TorchDEQ

Zhengyang Geng

Oct 31, 2023

CONTENTS

CHAPTER

ONE

GET STARTED

1.1 Installation

- Through pip.

```
pip install torchdeq
```

- Through conda.

```
conda install torchdeq
```

- From source.

```
git clone https://github.com/locuslab/torchdeq.git && cd torchdeq
pip install -e .
```

1.2 Quick Start

- Automatic arg parser decorator. You can call this function to add commonly used DEQ args to your program. See the explanation for args here!

```
add_deq_args(parser)
```

- Automatic DEQ definition. Call `get_deq` to get your DEQ module! It's highly decoupled implementation agnostic to your model design!

```
self.deq = get_deq(args)
```

- Automatic normalization for DEQ. You now do not need to add normalization manually to each weight in your DEQ module!

```
apply_norm(self.deq_layers)
```

- Easy DEQ forward. Even for a multi-equilibria system, you can call your DEQ in a single line!

```
# Assume f is a functioin of three variable a, b, c.
def fn(a, b, c):
    # Do something here...
    # Having the same input and output tensor shapes.
    return a, b, c
```

(continues on next page)

(continued from previous page)

```
# A callable object (`fn` here) that defines your fixed point system.
# `fn` can be a functor defined in your Pytorch forward function.
# A functor can take your input injection from the local variables.
# You can also pass a Pytorch Module into the DEQ class.
z_out, info = self.deq(fn, (a0, b0, c0))
```

- Automatic DEQ backward. Gradients (both exact and inexact grad) are tracked automatically! The DEQ class can sample the convergence trajectory for addition operation/supervision. Just post-process `z_out` as you want!

1.3 Sample Code

```
import argparse

import torch

from torchdeq import get_deq, apply_norm, reset_norm
from torchdeq.utils import add_deq_args

from .layers import Injection, DEQFunc, Decoder

class DEQDemo(torch.nn.Module):
    def __init__(self, args):
        super().__init__()
        self.deq_func = DEQFunc(args)
        apply_norm(self.deq_func, args=args)
        self.deq = get_deq(args)

    def forward(self, x, z0):
        reset_norm(self.deq_func)
        f = lambda z: self.deq_func(z, x)
        return self.deq(f, z0)

def train(args, inj, deq, decoder, loader, loss, opt):
    for x, y in loader:
        z0 = torch.randn(args.z_shape)
        z_out, info = deq(inj(x), z0)
        l = loss(decoder(z_out[-1]), y)
        l.backward()
        opt.step()
        logger.info(f'Loss: {l.item()}, '
          +f'Rel: {info['rel_lowest'].mean().item()}'
          +f'Abs: {info['abs_lowest'].mean().item()}')

'''Add other arguments.'''
parser = argparse.ArgumentParser()
add_deq_args(parser)
args = parser.parse_args()

inj = Injection(args)
```

(continues on next page)

(continued from previous page)

```
deq = DEQDemo(args)
decoder = Decoder(args)

''' Set up loader, logger, loss, opt, etc as in standard PyTorch. '''
train(args, inj, deq, decoder, loader, loss, opt)
```

CHAPTER

TWO

TORCHDEQ.CORE

The DEQ models are a class of implicit models that solve for fixed points to make predictions. This module provides the core classes and functions for implementing Deep Equilibrium (DEQ) models in PyTorch.

The main classes in this module are *DEQBase*, *DEQIndexing*, and *DEQSliced*. *DEQBase* is the base class for DEQ models, and *DEQIndexing* and *DEQSliced* are two specific implementations of DEQ models that use different strategies for applying gradients during training.

The module also provides utility functions for creating and manipulating DEQ models, such as *get_deq* for creating a DEQ model based on command line arguments, *register_deq* for registering a new DEQ model class, and *reset_deq* for resetting the normalization and dropout layers of a DEQ model.

Example

To create a DEQ model, you can use the *get_deq* function:

```
>>> deq = get_deq(args)
```

To reset the normalization and dropout layers of a DEQ model, you can use the *reset_deq* function:

```
>>> deq_layer = DEQLayer(args)          # A Pytorch Module used in the f of z* = f(z*, x).
>>> reset_deq(deq_layer)
```

2.1 Core Function

`torchdeq.core.`**`get_deq`**(*args=None*, ***kwargs*)

Factory function to generate an instance of a DEQ model based on the command line arguments.

This function returns an instance of a DEQ model class based on the DEQ computational core specified in the command line arguments `args.core`. For example, `--core indexing` for DEQIndexing, `--core sliced` for DEQSliced, etc.

DEQIndexing and DEQSliced build different computational graphs in training but keep the same for test.

For *DEQIndexing*, it defines a computational graph with tracked gradients by indexing the internal solver states and applying the gradient function to the sampled states. This is equivalent to attaching the gradient function aside the full solver computational graph. The maximum number of DEQ function calls is defined by `args.f_max_iter`.

For *DEQSliced*, it slices the full solver steps into several smaller graphs (w/o grad). The gradient function will be applied to the returned state of each subgraph. Then a new fixed point solver will resume from the output

of the gradient function. This is equivalent to inserting the gradient function into the full solver computational graph. The maximum number of DEQ function calls is defined by, for example, `args.f_max_iter + args.n_states * args.grad`.

Parameters

- **args** (*Union[argparse.Namespace, dict, DEQConfig, Any]*) – Configuration specifying the config of the DEQ model. Default None. This can be an instance of argparse.Namespace, a dictionary, or an instance of DEQConfig. Unknown config will be processed using *get_attr* function.
- ****kwargs** – Additional keyword arguments to update the config.

Returns
DEQ module that defines the computational graph from the specified config.

Return type
DEQBase (torch.nn.Module)

Example

To instantiate a DEQ module, you can directly pass keyword arguments to this function:

```
>>> deq = get_deq(core='sliced')
```

Alternatively, if you're using a config system like *argparse*, you can pass the parsed config as a single object:

```
>>> args = argparse.Namespace(core='sliced')
>>> deq = get_deq(args)
```

torchdeq.core.**reset_deq**(*model*)

Resets the normalization and dropout layers of the given DEQ model (usually before each training iteration).

Parameters
model (*torch.nn.Module*) – The DEQ model to reset.

Example

```
>>> deq_layer = DEQLayer(args)          # A Pytorch Module used in the f of z* =␣
↪f(z*, x).
>>> reset_deq(deq_layer)
```

torchdeq.core.**register_deq**(*deq_type*, *core*)

Registers a user-defined DEQ class for the get_deq function.

This method adds a new entry to the DEQ class dict with the key as the specified DEQ type and the value as the DEQ class.

Parameters

- **deq_type** (*str*) – The type of DEQ model to register. This will be used as the key in the DEQ class dict.
- **core** (*type*) – The class defining the DEQ model. This will be used as the value in the DEQ class dict.

