable of torch.Tensor) An iterable of `torch.Tensor`s. Specifies what tensors should be optimized.
- **lr** (Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]) – (default: `1e-3`) This is a fixed global scaling factor.
- **betas** (Tuple[float, float]) – (default: `(0.9, 0.999)`) Coefficients used for computing running averages of gradient and its square.
- **eps** (float) – (default: `1e-8`) A small constant applied to denominator outside of the square root (as in the Adam paper) to avoid dividing by zero when rescaling.
- **weight_decay** (float) – (default: `1e-2`) Strength of the weight decay regularization. Note that this weight decay is multiplied with the learning rate. This is consistent with other frameworks such as PyTorch, but different from (Loshchilov et al, 2019) where the weight decay is only multiplied with the "schedule multiplier", but not the base learning rate.
- **eps_root** (float) – (default: `0.0`) A small constant applied to denominator inside the square root (as in RMSProp), to avoid dividing by zero when rescaling. This is needed for example when computing (meta-)gradients through Adam.
- **mask** (Optional[Union[Any, Callable[[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Any]]]) – (default: None) A tree with same structure as (or a prefix of) the params PyTree, or a Callable that returns such a pytree given the params/updates. The leaves should be booleans, `True` for leaves/subtrees you want to apply the weight decay to, and `False` for those you want to skip. Note that the Adam gradient transformations are applied to all parameters.
- **maximize** (bool) – (default: `False`) Maximize the params based on the objective, instead of minimizing.
- **use_accelerated_op** (bool) – (default: `False`) If `True` use our implemented fused operator.

1.7.3 Classic SGD Optimizer

class torchopt.**SGD**(*params*, *lr*, *momentum=0.0*, *weight_decay=0.0*, *dampening=0.0*, *nesterov=False*, *maximize=False*)

Bases: *Optimizer*

The classic SGD optimizer.

See also:

- The functional SGD optimizer: *torchopt.sgd()*.
- The differentiable meta-SGD optimizer: *torchopt.MetaSGD*.

The `init()` function.

Parameters

- **params** (Iterable[Tensor]) – (iterable of torch.Tensor) An iterable of `torch.Tensor`s. Specifies what tensors should be optimized.
- **lr** (Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]) – This is a fixed global scaling factor.

- **momentum** (float) – (default: 0.0) The decay rate used by the momentum term. The momentum is not used when it is set to 0.0.
- **weight_decay** (float) – (default: 0.0) Weight decay, add L2 penalty to parameters.
- **dampening** (float) – (default: 0.0) Dampening for momentum.
- **nesterov** (bool) – (default: False) Whether to use Nesterov momentum.
- **maximize** (bool) – (default: False) Maximize the params based on the objective, instead of minimizing.

1.7.4 Classic RMSProp Optimizer

class torchopt.**RMSProp**(*params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0.0*, *momentum=0.0*, *centered=False*, ***, *initial_scale=0.0*, *nesterov=False*, *maximize=False*)

Bases: *Optimizer*

The classic RMSProp optimizer.

See also:

- The functional RMSProp optimizer: *torchopt.rmsprop()*.
- The differentiable meta-RMSProp optimizer: *torchopt.MetaRMSProp*.

The *init* function.

Parameters

- **params** (Iterable[Tensor]) – (iterable of torch.Tensor) An iterable of torch.Tensors. Specifies what Tensors should be optimized.
- **lr** (Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]) – (default: 1e-2) This is a fixed global scaling factor.
- **alpha** (float) – (default: 0.99) Smoothing constant, the decay used to track the magnitude of previous gradients.
- **eps** (float) – (default: 1e-8) A small numerical constant to avoid dividing by zero when rescaling.
- **weight_decay** (float) – (default: 0.0) Weight decay, add L2 penalty to parameters.
- **momentum** (float) – (default: 0.0) The decay rate used by the momentum term. The momentum is not used when it is set to 0.0.
- **centered** (bool) – (default: False) If True, use the variance of the past gradients to rescale the latest gradients.
- **initial_scale** (float) – (default: 0.0) Initialization of accumulators tracking the magnitude of previous updates. PyTorch uses 0.0, TensorFlow 1.x uses 1.0. When reproducing results from a paper, verify the value used by the authors.
- **nesterov** (bool) – (default: False) Whether to use Nesterov momentum.
- **maximize** (bool) – (default: False) Maximize the params based on the objective, instead of minimizing.

1.8 Differentiable Meta-Optimizers

MetaAdam(module[, lr, betas, eps, ...])	The differentiable Adam optimizer.
MetaSGD(module, lr[, momentum, ...])	The differentiable Stochastic Gradient Descent optimizer.
MetaRMSProp(module[, lr, alpha, eps, ...])	The differentiable RMSProp optimizer.
MetaAdamW(module[, lr, betas, eps, ...])	The differentiable AdamW optimizer.

1.8.1 Differentiable Meta-Adam Optimizer

class torchopt.**MetaAdam**(*module*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.0*, *, *eps_root=0.0*, *moment_requires_grad=True*, *maximize=False*, *use_accelerated_op=False*)

Bases: *MetaOptimizer*

The differentiable Adam optimizer.

See also:

- The functional Adam optimizer: *torchopt.adam()*.
- The classic Adam optimizer: *torchopt.Adam*.

The `init()` function.

Parameters

- **module** (`Module`) – (nn.Module) A network whose parameters should be optimized.
- **lr** (`Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]`) – (default: `1e-3`) This is a fixed global scaling factor.
- **betas** (`Tuple[float, float]`) – (default: `(0.9, 0.999)`) Coefficients used for computing running averages of gradient and its square.
- **eps** (`float`) – (default: `1e-8`) A small constant applied to denominator outside of the square root (as in the Adam paper) to avoid dividing by zero when rescaling.
- **weight_decay** (`float`) – (default: `0.0`) Weight decay, add L2 penalty to parameters.
- **eps_root** (`float`) – (default: `0.0`) A small constant applied to denominator inside the square root (as in RMSProp), to avoid dividing by zero when rescaling. This is needed for example when computing (meta-)gradients through Adam.
- **moment_requires_grad** (`bool`) – (default: `True`) If `True` the momentums will be created with flag `requires_grad=True`, this flag is often used in Meta-Learning algorithms.
- **maximize** (`bool`) – (default: `False`) Maximize the params based on the objective, instead of minimizing.
- **use_accelerated_op** (`bool`) – (default: `False`) If `True` use our implemented fused operator.

1.8.2 Differentiable Meta-AdamW Optimizer

class torchopt.**MetaAdamW**(*module*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.01*, ***, *eps_root=0.0*, *mask=None*, *moment_requires_grad=False*, *maximize=False*, *use_accelerated_op=False*)

Bases: *MetaOptimizer*

The differentiable AdamW optimizer.

See also:

- The functional AdamW optimizer: *torchopt.adamw()*.
- The classic AdamW optimizer: *torchopt.AdamW*.

The `init()` function.

Parameters

- **module** (`Module`) – (nn.Module) A network whose parameters should be optimized.
- **lr** (`Union`[`float`, `Callable`[[`Union`[`Tensor`, `float`, `int`, `bool`]], `Union`[`Tensor`, `float`, `int`, `bool`]]]) – (default: `1e-3`) This is a fixed global scaling factor.
- **betas** (`Tuple`[`float`, `float`]) – (default: `(0.9, 0.999)`) Coefficients used for computing running averages of gradient and its square.
- **eps** (`float`) – (default: `1e-8`) A small constant applied to denominator outside of the square root (as in the Adam paper) to avoid dividing by zero when rescaling.
- **weight_decay** (`float`) – (default: `1e-2`) Strength of the weight decay regularization. Note that this weight decay is multiplied with the learning rate. This is consistent with other frameworks such as PyTorch, but different from (Loshchilov et al, 2019) where the weight decay is only multiplied with the "schedule multiplier", but not the base learning rate.
- **eps_root** (`float`) – (default: `0.0`) A small constant applied to denominator inside the square root (as in RMSProp), to avoid dividing by zero when rescaling. This is needed for example when computing (meta-)gradients through Adam.
- **mask** (`Optional`[`Union`[`Any`, `Callable`[[`Union`[`Tensor`, `Tuple`[TensorTree, `...`], `List`[TensorTree], `Dict`[`Any`, TensorTree], `Deque`[TensorTree], `CustomTreeNode`[TensorTree]]], `Any`]]]) – (default: `None`) A tree with same structure as (or a prefix of) the params PyTree, or a Callable that returns such a pytree given the params/updates. The leaves should be booleans, `True` for leaves/subtrees you want to apply the weight decay to, and `False` for those you want to skip. Note that the Adam gradient transformations are applied to all parameters.
- **moment_requires_grad** (`bool`) – (default: `False`) If `True` the momentums will be created with flag `requires_grad=True`, this flag is often used in Meta-Learning algorithms.
- **maximize** (`bool`) – (default: `False`) Maximize the params based on the objective, instead of minimizing.
- **use_accelerated_op** (`bool`) – (default: `False`) If `True` use our implemented fused operator.

1.8.3 Differentiable Meta-SGD Optimizer

class torchopt.**MetaSGD**(*module*, *lr*, *momentum=0.0*, *weight_decay=0.0*, *dampening=0.0*, *nesterov=False*, *moment_requires_grad=True*, *maximize=False*)

Bases: *MetaOptimizer*

The differentiable Stochastic Gradient Descent optimizer.