Example

```
>>> register_deq('custom', CustomDEQ)
```

2.2 DEQ Class

class torchdeq.core.**DEQBase**(*args=None*, *f_solver='fixed_point_iter'*, *b_solver='fixed_point_iter'*, *no_stat=None*, *f_max_iter=40*, *b_max_iter=40*, *f_tol=0.001*, *b_tol=1e-06*, *f_stop_mode='abs'*, *b_stop_mode='abs'*, *eval_factor=1.0*, *eval_f_max_iter=0*, ***kwargs*)

Base class for Deep Equilibrium (DEQ) model.

This class is not intended to be directly instantiated as the actual DEQ module. Instead, you should create an instance of a subclass of this class.

If you are looking to implement a new computational graph for DEQ models, you can inherit from this class. This allows you to leverage other components in the library in your implementation.

Parameters

- **args** (*Union[argparse.Namespace, dict, DEQConfig, Any], optional*) – Configuration for the DEQ model. This can be an instance of argparse.Namespace, a dictionary, or an instance of DEQConfig. Unknown config will be processed using *get_attr* function. Priority: `args` > `norm_kwargs`. Default None.
- **f_solver** (*str, optional*) – The forward solver function. Default solver is `'fixed_point_iter'`.
- **b_solver** (*str, optional*) – The backward solver function. Default solver is `'fixed_point_iter'`.
- **no_stat** (*bool, optional*) – Skips the solver stats computation if True. Default None.
- **f_max_iter** (*int, optional*) – Maximum number of iterations (NFE) for the forward solver. Default 40.
- **b_max_iter** (*int, optional*) – Maximum number of iterations (NFE) for the backward solver. Default 40.
- **f_tol** (*float, optional*) – The forward pass solver stopping criterion. Default 1e-3.
- **b_tol** (*float, optional*) – The backward pass solver stopping criterion. Default 1e-6.
- **f_stop_mode** (*str, optional*) – The forward pass fixed-point convergence stop mode. Default `'abs'`.
- **b_stop_mode** (*str, optional*) – The backward pass fixed-point convergence stop mode. Default `'abs'`.
- **eval_factor** (*int, optional*) – The max iteration for the forward pass at test time, calculated as `f_max_iter * eval_factor`. Default 1.0.
- **eval_f_max_iter** (*int, optional*) – The max iteration for the forward pass at test time. Overwrite `eval_factor` by an exact number.
- ****kwargs** – Additional keyword arguments to update the configuration.

forward(*func*, *z_init*, *solver_kwargs=None*, *sradius_mode=False*, *backward_writer=None*, ***kwargs*)

Defines the computation graph and gradients of DEQ. Must be overridden in subclasses.

Parameters

- **func** (*callable*) – The DEQ function.
- **z_init** (*torch.Tensor*) – Initial tensor for fixed point solver.
- **solver_kwargs** (*dict, optional*) – Additional arguments for the solver used in this forward pass. These arguments will overwrite the default solver arguments. Refer to the documentation of the specific solver for the list of accepted arguments. Default None.
- **sradius_mode** (*bool, optional*) – If True, computes the spectral radius in validation and adds 'sradius' to the `info` dictionary. Default False.
- **backward_writer** (*callable, optional*) – Callable function to monitor the backward pass. It should accept the solver statistics dictionary as input. Default None.

Raises

NotImplementedError – If the method is not overridden.

class torchdeq.core.**DEQIndexing**(*args=None*, *ift=False*, *hook_ift=False*, *grad=1*, *tau=1.0*, *sup_gap=-1*, *sup_loc=None*, *n_states=1*, *indexing=None*, ***kwargs*)

DEQ computational graph that samples fixed point states at specific indices.

For *DEQIndexing*, it defines a computational graph with tracked gradients by indexing the internal solver states and applying the gradient function to the sampled states. This is equivalent to attaching the gradient function aside the full solver computational graph. The maximum number of DEQ function calls is defined by `args.f_max_iter`.

Parameters

- **args** (*Union[argparse.Namespace, dict, DEQConfig, Any], optional*) – Configuration for the DEQ model. This can be an instance of argparse.Namespace, a dictionary, or an instance of DEQConfig. Unknown config will be processed using *get_attr* function. Priority: `args` > `norm_kwargs`. Default None.
- **f_solver** (*str, optional*) – The forward solver function. Default `'fixed_point_iter'`.
- **b_solver** (*str, optional*) – The backward solver function. Default `'fixed_point_iter'`.
- **no_stat** (*bool, optional*) – Skips the solver stats computation if True. Default None.
- **f_max_iter** (*int, optional*) – Maximum number of iterations (NFE) for the forward solver. Default 40.
- **b_max_iter** (*int, optional*) – Maximum number of iterations (NFE) for the backward solver. Default 40.
- **f_tol** (*float, optional*) – The forward pass solver stopping criterion. Default 1e-3.
- **b_tol** (*float, optional*) – The backward pass solver stopping criterion. Default 1e-6.
- **f_stop_mode** (*str, optional*) – The forward pass fixed-point convergence stop mode. Default `'abs'`.
- **b_stop_mode** (*str, optional*) – The backward pass fixed-point convergence stop mode. Default `'abs'`.
- **eval_factor** (*int, optional*) – The max iteration for the forward pass at test time, calculated as `f_max_iter * eval_factor`. Default 1.0.

- **eval_f_max_iter** (*int, optional*) – The max iteration for the forward pass at test time. Overwrite `eval_factor` by an exact number.
- **ift** (*bool, optional*) – If true, enable Implicit Differentiation. IFT=Implicit Function Theorem. Default False.
- **hook_ift** (*bool, optional*) – If true, enable a Pytorch backward hook implementation of IFT. Furthure reduces memory usage but may affect stability. Default False.
- **grad** (*Union[int, list[int], tuple[int]], optional*) – Specifies the steps of PhantomGrad. It allows for using multiple values to represent different gradient steps in the sampled trajectory states. Default 1.
- **tau** (*float, optional*) – Damping factor for PhantomGrad. Default 1.0.
- **sup_gap** (*int, optional*) – The gap for uniformly sampling trajectories from PhantomGrad. Sample every `sup_gap` states if `sup_gap > 0`. Default -1.
- **sup_loc** (*list[int], optional*) – Specifies trajectory steps or locations in PhantomGrad from which to sample. Default None.
- **n_states** (*int, optional*) – Uniformly samples trajectory states from the solver. The backward passes of sampled states will be automactically tracked. IFT will be applied to the best fixed-point estimation when `ift=True`, while internal states are tracked by PhantomGrad. Default 1. By default, only the best fixed point estimation will be returned.
- **indexing** (*int, optional*) – Samples specific trajectory states at the given steps in `indexing` from the solver. Similar to `n_states` but more flexible. Default None.
- ****kwargs** – Additional keyword arguments to update the configuration.

`arg_indexing`

Define gradient functions through the backward factory.

forward(*func*, *z_init*, *solver_kwargs=None*, *sradius_mode=False*, *backward_writer=None*, ***kwargs*)

Defines the computation graph and gradients of DEQ.

This method carries out the forward pass computation for the DEQ model, by solving for the fixed point. During training, it also keeps track of the trajectory of the solution. In inference mode, it returns the final fixed point.