See also:

- The functional SGD optimizer: *torchopt.sgd()*.
- The classic SGD optimizer: *torchopt.SGD*.

The `init()` function.

Parameters

- **module** (`Module`) – (nn.Module) A network whose parameters should be optimized.
- **lr** (`Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]`) – This is a fixed global scaling factor.
- **momentum** (`float`) – (default: `0.0`) The decay rate used by the momentum term. The momentum is not used when it is set to `0.0`.
- **weight_decay** (`float`) – (default: `0.0`) Weight decay, add L2 penalty to parameters.
- **dampening** (`float`) – (default: `0.0`) Dampening for momentum.
- **nesterov** (`bool`) – (default: `False`) Whether to use Nesterov momentum.
- **moment_requires_grad** (`bool`) – (default: `True`) If `True` the momentums will be created with flag `requires_grad=True`, this flag is often used in Meta-Learning algorithms.
- **maximize** (`bool`) – (default: `False`) Maximize the params based on the objective, instead of minimizing.

1.8.4 Differentiable Meta-RMSProp Optimizer

class torchopt.**MetaRMSProp**(*module*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0.0*, *momentum=0.0*, *centered=False*, ***, *initial_scale=0.0*, *nesterov=False*, *maximize=False*)

Bases: *MetaOptimizer*

The differentiable RMSProp optimizer.

See also:

- The functional RMSProp optimizer: *torchopt.rmsprop()*.
- The classic RMSProp optimizer: *torchopt.RMSProp*.

The `init()` function.

Parameters

- **module** (`Module`) – (nn.Module) A network whose parameters should be optimized.
- **lr** (`Union[float, Callable[[Union[Tensor, float, int, bool]], Union[Tensor, float, int, bool]]]`) – (default: `1e-2`) This is a fixed global scaling factor.

- **alpha** (float) – (default: 0.99) Smoothing constant, the decay used to track the magnitude of previous gradients.
- **eps** (float) – (default: 1e-8) A small numerical constant to avoid dividing by zero when rescaling.
- **weight_decay** (float) – (default: 0.0) Weight decay, add L2 penalty to parameters.
- **momentum** (float) – (default: 0.0) The decay rate used by the momentum term. The momentum is not used when it is set to 0.0.
- **centered** (bool) – (default: False) If True, use the variance of the past gradients to rescale the latest gradients.
- **initial_scale** (float) – (default: 0.0) Initialization of accumulators tracking the magnitude of previous updates. PyTorch uses 0.0, TensorFlow 1.x uses 1.0. When reproducing results from a paper, verify the value used by the authors.
- **nesterov** (bool) – (default: False) Whether to use Nesterov momentum.
- **maximize** (bool) – (default: False) Maximize the params based on the objective, instead of minimizing.

1.9 Implicit differentiation

custom_root(optimality_fn, argnums[, ...])	Decorator for adding implicit differentiation to a root solver.
nn.ImplicitMetaGradientModule()	The base class for differentiable implicit meta-gradient models.

1.9.1 Custom solvers

torchopt.diff.implicit.**custom_root**(*optimality_fn*, *argnums*, *has_aux=False*, *solve=functools.partial(<function _solve_normal_cg>)*)

Decorator for adding implicit differentiation to a root solver.

This wrapper should be used as a decorator:

```
def optimality_fn(optimal_params, ...):
    ...
    return residual

@custom_root(optimality_fn, argnums=argnums)
def solver_fn(params, arg1, arg2, ...):
    ...
    return optimal_params

optimal_params = solver_fn(init_params, ...)
```

The first argument to `optimality_fn` and `solver_fn` is preserved as the parameter input. The `argnums` argument refers to the indices of the variables in `solver_fn`'s signature. For example, setting `argnums=(1, 2)` will compute the gradient of `optimal_params` with respect to `arg1` and `arg2` in the above snippet. Note that, except

the first argument, the keyword arguments of the `optimality_fn` should be a subset of the ones of `solver_fn`. **In best practice, the ``optimality_fn`` should have the same signature as ``solver_fn``.**

Parameters

- **optimality_fn** (`Callable[..., Union[Tensor, Sequence[Tensor]]]`) – (callable) An equation function, `optimality_fn(params, *args)`. The invariant is `optimality_fn(solution, *args) == 0` at the solution / root of `solution`.
- **argnums** (`Union[int, Tuple[int, ...]]`) – (int or tuple of ints) Specifies arguments to compute gradients with respect to. The `argnums` can be an integer or a tuple of integers, which respect to the zero-based indices of the arguments of the `solver_fn(params, *args)` function. The argument `params` is included for the counting, while it is indexed as `argnums=0`.
- **has_aux** (`bool`) – (default: `False`) Whether the decorated solver function returns auxiliary data.
- **solve** (`Callable[..., Union[Tensor, Sequence[Tensor]]]`) – (callable, optional, default: `linear_solve.solve_normal_cg()`) a linear solver of the form `solve(matvec, b)`.