Parameters

- **func** (*callable*) – The DEQ function.
- **z_init** (*torch.Tensor*) – Initial tensor for fixed point solver.
- **solver_kwargs** (*dict, optional*) – Additional arguments for the solver used in this forward pass. These arguments will overwrite the default solver arguments. Refer to the documentation of the specific solver for the list of accepted arguments. Default None.
- **sradius_mode** (*bool, optional*) – If True, computes the spectral radius in validation and adds `'sradius'` to the `info` dictionary. Default False.
- **backward_writer** (*callable, optional*) – Callable function to monitor the backward pass. It should accept the solver statistics dictionary as input. Default None.

Returns

a tuple containing the following.

- **list[torch.Tensor]:**

During training, returns the sampled fixed point trajectory (tracked gradients) according to `n_states` or `indexing`.

During inference, returns a list containing the fixed point solution only.

- **dict[str, torch.Tensor]:**
  A dict containing solver statistics in a batch. Please see *torchdeq.solver.stat.SolverStat* for more details.

Return type
tuple[list[torch.Tensor], dict[str, torch.Tensor]]

class torchdeq.core.**DEQSliced**(*args=None*, *ift=False*, *hook_ift=False*, *grad=1*, *tau=1.0*, *sup_gap=-1*, *sup_loc=None*, *n_states=1*, *indexing=None*, ***kwargs*)

DEQ computational graph that slices the full solver trajectory to apply gradients.

For *DEQSliced*, it slices the full solver steps into several smaller graphs (w/o grad). The gradient function will be applied to the returned state of each subgraph. Then a new fixed point solver will resume from the output of the gradient function. This is equivalent to inserting the gradient function into the full solver computational graph. The maximum number of DEQ function calls is defined by, for example, `args.f_max_iter + args.n_states * args.grad`.

Parameters

- **args** (*Union[argparse.Namespace, dict, DEQConfig, Any], optional*) – Configuration for the DEQ model. This can be an instance of argparse.Namespace, a dictionary, or an instance of DEQConfig. Unknown config will be processed using *get_attr* function. Priority: `args` > `norm_kwargs`. Default None.
- **f_solver** (*str, optional*) – The forward solver function. Default `'fixed_point_iter'`.
- **b_solver** (*str, optional*) – The backward solver function. Default `'fixed_point_iter'`.
- **no_stat** (*bool, optional*) – Skips the solver stats computation if True. Default None.
- **f_max_iter** (*int, optional*) – Maximum number of iterations (NFE) for the forward solver. Default 40.
- **b_max_iter** (*int, optional*) – Maximum number of iterations (NFE) for the backward solver. Default 40.
- **f_tol** (*float, optional*) – The forward pass solver stopping criterion. Default 1e-3.
- **b_tol** (*float, optional*) – The backward pass solver stopping criterion. Default 1e-6.
- **f_stop_mode** (*str, optional*) – The forward pass fixed-point convergence stop mode. Default `'abs'`.
- **b_stop_mode** (*str, optional*) – The backward pass fixed-point convergence stop mode. Default `'abs'`.
- **eval_factor** (*int, optional*) – The max iteration for the forward pass at test time, calculated as `f_max_iter * eval_factor`. Default 1.0.
- **eval_f_max_iter** (*int, optional*) – The max iteration for the forward pass at test time. Overwrite `eval_factor` by an exact number.
- **ift** (*bool, optional*) – If true, enable Implicit Differentiation. IFT=Implicit Function Theorem. Default False.
- **hook_ift** (*bool, optional*) – If true, enable a Pytorch backward hook implementation of IFT. Furthure reduces memory usage but may affect stability. Default False.

- **grad** (*Union[int, list[int], tuple[int]], optional*) – Specifies the steps of PhantomGrad. It allows for using multiple values to represent different gradient steps in the sampled trajectory states. Default 1.
- **tau** (*float, optional*) – Damping factor for PhantomGrad. Default 1.0.
- **sup_gap** (*int, optional*) – The gap for uniformly sampling trajectories from PhantomGrad. Sample every `sup_gap` states if `sup_gap > 0`. Default -1.
- **sup_loc** (*list[int], optional*) – Specifies trajectory steps or locations in PhantomGrad from which to sample. Default None.
- **n_states** (*int, optional*) – Uniformly samples trajectory states from the solver. The backward passes of sampled states will be automactically tracked. IFT will be applied to the best fixed-point estimation when `ift=True`, while internal states are tracked by PhantomGrad. Default 1. By default, only the best fixed point estimation will be returned.
- **indexing** (*int, optional*) – Samples specific trajectory states at the given steps in `indexing` from the solver. Similar to `n_states` but more flexible. Default None.
- ****kwargs** – Additional keyword arguments to update the configuration.

arg_indexing

Define gradient functions through the backward factory.

forward(*func*, *z_star*, *solver_kwargs=None*, *sradius_mode=False*, *backward_writer=None*, ***kwargs*)

Defines the computation graph and gradients of DEQ.

Parameters

- **func** (*callable*) – The DEQ function.
- **z_init** (*torch.Tensor*) – Initial tensor for fixed point solver.
- **solver_kwargs** (*dict, optional*) – Additional arguments for the solver used in this forward pass. These arguments will overwrite the default solver arguments. Refer to the documentation of the specific solver for the list of accepted arguments. Default None.
- **sradius_mode** (*bool, optional*) – If True, computes the spectral radius in validation and adds `'sradius'` to the `info` dictionary. Default False.
- **backward_writer** (*callable, optional*) – Callable function to monitor the backward pass. It should accept the solver statistics dictionary as input. Default None.

Returns

a tuple containing the following.

- **list[torch.Tensor]:**

  During training, returns the sampled fixed point trajectory (tracked gradients) according to `n_states` or `indexing`.

  During inference, returns a list containing the fixed point solution only.

- **dict[str, torch.Tensor]:**
  A dict containing solver statistics in a batch. Please see *torchdeq.solver.stat.SolverStat* for more details.

Return type

tuple[list[torch.Tensor], dict[str, torch.Tensor]]

CHAPTER

THREE

TORCHDEQ.GRAD

The *torchdeq.grad* module offers a factory function, *backward_factory*, which is designed to facilitate the customization of various differentiation methods during the backward pass.

This function is integral to the construction of the backward computational graph in the DEQ class, as it is invoked multiple times to generate gradient functors.

While the backward_factory function is a powerful tool, it is generally not recommended for direct use outside of the library. Instead, users should primarily interact with the DEQ class via the *torch.core* entry point for most DEQ computations. This approach ensures the appropriate and efficient use of the library's features.

`torchdeq.grad.`**`backward_factory`**(*grad_type='ift'*, *hook_ift=False*, *b_solver=None*, *b_solver_kwargs={}*, *sup_gap=-1*, *sup_loc=None*, *tau=1.0*, ***grad_factory_kwargs*)

Factory for the backward pass of implicit deep learning, e.g., DEQ (implicit models), Hamburger (optimization layers), etc. This function implements various gradients like Implicit Differentiation (IFT), 1-step Grad and Phantom Grad.