Return type

`Callable[[Callable[..., Union[Tensor, Sequence[Tensor], Tuple[Union[Tensor, Sequence[Tensor]], Any]]]], Callable[..., Union[Tensor, Sequence[Tensor], Tuple[Union[Tensor, Sequence[Tensor]], Any]]]]`

Returns

A solver function decorator, i.e., `custom_root(optimality_fn)(solver_fn)`.

1.9.2 Implicit Meta-Gradient Module

class torchopt.diff.implicit.nn.**ImplicitMetaGradientModule**

Bases: `MetaGradientModule`

The base class for differentiable implicit meta-gradient models.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

linear_solve: Optional[Callable[[Callable[[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]]]

classmethod __init_subclass__(*linear_solve=None*)

Validates and initializes the subclass.

Return type

`None`

solve(**input*, ***kwargs*)

Solves the inner optimization problem.

> **Warning:** For gradient-based optimization methods, the parameter inputs should be explicitly specified in the `torch.autograd.backward()` function as argument `inputs`. Otherwise, if not provided, the gradient is accumulated into all the leaf Tensors (including the meta-parameters) that were used to compute the objective output. Alternatively, please use `torch.autograd.grad()` instead.

Example:

```
def solve(self, batch, labels):
    parameters = tuple(self.parameters())
    optimizer = torch.optim.Adam(parameters, lr=1e-3)
    with torch.enable_grad():
        for _ in range(100):
            loss = self.objective(batch, labels)
            optimizer.zero_grad()
            # Only update the `.grad` attribute for parameters
            # and leave the meta-parameters unchanged
            loss.backward(inputs=parameters)
            optimizer.step()
    return self
```

Return type
Any

optimality(**input*, ***kwargs*)

Computes the optimality residual.

This method stands for the optimality residual to the optimal parameters after solving the inner optimization problem (*solve()*), i.e.:

```
module.solve(*input, **kwargs)
module.optimality(*input, **kwargs)  # -> 0
```

1. For gradient-based optimization, the *optimality()* function is the KKT condition, usually it is the gradients of the *objective()* function with respect to the module parameters (not the meta-parameters). If this method is not implemented, it will be automatically derived from the gradient of the *objective()* function.

$$\text{optimality residual} = \nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{\theta}) \to \mathbf{0}$$

where $\boldsymbol{x}$ is the joint vector of the module parameters and $\boldsymbol{\theta}$ is the joint vector of the meta-parameters.

References

- Karush-Kuhn-Tucker (KKT) conditions: https://en.wikipedia.org/wiki/Karush-Kuhn-Tucker_conditions

2. For fixed point iteration, the *optimality()* function can be the residual of the parameters between iterations, i.e.:

$$\text{optimality residual} = f(\boldsymbol{x}, \boldsymbol{\theta}) - \boldsymbol{x} \to \mathbf{0}$$

where $\boldsymbol{x}$ is the joint vector of the module parameters and $\boldsymbol{\theta}$ is the joint vector of the meta-parameters.

Return type

Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], CustomTreeNode[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]]]

Returns

A tree of tensors, the optimality residual to the optimal parameters after solving the inner optimization problem.

objective(**input*, ***kwargs*)

Computes the objective function value.

This method is used to calculate the *optimality()* if it is not implemented. Otherwise, this method is optional.

Return type

Tensor

Returns

A scalar tensor (dim=0), the objective function value.