Implicit Differentiation:

[2018-ICML] Reviving and Improving Recurrent Back-Propagation

[2019-NeurIPS] Deep Equilibrium Models

[2019-NeurIPS] Meta-Learning with Implicit Gradients

...

1-step Grad & Higher-order Grad:

[2021-ICLR] Is Attention Better Than Matrix Decomposition?

[2022-AAAI] JFB: Jacobian-Free Backpropagation for Implicit Networks

[2021-NeurIPS] On Training Implicit Models

...

Parameters

- **`grad_type`** (*`str, int, optional`*) – Gradient type to use. grad_type should be `'ift'` for IFT or an int for PhantomGrad. Default `'ift'`. Set to `'ift'` to enable the implicit differentiation (IFT) mode. When passing a number `k` to this function, it runs UPG with steps `k` and damping factor `tau`.
- **`hook_ift`** (*`bool, optional`*) – Set to `True` to enable an $\Omega(1)$ memory (w.r.t. activations) implementation using the Pytorch hook for IFT.

  Set to `False` to enable the $\Omega(2)$ memory implementation using `torch.autograd.Function` to avoid the (potential) segment fault in older PyTorch versions.

Note that the `torch.autograd.Function` implementation is more stable than this hook in numerics and execution, even though they should be conceptually the same. For PyTorch version < 1.7.1 on some machines, this $\Omega(1)$ hook seems to trigger a segment fault after some training steps. This issue is not caused by TorchDEQ but rather due to the hook.remove() call and some interactions between Python and PyTorch. The `torch.autograd.Function` implementation also introduces slightly better numerical stability when the forward solver introduces some fixed point errors.

Default `False`.

- **b_solver** (*str, optional*) – Solver for the IFT backward pass. Default None. Supported solvers: `'anderson'`, `'broyden'`, `'fixed_point_iter'`, `'simple_fixed_point_iter'`.
- **b_solver_kwargs** (*dict, optional*) – Collection of backward solver kwargs, e.g., max_iter (int, optional), max steps for the backward solver, stop_mode (str, optional), criterion for convergence, etc. See torchdeq.solver for all kwargs.
- **sup_gap** (*int, optional*) – The gap for uniformly sampling trajectories from PhantomGrad. Sample every `sup_gap` states if `sup_gap > 0`. Default -1.
- **sup_loc** (*list[int], optional*) – Specifies trajectory steps or locations in PhantomGrad from which to sample. Default None.
- **tau** (*float, optional*) – Damping factor for PhantomGrad. Default 1.0. 0.5-0.7 is recommended for MDEQ. 1.0 for DEQ flow. For DEQ flow, the gating function in GRU naturally produces adaptive tau values.
- **grad_factory_kwargs** – Extra arguments are ignored.

Returns

A gradient functor for implicit deep learning. The function takes trainer, func and z_pred as arguments and returns a list of tensors with the gradient information.

Args:

trainer (torch.nn.Module):
the module that employs implicit deep learning.

func (type):
function that defines the f in $z = f(z)$.

z_pred (torch.Tensor):
latent state to run the backward pass.

writer (callable, optional):
Callable function to monitor the backward pass. It should accept the solver statistics dictionary as input. Default None.

Returns:

list[torch.Tensor]:
a list of tensors that tracks the gradient info. These tensors can be directly applied to downstream networks, while all the gradient info will be automatically tracked in the backward pass.

Return type
callable

CHAPTER

FOUR

TORCHDEQ.SOLVER

The *torchdeq.solver* module provides a set of solvers for finding fixed points in Deep Equilibrium Models (DEQs). These solvers are used to iteratively refine the predictions of a DEQ model until they reach a stable state, or "equilibrium".

This module includes implementations of several popular fixed-point solvers, including Anderson acceleration (*anderson_solver*), Broyden's method (*broyden_solver*), and fixed-point iteration (*fixed_point_iter*). It also provides a faster version of fixed-point iteration (*simple_fixed_point_iter*) that omits convergence monitoring for speed improvements.

The *get_solver* function allows users to retrieve a specific solver by its key, and the *register_solver* function allows users to add their own custom solvers to the module.

Example

To retrieve a solver, call this *get_solver* function:

```
>>> solver = get_solver('anderson')
```

To register a user-developed solver, call this *register_solver* function:

```
>>> register_solver('newton', newton_solver)
```

4.1 Solver Function

torchdeq.solver.**get_solver**(*key*)

Retrieves a fixed point solver from the registered solvers by its key.

Supported solvers: `'anderson'`, `'broyden'`, `'fixed_point_iter'`, `'simple_fixed_point_iter'`.

Parameters

key (*str*) – The key of the solver to retrieve. This should match one of the keys used to register a solver.

Returns

The solver function associated with the provided key.

Return type

callable

Raises

AssertionError – If the key does not match any of the registered solvers.

Example

```
>>> solver = get_solver('anderson')
```

torchdeq.solver.**register_solver**(*solver_type*, *solver*)

Registers a user-defined fixed point solver. This solver can be designated using args.f_solver and args.b_solver.

This method adds a new entry to the solver dict with the key as the specified `solver_type` and the value as the `solver`.

Parameters

- **solver_type** (*str*) – The type of solver to register. This will be used as the key in the solver dict.
- **solver_class** (*callable*) – The solver function. This will be used as the value in the solver dict.

Example

```
>>> register_solver('newton', newton_solver)
```

4.2 Solver

torchdeq.solver.fp_iter.**fixed_point_iter**(*func*, *x0*, *max_iter=50*, *tol=0.001*, *stop_mode='abs'*, *indexing=None*, *tau=1.0*, *return_final=False*, ***kwargs*)

Implements the fixed-point iteration solver for solving a system of nonlinear equations.

Parameters

- **func** (*callable*) – The function for which we seek a fixed point.
- **x0** (*torch.Tensor*) – The initial guess for the root.
- **max_iter** (*int, optional*) – The maximum number of iterations. Default: 50.
- **tol** (*float, optional*) – The convergence criterion. Default: 1e-3.
- **stop_mode** (*str, optional*) – The stopping criterion. Can be either 'abs' or 'rel'. Default: 'abs'.
- **indexing** (*list, optional*) – List of iteration indices at which to store the solution. Default: None.
- **tau** (*float, optional*) – Damping factor. It is used to control the step size in the direction of the solution. Default: 1.0.
- **return_final** (*bool, optional*) – If True, run all steps and returns the final solution instead of the one with smallest residual. Default: False.
- **kwargs** (*dict, optional*) – Extra arguments are ignored.

Returns

a tuple containing the following.

- torch.Tensor: Fixed point solution.
- list[torch.Tensor]: List of the solutions at the specified iteration indices.

- **dict[str, torch.Tensor]:**
  A dict containing solver statistics in a batch. Please see *torchdeq.solver.stat.SolverStat* for more details.