__annotations__ = {'__call__': 'Callable[..., Any]', '_backward_hooks': 'Dict[int, Callable]', '_buffers': 'Dict[str, Optional[Tensor]]', '_custom_objective': <class 'bool'>, '_custom_optimality': <class 'bool'>, '_forward_hooks': 'Dict[int, Callable]', '_forward_pre_hooks': 'Dict[int, Callable]', '_is_full_backward_hook': 'Optional[bool]', '_load_state_dict_post_hooks': 'Dict[int, Callable]', '_load_state_dict_pre_hooks': 'Dict[int, Callable]', '_meta_inputs': 'MetaInputsContainer', '_meta_modules': 'Dict[str, Optional[nn.Module]]', '_meta_parameters': 'Dict[str, Optional[torch.Tensor]]', '_modules': "Dict[str, Optional['Module']]", '_non_persistent_buffers_set': 'Set[str]', '_parameters': 'Dict[str, Optional[Parameter]]', '_state_dict_hooks': 'Dict[int, Callable]', '_version': 'int', 'dump_patches': 'bool', 'forward': 'Callable[..., Any]', 'linear_solve': typing.Optional[typing.Callable[[typing.Callable[[typing.Union[torch.Tensor, typing.Tuple[ForwardRef('TensorTree'), ...], typing.List[ForwardRef('TensorTree')], typing.Dict[typing.Any, ForwardRef('TensorTree')], typing.Deque[ForwardRef('TensorTree')], optree.typing.CustomTreeNode[ForwardRef('TensorTree')]]], typing.Union[torch.Tensor, typing.Tuple[ForwardRef('TensorTree'), ...], typing.List[ForwardRef('TensorTree')], typing.Dict[typing.Any, ForwardRef('TensorTree')], typing.Deque[ForwardRef('TensorTree')], optree.typing.CustomTreeNode[ForwardRef('TensorTree')]]], typing.Union[torch.Tensor, typing.Tuple[ForwardRef('TensorTree'), ...], typing.List[ForwardRef('TensorTree')], typing.Dict[typing.Any, ForwardRef('TensorTree')], typing.Deque[ForwardRef('TensorTree')], optree.typing.CustomTreeNode[ForwardRef('TensorTree')]]], typing.Union[torch.Tensor, typing.Tuple[ForwardRef('TensorTree'), ...], typing.List[ForwardRef('TensorTree')], typing.Dict[typing.Any, ForwardRef('TensorTree')], typing.Deque[ForwardRef('TensorTree')], optree.typing.CustomTreeNode[ForwardRef('TensorTree')]]]], 'training': 'bool'}

1.10 Linear system solvers

solve_cg(**kwargs)	A wrapper that returns a solver function to solve `A x = b` using conjugate gradient.
solve_normal_cg(**kwargs)	A wrapper that returns a solver function to solve `A^T A x = A^T b` using conjugate gradient.
solve_inv(**kwargs)	A wrapper that returns a solver function to solve `A x = b` using matrix inversion.

1.10.1 Indirect solvers

torchopt.linear_solve.**solve_cg**(***kwargs*)

A wrapper that returns a solver function to solve `A x = b` using conjugate gradient.

This assumes that `A` is a hermitian, positive definite matrix.

Parameters

- **ridge** – Optional ridge regularization. Solves the equation for `(A + ridge * I) @ x = b`.
- **init** – Optional initialization to be used by conjugate gradient.
- ****kwargs** – Additional keyword arguments for the conjugate gradient solver `torchopt.linalg.cg()`.

Returns

A solver function with signature `(matvec, b) -> x` that solves `A x = b` using conjugate gradient where `matvec(v) = A v`.

See also:

Conjugate gradient iteration `torchopt.linalg.cg()`.

Example:

```
>>> A = {'a': torch.eye(5, 5), 'b': torch.eye(3, 3)}
>>> x = {'a': torch.randn(5), 'b': torch.randn(3)}
>>> def matvec(x: TensorTree) -> TensorTree:
...     return {'a': A['a'] @ x['a'], 'b': A['b'] @ x['b']}
>>> b = matvec(x)
>>> solver = solve_cg(init={'a': torch.zeros(5), 'b': torch.zeros(3)})
>>> x_hat = solver(matvec, b)
>>> assert torch.allclose(x_hat['a'], x['a']) and torch.allclose(x_hat['b'], x['b'])
```

torchopt.linear_solve.**solve_normal_cg**(***kwargs*)

A wrapper that returns a solver function to solve `A^T A x = A^T b` using conjugate gradient.

This can be used to solve `A x = b` using conjugate gradient when `A` is not hermitian, positive definite.

Parameters

- **ridge** – Optional ridge regularization. Solves the equation for `(A.T @ A + ridge * I) @ x = A.T @ b`.
- **init** – Optional initialization to be used by normal conjugate gradient.
- ****kwargs** – Additional keyword arguments for the conjugate gradient solver `torchopt.linalg.cg()`.

Returns

A solver function with signature `(matvec, b) -> x` that solves `A^T A x = A^T b` using conjugate gradient where `matvec(v) = A v`.

See also:

Conjugate gradient iteration `torchopt.linalg.cg()`.

Example:

```
>>> A = {'a': torch.randn(5, 5), 'b': torch.randn(3, 3)}
>>> x = {'a': torch.randn(5), 'b': torch.randn(3)}
>>> def matvec(x: TensorTree) -> TensorTree:
...     return {'a': A['a'] @ x['a'], 'b': A['b'] @ x['b']}
>>> b = matvec(x)
>>> solver = solve_normal_cg(init={'a': torch.zeros(5), 'b': torch.zeros(3)})
>>> x_hat = solver(matvec, b)
>>> assert torch.allclose(x_hat['a'], x['a']) and torch.allclose(x_hat['b'], x['b'])
```

`torchopt.linear_solve.`**solve_inv**(***kwargs*)

A wrapper that returns a solver function to solve `A x = b` using matrix inversion.

If `ns = False`, this assumes the matrix `A` is a constant matrix and will materialize it in memory.

Parameters

- **ridge** – Optional ridge regularization. Solves the equation for `(A + ridge * I) @ x = b`.
- **ns** – Whether to use Neumann Series matrix inversion approximation. If `False`, materialize the matrix `A` in memory and use `torch.linalg.solve()` instead.
- ****kwargs** – Additional keyword arguments for the Neumann Series matrix inversion approximation solver `torchopt.linalg.ns()`.

Returns

A solver function with signature `(matvec, b) -> x` that solves `A x = b` using matrix inversion where `matvec(v) = A v`.

See also:

Neumann Series matrix inversion approximation `torchopt.linalg.ns()`.

Example:

```
>>> A = {'a': torch.eye(5, 5), 'b': torch.eye(3, 3)}
>>> x = {'a': torch.randn(5), 'b': torch.randn(3)}
>>> def matvec(x: TensorTree) -> TensorTree:
...     return {'a': A['a'] @ x['a'], 'b': A['b'] @ x['b']}
>>> b = matvec(x)
>>> solver = solve_inv(ns=True, maxiter=10)
>>> x_hat = solver(matvec, b)
>>> assert torch.allclose(x_hat['a'], x['a']) and torch.allclose(x_hat['b'], x['b'])
```

1.11 Optimizer Hooks

register_hook(hook)	Stateless identity transformation that leaves input gradients untouched.
zero_nan_hook(g)	A zero `nan` hook to replace `nan` with zero.
nan_to_num_hook([nan, posinf, neginf])	Returns a `nan` to num hook to replace `nan` / `+inf` / `-inf` with the given numbers.

1.11.1 Hook

torchopt.hook.**register_hook**(*hook*)

Stateless identity transformation that leaves input gradients untouched.

This function passes through the *gradient updates* unchanged.

Return type
`GradientTransformation`

Returns
An `(init_fn, update_fn)` tuple.

torchopt.hook.**zero_nan_hook**(*g*)

A zero `nan` hook to replace `nan` with zero.

Return type
`Tensor`

torchopt.hook.**nan_to_num_hook**(*nan=0.0*, *posinf=None*, *neginf=None*)

Returns a `nan` to num hook to replace `nan` / `+inf` / `-inf` with the given numbers.

Return type
`Callable`[[`Tensor`], `Tensor`]

1.12 Gradient Transformation

clip_grad_norm(max_norm[, norm_type, ...])	Clips gradient norm of an iterable of parameters.
nan_to_num([nan, posinf, neginf])	Replaces updates with values `nan` / `+inf` / `-inf` to the given numbers.

1.12.1 Transforms

torchopt.**clip_grad_norm**(*max_norm*, *norm_type=2.0*, *error_if_nonfinite=False*)

Clips gradient norm of an iterable of parameters.

Parameters

- **max_norm** (*float or int*) – The maximum absolute value for each element in the update.
- **norm_type** (*float or int*) – type of the used p-norm. Can be `'inf'` for infinity norm.
- **error_if_nonfinite** (*bool*) – if `True`, an error is thrown if the total norm of the gradients from `updates` is `nan`, `inf`, or `-inf`.

Return type
`GradientTransformation`

Returns
An `(init_fn, update_fn)` tuple.

torchopt.**nan_to_num**(*nan=0.0*, *posinf=None*, *neginf=None*)

Replaces updates with values `nan` / `+inf` / `-inf` to the given numbers.

Return type
`GradientTransformation`

Returns
An `(init_fn, update_fn)` tuple.

1.13 Optimizer Schedules

linear_schedule(init_value, end_value, ...)	Alias polynomial schedule to linear schedule for convenience.
polynomial_schedule(init_value, end_value, ...)	Constructs a schedule with polynomial transition from init to end value.

1.13.1 Schedules

torchopt.schedule.**linear_schedule**(*init_value*, *end_value*, *transition_steps*, *transition_begin=0*)

Alias polynomial schedule to linear schedule for convenience.

Return type
`Callable`[[`Union`[`Tensor`, `float`, `int`, `bool`]], `Union`[`Tensor`, `float`, `int`, `bool`]]

torchopt.schedule.**polynomial_schedule**(*init_value*, *end_value*, *power*, *transition_steps*, *transition_begin=0*)

Constructs a schedule with polynomial transition from init to end value.

Parameters

- **init_value** (`Union`[`float`, `int`, `bool`]) – Initial value for the scalar to be annealed.
- **end_value** (`Union`[`float`, `int`, `bool`]) – End value of the scalar to be annealed.
- **power** (`Union`[`float`, `int`, `bool`]) – The power of the polynomial used to transition from `init` to end.