Return type
tuple[torch.Tensor, list[torch.Tensor], dict[str, torch.Tensor]]

Examples

```
>>> f = lambda z: torch.cos(z)                    # Function for which we seek a␣
↪fixed point
>>> z0 = torch.tensor(0.0)                        # Initial estimate
>>> z_star, _, _ = fixed_point_iter(f, z0)        # Run Fixed Point iterations.
>>> print((z_star - f(z_star)).norm(p=1))         # Print the numerical error
```

torchdeq.solver.fp_iter.**simple_fixed_point_iter**(*func*, *x0*, *max_iter=50*, *tau=1.0*, *indexing=None*, ***kwargs*)

Implements a simplified fixed-point solver for solving a system of nonlinear equations.

Speeds up by removing statistics monitoring.

Parameters

- **func** (*callable*) – The function for which the fixed point is to be computed.
- **x0** (*torch.Tensor*) – The initial guess for the fixed point.
- **max_iter** (*int, optional*) – The maximum number of iterations. Default: 50.
- **tau** (*float, optional*) – Damping factor to control the step size in the solution direction. Default: 1.0.
- **indexing** (*list, optional*) – List of iteration indices at which to store the solution. Default: None.
- **kwargs** (*dict, optional*) – Extra arguments are ignored.

Returns

a tuple containing the following.

- torch.Tensor: The approximate solution.
- list[torch.Tensor]: List of the solutions at the specified iteration indices.
- **dict[str, torch.Tensor]:**
  A dummy dict for solver statistics. All values are initialized as -1 of tensor shape (1, 1).

Return type
tuple[torch.Tensor, list[torch.Tensor], dict[str, torch.Tensor]]

Examples

```
>>> f = lambda z: torch.cos(z)                          # Function for which we seek a␣
↪fixed point
>>> z0 = torch.tensor(0.0)                              # Initial estimate
>>> z_star, _, _ = simple_fixed_point_iter(f, z0)       # Run fixed point iterations
>>> print((z_star - f(z_star)).norm(p=1))               # Print the numerical error
```

torchdeq.solver.anderson.**anderson_solver**(*func*, *x0*, *max_iter=50*, *tol=0.001*, *stop_mode='abs'*, *indexing=None*, *m=6*, *lam=0.0001*, *tau=1.0*, *return_final=False*, ***kwargs*)

Implements the Anderson acceleration for fixed-point iteration.

Anderson acceleration is a method that can accelerate the convergence of fixed-point iterations. It improves the rate of convergence by generating a sequence that converges to the fixed point faster than the original sequence.

Parameters

- **func** (*callable*) – The function for which we seek a fixed point.
- **x0** (*torch.Tensor*) – Initial estimate for the fixed point.
- **max_iter** (*int, optional*) – Maximum number of iterations. Default: 50.
- **tol** (*float, optional*) – Tolerance for stopping criteria. Default: 1e-3.
- **stop_mode** (*str, optional*) – Stopping criterion. Can be 'abs' for absolute or 'rel' for relative. Default: 'abs'.
- **indexing** (*None or list, optional*) – Indices for which to store and return solutions. If None, solutions are not stored. Default: None.
- **m** (*int, optional*) – Maximum number of stored residuals in Anderson mixing. Default: 6.
- **lam** (*float, optional*) – Regularization parameter in Anderson mixing. Default: 1e-4.
- **tau** (*float, optional*) – Damping factor. It is used to control the step size in the direction of the solution. Default: 1.0.
- **return_final** (*bool, optional*) – If True, returns the final solution instead of the one with smallest residual. Default: False.
- **kwargs** (*dict, optional*) – Extra arguments are ignored.

Returns

a tuple containing the following.

- torch.Tensor: Fixed point solution.
- list[torch.Tensor]: List of the solutions at the specified iteration indices.
- **dict[str, torch.Tensor]:**
  A dict containing solver statistics in a batch. Please see *torchdeq.solver.stat.SolverStat* for more details.

Return type

tuple[torch.Tensor, list[torch.Tensor], dict[str, torch.Tensor]]

Examples

```
>>> f = lambda z: 0.5 * (z + 2 / z)                    # Function for which we seek a␣
↪fixed point
>>> z0 = torch.tensor(1.0)                             # Initial estimate
>>> z_star, _, _ = anderson_solver(f, z0)              # Run Anderson Acceleration
>>> print((z_star - f(z_star)).norm(p=1))              # Print the numerical error
```

torchdeq.solver.broyden.**broyden_solver**(*func*, *x0*, *max_iter=50*, *tol=0.001*, *stop_mode='abs'*, *indexing=None*, *LBFGS_thres=None*, *ls=False*, *return_final=False*, ***kwargs*)

Implements the Broyden's method for solving a system of nonlinear equations.

Parameters

- **func** (*callable*) – The function for which we seek a fixed point.
- **x0** (*torch.Tensor*) – The initial guess for the root.
- **max_iter** (*int, optional*) – The maximum number of iterations. Default: 50.
- **tol** (*float, optional*) – The convergence criterion. Default: 1e-3.
- **stop_mode** (*str, optional*) – The stopping criterion. Can be either 'abs' or 'rel'. Default: 'abs'.
- **indexing** (*list, optional*) – List of iteration indices at which to store the solution. Default: None.
- **LBFGS_thres** (*int, optional*) – The max_iter for the limited memory BFGS method. None for storing all. Default: None.
- **ls** (*bool, optional*) – If True, perform a line search at each step. Default: False.
- **return_final** (*bool, optional*) – If True, returns the final solution instead of the one with smallest residual. Default: False.
- **kwargs** (*dict, optional*) – Extra arguments are ignored.

Returns

a tuple containing the following.

- torch.Tensor: Fixed point solution.
- list[torch.Tensor]: List of the solutions at the specified iteration indices.
- **dict[str, torch.Tensor]:**
  A dict containing solver statistics in a batch. Please see *torchdeq.solver.stat.SolverStat* for more details.

Return type

tuple[torch.Tensor, list[torch.Tensor], dict[str, torch.Tensor]]

Examples

```
>>> f = lambda z: 0.5 * (z + 2 / z)                    # Function for which we seek a␣
↪fixed point
>>> z0 = torch.tensor(1.0)                             # Initial estimate
>>> z_star, _, _ = broyden_solver(f, z0)               # Run the Broyden's method
>>> print((z_star - f(z_star)).norm(p=1))              # Print the numerical error
```

4.3 Solver Stat

class torchdeq.solver.stat.**SolverStat**(**args*, ***kwargs*)

A class for storing solver statistics.

This class is a subclass of dict, which allows users to query the solver statistics as dictionary keys.