- **transition_steps** (int) – Number of steps over which annealing takes place, the scalar starts changing at `transition_begin` steps and completes the transition by `transition_begin + transition_steps` steps. If `transition_steps <= 0`, then the entire annealing process is disabled and the value is held fixed at `init_value`.
- **transition_begin** (int) – Must be *positive*. After how many steps to start annealing (before this many steps the scalar value is held fixed at `init_value`).

Returns
A function that maps step counts to values.

Return type
schedule

1.14 Apply Parameter Updates

apply_updates(params, updates, *[, inplace])	Applies an update to the corresponding parameters.

1.14.1 Apply Updates

torchopt.**apply_updates**(*params*, *updates*, *, *inplace=True*)

Applies an update to the corresponding parameters.

This is a utility functions that applies an update to a set of parameters, and then returns the updated parameters to the caller. As an example, the update may be a gradient transformed by a sequence of `GradientTransformations`. This function is exposed for convenience, but it just adds updates and parameters; you may also apply updates to parameters manually, using `tree_map()` (e.g. if you want to manipulate updates in custom ways before applying them).

Parameters

- **params** (Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], CustomTreeNode[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]]]) – A tree of parameters.
- **updates** (Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], CustomTreeNode[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree],

CustomTreeNode[TensorTree]]]]) – A tree of updates, the tree structure and the shape of the leaf nodes must match that of `params`.

- **inplace** (bool) – If True, will update params in a inplace manner.

Return type
Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], CustomTreeNode[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]]]

Returns
Updated parameters, with same structure, shape and type as `params`.

1.15 Combining Optimizers

chain(*transformations)	Applies a list of chainable update transformations.

1.15.1 Chain

torchopt.combine.**chain**(**transformations*)

Applies a list of chainable update transformations.

Given a sequence of chainable transforms, *chain()* returns an `init_fn()` that constructs a `state` by concatenating the states of the individual transforms, and returns an `update_fn()` which chains the update transformations feeding the appropriate state to each.

Parameters
***transformations** (GradientTransformation) – A sequence of chainable `(init_fn, update_fn)` tuples.

Return type
GradientTransformation

Returns
A single `(init_fn, update_fn)` tuple.

1.16 General Utilities

extract_state_dict()	Extract target state.
recover_state_dict(target, state)	Recover state.
stop_gradient(target)	Stop the gradient for the input object.

1.16.1 Extract State Dict

torchopt.**extract_state_dict**(*target: Module*, *, *by: Literal['reference', 'copy', 'deepcopy', 'ref', 'clone', 'deepclone'] = 'reference'*, *device: Optional[Union[device, str, int]] = None*, *with_buffers: bool = True*, *enable_visual: bool = False*, *visual_prefix: str = ''*) → ModuleState

torchopt.**extract_state_dict**(*target:* MetaOptimizer, *, *by: Literal['reference', 'copy', 'deepcopy', 'ref', 'clone', 'deepclone'] = 'reference'*, *device: Optional[Union[device, str, int]] = None*, *with_buffers: bool = True*, *enable_visual: bool = False*, *visual_prefix: str = ''*) → Tuple[Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], CustomTreeNode[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]]], ...]

Extract target state.

Since a tensor use `grad_fn` to connect itself with the previous computation graph, the backpropagated gradient will flow over the tensor and continue flow to the tensors that is connected by `grad_fn`. Some algorithms requires manually detaching tensors from the computation graph.

Note that the extracted state is a reference, which means any in-place operator will affect the target that the state is extracted from.

Parameters

- **target** (Union[Module, *MetaOptimizer*]) – It could be a `nn.Module` or *torchopt.MetaOptimizer*.
- **by** (Literal['reference', 'copy', 'deepcopy', 'ref', 'clone', 'deepclone']) – The extract policy of tensors in the target. - `'reference'`: The extracted tensors will be references to the original tensors. - `'copy'`: The extracted tensors will be clones of the original tensors. This makes the copied tensors have `grad_fn` to be a `<CloneBackward>` function points to the original tensors. - `'deepcopy'`: The extracted tensors will be deep-copied from the original tensors. The deep-copied tensors will detach from the original computation graph.
- **device** (Optional[Union[device, str, int]]) – If specified, move the extracted state to the specified device.
- **with_buffers** (bool) – Extract buffer together with parameters, this argument is only used if the input target is `nn.Module`.
- **detach_buffers** (bool) – Whether to detach the reference to the buffers, this argument is only used if the input target is `nn.Module` and `by='reference'`.
- **enable_visual** (bool) – Add additional annotations, which could be used in computation graph visualization. Currently, this flag only has effect on `nn.Module` but we will support *torchopt.MetaOptimizer* later.
- **visual_prefix** (str) – Prefix for the visualization annotations.

Return type

Union[ModuleState, Tuple[Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree],

CustomTreeNode[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], CustomTreeNode[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]]], ...]]

Returns

State extracted of the input object.

1.16.2 Recover State Dict

torchopt.**recover_state_dict**(*target*, *state*)

Recover state.

This function is compatible for the `extract_state`.

Note that the recovering process is not in-place, so the tensors of the object will not be modified.

Parameters

- **target** (Union[Module, *MetaOptimizer*]) – Target that need to recover.
- **state** (Union[ModuleState, Sequence[Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], CustomTreeNode[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]]]]]) – The recovering state.

Return type

None

1.16.3 Stop Gradient

torchopt.**stop_gradient**(*target*)

Stop the gradient for the input object.

Since a tensor use `grad_fn` to connect itself with the previous computation graph, the backpropagated gradient will flow over the tensor and continue flow to the tensors that is connected by `grad_fn`. Some algorithms requires manually detaching tensors from the computation graph.

Note that the *stop_gradient()* operation is in-place.

Parameters

- **target** (Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree,

...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], CustomTreeNode[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], CustomTreeNode[TensorTree]]], ModuleState, Module, *MetaOptimizer*]) – The target that to be detached from the computation graph, it could be a `nn.Module`, *torchopt.MetaOptimizer*, state of the *torchopt.MetaOptimizer*, or just a plain list of tensors.

- **inplace** – If `True`, the target will be detached in-place. if `Frue`, this function will return a detached copy of the target. The in-place operation is fast and memory efficient but may raise backpropagation error.

Return type
`None`

1.17 Visualizing Gradient Flow

make_dot(var[, params, show_attrs, ...])	Produces Graphviz representation of PyTorch autograd graph.

1.17.1 Make Dot

torchopt.visual.**make_dot**(*var*, *params=None*, *show_attrs=False*, *show_saved=False*, *max_attr_chars=50*)

Produces Graphviz representation of PyTorch autograd graph.

If a node represents a backward function, it is gray. Otherwise, the node represents a tensor and is either blue, orange, or green:

- **Blue**
  Reachable leaf tensors that requires grad (tensors whose `grad` fields will be populated during `backward()`).
- **Orange**
  Saved tensors of custom autograd functions as well as those saved by built-in backward nodes.
- **Green**
  Tensor passed in as outputs.
- **Dark green**
  If any output is a view, we represent its base tensor with a dark green node.

Parameters

- **var** (`Union[Tensor, Sequence[Tensor]]`) – Output tensor.
- **params** (`Optional[Union[Mapping[str, Tensor], ModuleState, Generator, Iterable[Union[Mapping[str, Tensor], ModuleState, Generator]]]]`) – ([dict of (name, tensor) or state_dict]) Parameters to add names to node that requires grad.
- **show_attrs** (`bool`) – Whether to display non-tensor attributes of backward nodes (Requires PyTorch version >= 1.9)

- **show_saved** (bool) – Whether to display saved tensor nodes that are not by custom autograd functions. Saved tensor nodes for custom functions, if present, are always displayed. (Requires PyTorch version >= 1.9)
- **max_attr_chars** (int) – If show_attrs is True, sets max number of characters to display for any given attribute.

Return type
Digraph

CHAPTER

TWO

THE TEAM

TorchOpt is a work by

- Jie Ren (JieRen98)
- Xidong Feng (waterhorse1)
- Bo Liu (Benjamin-eecs)
- Xuehai Pan (XuehaiPan)
- Luo Mai (luomai)
- Yaodong Yang (PKU-YYang).

CHAPTER

THREE

SUPPORT

If you are having issues, please let us know by filing an issue on our issue tracker.

CHAPTER

FOUR

CHANGELOG

See CHANGELOG.md.

CHAPTER

FIVE

LICENSE

TorchOpt is licensed under the Apache 2.0 License.

CHAPTER

SIX

CITING

If you find TorchOpt useful, please cite it in your publications.

```bibtex
@article{torchopt,
  title   = {TorchOpt: An Efficient Library for Differentiable Optimization},
  author  = {Ren, Jie and Feng, Xidong and Liu, Bo and Pan, Xuehai and Fu, Yao and Mai,␣
↪Luo and Yang, Yaodong},
  journal = {arXiv preprint arXiv:2211.06934},
  year    = {2022}
}
```

6.1 Indices and tables

- genindex

BIBLIOGRAPHY

[FAL17] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of Proceedings of Machine Learning Research, 1126–1135. PMLR, 2017. URL: http://proceedings.mlr.press/v70/finn17a.html.

INDEX

Symbols

A

C

E

F

I

L

M

N

O

P

R

S

Z