Valid Keys:

- **'abs_lowest':**
  The lowest absolute fixed point errors achieved, i.e. $\|z - f(z)\|$. torch.Tensor of shape $(B,)$.
- **'rel_lowest':**
  The lowest relative fixed point errors achieved, i.e., $\|z - f(z)\|/\|f(z)\|$. torch.Tensor of shape $(B,)$.
- **'abs_trace':**
  The absolute fixed point errors achieved along the solver steps. torch.Tensor of shape (B, N), where N is the solver step consumed.
- **'rel_trace':**
  The relative fixed point errors achieved along the solver steps. torch.Tensor of shape (B, N), where N is the solver step consumed.
- **'nstep':**
  The number of step when the fixed point errors were achieved. torch.Tensor of shape $(B,)$.
- **'sradius':**
  Optional. The largest (abs.) eigenvalue estimated by power method. Available in the eval mode when `sradius_mode` set to `True`. torch.Tensor of shape $(B,)$.

CHAPTER

FIVE

TORCHDEQ.NORM

The *torchdeq.norm* module provides a set of tools for managing normalization in Deep Equilibrium Models (DEQs). It includes factory functions for applying, resetting, and removing normalization, as well as for registering new normalization types and modules.

The module also provides classes for specific types of normalization, such as *WeightNorm* and *SpectralNorm*.

Example

To apply normalization to a model, call this *apply_norm* function:

```
>>> apply_norm(model, 'weight_norm', filter_out=['embedding'])
```

To reset the all normalization within a DEQ model, call this *reset_norm* function:

```
>>> reset_norm(model)
```

To remove the normalization of a DEQ model, call *remove_norm* function:

```
>>> remove_norm(model)
```

To register a user-defined normalization type, call *register_norm* function:

```
>>> register_norm('custom_norm', CustomNorm)
```

To register a new module for a user-define normalization, call *register_norm_module* function:

```
>>> register_norm_module(Conv2d, 'custom_norm', 'weight', 0)
```

5.1 Norm Function

torchdeq.norm.**apply_norm**(*model*, *norm_type='weight_norm'*, *prefix_filter_out=None*, *filter_out=None*, *args=None*, ***norm_kwargs*)

Auto applies normalization to all weights of a given layer based on the `norm_type`.

The currently supported normalizations include `'weight_norm'`, `'spectral_norm'`, and `'none'` (No Norm applied). Skip the weights whose name contains any string of `filter_out` or starts with any of `prefix_filter_out`.

Parameters

- **model** (*torch.nn.Module*) – Model to apply normalization.

- **norm_type** (*str, optional*) – Type of normalization to be applied. Default is `'weight_norm'`.
- **prefix_filter_out** (*list or str, optional*) – List of module weights prefixes to skip out when applying normalization. Default is None.
- **filter_out** (*list or str, optional*) – List of module weights names to skip out when applying normalization. Default is None.
- **args** (*Union[argparse.Namespace, dict, DEQConfig, Any]*) – Configuration for the DEQ model. This can be an instance of argparse.Namespace, a dictionary, or an instance of DEQConfig. Unknown config will be processed using `get_attr` function. Priority: `args` > `norm_kwargs`. Default is None.
- **norm_kwargs** – Keyword arguments for the normalization layer.

Raises

AssertionError – If the `norm_type` is not registered.

Example

```
>>> apply_norm(model, 'weight_norm', filter_out=['embedding'])
```

torchdeq.norm.**reset_norm**(*model*)

Auto resets the normalization of a given DEQ model.

Parameters

model (*torch.nn.Module*) – Model to reset normalization.

Example

```
>>> reset_norm(model)
```

torchdeq.norm.**remove_norm**(*model*)

Removes the normalization of a given DEQ model.

Parameters

model (*torch.nn.Module*) – A DEQ model to remove normalization.

Example

```
>>> remove_norm(model)
```

torchdeq.norm.**register_norm**(*norm_type*, *norm_class*)

Registers a user-defined normalization class for the apply_norm function.

This function adds a new entry to the Norm class dict with the key as the specified `norm_type` and the value as the `norm_class`.

Parameters

- **norm_type** (*str*) – The type of normalization to register. This will be used as the key in the Norm class dictionary.
- **norm_class** (*type*) – The class defining the normalization. This will be used as the value in the Norm class dictionary.

Example

```
>>> register_norm('custom_norm', CustomNorm)
```

torchdeq.norm.**register_norm_module**(*module_class*, *norm_type*, *names='weight'*, *dims=0*)

Registers a to-be-normed module for the user-defined normalization class in the *apply_norm* function.

This function adds a new entry to the _target_modules attribute of the specified normalization class in the _norm_class dictionary. The key is the module class and the value is a tuple containing the attribute name and dimension over which to compute the norm.

Parameters

- **module_class** (*type*) – Module class to be indexed for the user-defined normalization class.
- **norm_type** (*str*) – The type of normalization class that the module class should be registered for.
- **names** (*str, optional*) – Attribute name of `module_class` for the normalization to be applied. Default `'weight'`.
- **dims** (*int, optional*) – Dimension over which to compute the norm. Default 0.

Example

```
>>> register_norm_module(Conv2d, 'custom_norm', 'weight', 0)
```

5.2 Normalization

class torchdeq.norm.weight_norm.**WeightNorm**(*names*, *dims*, *learn_scale: bool = True*, *target_norm: float = 1.0*, *clip: bool = False*, *clip_value: float = 1.0*)

classmethod **apply**(*module*, *deq_args=None*, *names=None*, *dims=None*, *learn_scale=True*, *target_norm=1.0*, *clip=False*, *clip_value=1.0*)

Apply weight normalization to a given module.

Parameters

- **module** (*torch.nn.Module*) – The module to apply weight normalization to.
- **deq_args** (*Union[argparse.Namespace, dict, DEQConfig, Any]*) – Configuration for the DEQ model. This can be an instance of argparse.Namespace, a dictionary, or an instance of DEQConfig. Unknown config will be processed using *get_attr* function.
- **names** (*list or str, optional*) – The names of the parameters to apply spectral normalization to.
- **dims** (*list or int, optional*) – The dimensions along which to normalize.
- **learn_scale** (*bool, optional*) – If true, learn a scale factor during training. Default True.
- **target_norm** (*float, optional*) – The target norm value. Default 1.
- **clip** (*bool, optional*) – If true, clip the scale factor. Default False.

- **clip_value** (*float, optional*) – The value to clip the scale factor to. Default 1.

Returns
The WeightNorm instance.

Return type
WeightNorm

compute_weight(*module*, *name*, *dim*)

Computes the weight with weight normalization.

Parameters

- **module** (*torch.nn.Module*) – The module which holds the weight tensor.
- **name** (*str*) – The name of the weight parameter.
- **dim** (*int*) – The dimension along which to normalize.

Returns
The weight tensor after applying weight normalization.

Return type
Tensor

remove(*module*)

Removes weight normalization from the module.

Parameters
module (*torch.nn.Module*) – The module to remove weight normalization from.

class torchdeq.norm.spectral_norm.**SpectralNorm**(*names*, *dims*, *learn_scale: bool = True*, *target_norm: float = 1.0*, *clip: bool = False*, *clip_value: float = 1.0*, *n_power_iterations: int = 1*, *eps: float = 1e-12*)

classmethod apply(*module*, *deq_args=None*, *names=None*, *dims=None*, *learn_scale=True*, *target_norm=1.0*, *clip=False*, *clip_value=1.0*, *n_power_iterations=1*, *eps=1e-12*)

Applies spectral normalization to a given module.

Parameters

- **module** (*torch.nn.Module*) – The module to apply spectral normalization to.
- **deq_args** (*Union[argparse.Namespace, dict, DEQConfig, Any]*) – Configuration for the DEQ model. This can be an instance of argparse.Namespace, a dictionary, or an instance of DEQConfig. Unknown config will be processed using *get_attr* function.
- **names** (*list or str, optional*) – The names of the parameters to apply spectral normalization to.
- **dims** (*list or int, optional*) – The dimensions along which to normalize.
- **learn_scale** (*bool, optional*) – If true, learn a scale factor during training. Default True.
- **target_norm** (*float, optional*) – The target norm value. Default 1.
- **clip** (*bool, optional*) – If true, clip the scale factor. Default False.
- **clip_value** (*float, optional*) – The value to clip the scale factor to. Default 1.
- **n_power_iterations** (*int, optional*) – The number of power iterations to perform. Default 1.

- **eps** (*float, optional*) – A small constant for numerical stability. Default 1e-12.

Returns
The SpectralNorm instance.

Return type
SpectralNorm

compute_weight(*module*, *do_power_iteration*, *name*, *dim*)

Computes the weight with spectral normalization.

Parameters

- **module** (*torch.nn.Module*) – The module which holds the weight tensor.
- **do_power_iteration** (*bool*) – If true, do power iteration for approximating singular vectors.
- **name** (*str*) – The name of the weight parameter.
- **dim** (*int*) – The dimension along which to normalize.

Returns
The computed weight tensor.

Return type
torch.Tensor

remove(*module*)

Removes spectral normalization from the module.

Parameters
module (*torch.nn.Module*) – The module to remove spectral normalization from.

CHAPTER

SIX

TORCHDEQ.DROPOUT

A module containing several implementations of variational dropout.

Variational dropout is a type of dropout where a single dropout mask is generated once per sample and applied consistently across all solver steps in the sample. This is particularly effective when used with implicit models, as it counters overfitting while preserving the dynamics.

This module provides variational dropout for 1d, 2d, and 3d inputs, with both channel-wise and token-wise options.

6.1 Dropout Function

torchdeq.dropout.**reset_dropout**(*model*)

Resets the dropout mask for all variational dropout layers in the model at the beginning of a training iteration.

Parameters

model (*torch.nn.Module*) – A DEQ layer in which the dropout masks should be reset.

6.2 Dropout

class torchdeq.dropout.**VariationalDropout**(*dropout=0.5*)

Applies Variational Dropout to the input tensor.

During training, randomly zeros some of the elements of the input tensor with probability 'dropout' using a mask tensor sampled from a Bernoulli distribution.

The same mask is used for each input in a training iteration. (for fixed point convergence) This random mask is reset at the beginning of the next training iteration using *reset_dropout*.

Parameters

dropout (*float, optional*) – The probability of an element to be zeroed. Default: 0.5.

Shape:

- Input: Tensor of any shape.
- Output: Tensor of the same shape as input.

Examples

```
>>> m = VariationalDropout(dropout=0.5)
>>> input = torch.randn(20, 16)
>>> output = m(input)
```

reset_mask(*x*)

Resets the dropout mask. Subclasses should implement this method according to the dimensionality of the input tensor.

class torchdeq.dropout.**VariationalDropout1d**(*dropout=0.5*, *token_first=True*)

Applies Variational Dropout to the input tensor.

During training, randomly zero out the entire channel/feature dimension of the input 1d tensor with probability 'dropout' using a mask tensor sample from a Bernoulli distribution.

The channel/feature dimension of 1d tensor is the $*$ slice of $(B, L, *)$ for token_first=True, or $(B, *, L)$ for token_first=False.

The same mask is used for each input in a training iteration. (for fixed point convergence) This random mask is reset at the beginning of the next training iteration using *reset_dropout*.

Parameters

- **dropout** (*float, optional*) – The probability of an element to be zeroed. Default: 0.5
- **token_first** (*bool, optional*) – If True, expects input tensor in shape (B, L, D), otherwise expects (B, D, L). Here, *B* is batch size, *L* is sequence length, and *D* is feature dimension. Default: False.

Shape:

- Input: (B, L, D) or (B, D, L).
- Output: (B, L, D) or (B, D, L) (same shape as input).

reset_mask(*x*)

Resets the dropout mask. Subclasses should implement this method according to the dimensionality of the input tensor.

class torchdeq.dropout.**VariationalDropout2d**(*dropout=0.5*, *token_first=True*)

Applies Variational Dropout to the input tensor.

During training, randomly zero out the entire channel/feature dimension of the input 2d tensor with probability 'dropout' using a mask tensor sample from a Bernoulli distribution.

The channel/feature dimension of 2d tensor is the $*$ of $(B, H, W, *)$ for token_first=True, or $(B, *, H, W)$ for token_first=False.

During the fixed point solving, a fixed mask will be applied until convergence. Reset this random mask at the beginning of the next training iteration using *reset_dropout*.

Parameters

- **dropout** (*float, optional*) – The probability of an element to be zeroed. Default: 0.5
- **token_first** (*bool, optional*) – If True, expect input tensor in shape (B, H, W, D), otherwise expect (B, D, H, W). Here, *B* is batch size, and *D* is feature dimension. Default: False

Shape:

- Input: (B, H, W, D) or (B, D, H, W).
- Output: (B, H, W, D) or (B, D, H, W) (same shape as input).

reset_mask(x)

Resets the dropout mask. Subclasses should implement this method according to the dimensionality of the input tensor.

class torchdeq.dropout.**VariationalDropout3d**(*dropout=0.5*, *token_first=True*)

Applies Variational Dropout to the input tensor.

During training, randomly zero out the entire channel/feature dimension of the input 3d tensor with probability 'dropout' using a mask tensor sample from a Bernoulli distribution.

The channel/feature dimension of 3d tensor is the $*$ slice of $(B, T, H, W, *)$ for token_first=True, or $(B, *, T, H, W)$ for token_first=False.

During the fixed point solving, a fixed mask will be applied until convergence. Reset this random mask at the beginning of the next training iteration using *reset_dropout*.

Parameters

- **dropout** (*float, optional*) – The probability of an element to be zeroed. Default: 0.5
- **token_first** (*bool, optional*) – If True, expect input tensor in shape (B, T, H, W, D), otherwise expect (B, D, T, H, W). Here, *B* is batch size, and *D* is feature dimension. Default: False

Shape:

- Input: (B, T, H, W, D) or (B, D, T, H, W).
- Output: (B, T, H, W, D) or (B, D, T, H, W) (same shape as input).

reset_mask(x)

Resets the dropout mask. Subclasses should implement this method according to the dimensionality of the input tensor.

class torchdeq.dropout.**VariationalDropToken1d**(*dropout=0.5*, *token_first=True*)

Applies Variational Dropout to the input tensor.

During training, randomly zero out the entire token/sequence dimension of the input 1d tensor with probability 'dropout' using a mask tensor sample from a Bernoulli distribution.

The token/sequence dimension of 1d tensor is the $*$ slice of $(B, *, L)$ for token_first=True, or $(B, D, *)$ for token_first=False.

During the fixed point solving, a fixed mask will be applied until convergence. Reset this random mask at the beginning of the next training iteration using *reset_dropout*.

Parameters

- **dropout** (*float, optional*) – The probability of an element to be zeroed. Default: 0.5
- **token_first** (*bool, optional*) – If True, expect input tensor in shape (B, L, D), otherwise expect (B, D, L). Here, *B* is batch size, and *D* is feature dimension. Default: False

Shape:

- Input: (B, L, D) or (B, D, L).
- Output: (B, L, D) or (B, D, L) (same shape as input).

reset_mask(*x*)

Resets the dropout mask. Subclasses should implement this method according to the dimensionality of the input tensor.

class torchdeq.dropout.**VariationalDropToken2d**(*dropout=0.5*, *token_first=True*)

Applies Variational Dropout to the input tensor.

During training, randomly zero out the entire token/sequence dimension of the input 2d tensor with probability 'dropout' using a mask tensor sample from a Bernoulli distribution.

The token/sequence dimension of 2d tensor is the $*$ slice of $(B, H, W, *)$ for token_first=True, or $(B, *, H, W)$ for token_first=False.

During the fixed point solving, a fixed mask will be applied until convergence. Reset this random mask at the beginning of the next training iteration using *reset_dropout*.

Parameters

- **dropout** (*float, optional*) – The probability of an element to be zeroed. Default: 0.5
- **token_first** (*bool, optional*) – If True, expect input tensor in shape (B, H, W, D), otherwise expect (B, D, H, W). Here, *B* is batch size, and *D* is feature dimension. Default: False

Shape:

- Input: (B, H, W, D) or (B, D, H, W).
- Output: (B, H, W, D) or (B, D, H, W) (same shape as input).

reset_mask(*x*)

Resets the dropout mask. Subclasses should implement this method according to the dimensionality of the input tensor.

class torchdeq.dropout.**VariationalDropToken3d**(*dropout=0.5*, *token_first=True*)

Applies Variational Dropout to the input tensor.

During training, randomly zero out the entire token/sequence dimension of the input 3d tensor with probability 'dropout' using a mask tensor sample from a Bernoulli distribution.

The token/sequence dimension of 3d tensor is the $*$ slice of $(B, T, H, W, *)$ for token_first=True, or $(B, *, T, H, W)$ for token_first=False.

During the fixed point solving, a fixed mask will be applied until convergence. Reset this random mask at the beginning of the next training iteration using *reset_dropout*.

Parameters

- **dropout** (*float, optional*) – The probability of an element to be zeroed. Default: 0.5
- **token_first** (*bool, optional*) – If True, expect input tensor in shape (B, T, H, W, D), otherwise expect (B, D, T, H, W). Here, *B* is batch size, and *D* is feature dimension. Default: False

Shape:

- Input: (B, T, H, W, D) or (B, D, T, H, W).
- Output: (B, T, H, W, D) or (B, D, T, H, W) (same shape as input).

reset_mask(x)

Resets the dropout mask. Subclasses should implement this method according to the dimensionality of the input tensor.

CHAPTER

SEVEN

TORCHDEQ.LOSS

7.1 Correction

torchdeq.loss.**fp_correction**(*crit*, *args*, *weight_func='exp'*, *return_loss_values=False*, ***kwargs*)

Computes fixed-point correction for stabilizing Deep Equilibrium (DEQ) models.

Fixed point correction applies the loss function to a sequence of tensors that converge to the fixed point. The loss value of each tensor tuple is weighted by the weight function. This function automatically aligns the input arguments to be of the same length.

The currently supported weight functions include `'const'` (constant), `'linear'`, and `'exp'` (exponential).

Parameters

- **crit** (*callable*) – Loss function. Can be the instance of torch.nn.Module or functor.
- **args** (*list or tuple*) – List of arguments to pass to the criterion.
- **weight_func** (*str, optional*) – Name of the weight function to use. Default 'exp'.
- **return_loss_values** (*bool, optional*) – Whether to return the loss values. Default False.
- ****kwargs** – Additional keyword arguments for the weight function.

Returns

The computed loss. list[float]: List of individual loss values. Returned only if return_loss_values is set to True.

Return type

torch.Tensor

Examples

```
>>> x = [torch.randn(16, 32, 32) for _ in range(3)]
>>> y = torch.randn(16, 32, 32)
>>> mask = torch.rand(16, 32, 32)
>>> crit = lambda x, y, mask: ((x - y) * mask).abs().mean()
>>> loss = fp_correction(crit, (x, y, mask))
```

torchdeq.loss.**register_weight_func**(*name*, *func*)

Registers a new weight function for fixed point correction.

The weight function should map a pair of integers (n, k) to a float, serving as the weight of loss, where 'n' is the total length of the sequence that converges to the fixed point, and 'k' is the order of the current state in the sequence.

Parameters

- **name** (*str*) – Identifier to associate with the new weight function.
- **func** (*callable*) – The weight function to register, mapping (n, k) to a float value.

Raises

AssertionError – If `func` is not callable.

7.2 Jacobian

`torchdeq.loss.jac_reg(f0, z0, vecs=1, create_graph=True)`

Estimates tr(J^TJ)=tr(JJ^T) via Hutchinson estimator.

Parameters

- **f0** (*torch.Tensor*) – Output of the function f (whose J is to be analyzed)
- **z0** (*torch.Tensor*) – Input to the function f
- **vecs** (*int, optional*) – Number of random Gaussian vectors to use. Defaults to 2.
- **create_graph** (*bool, optional*) – Whether to create backward graph (e.g., to train on this loss). Defaults to True.

Returns

A 1x1 torch tensor that encodes the (shape-normalized) jacobian loss

Return type

torch.Tensor

`torchdeq.loss.power_method(f0, z0, n_iters=100)`

Estimates the spectral radius of J using power method.

Parameters

- **f0** (*torch.Tensor*) – Output of the function f (whose J is to be analyzed)
- **z0** (*torch.Tensor*) – Input to the function f
- **n_iters** (*int, optional*) – Number of power method iterations. Default is 100.

Returns

(largest eigenvector, largest (abs.) eigenvalue)

Return type

tuple

CHAPTER

EIGHT

TORCHDEQ.UTILS

8.1 Config

torchdeq.utils.**add_deq_args**(*parser*)

Decorate the commonly used argument parser with arguments used in TorchDEQ.

Parameters

parser (*argparse.Namespace*) – Command line arguments.

8.2 Memory

torchdeq.utils.**mem_gc**(*func*, *in_args=None*)

Performs the forward and backward pass of a PyTorch Module using gradient checkpointing.

This function is designed for use with iterative computational graphs and the PyTorch DDP training protocol. In the forward pass, it does not store any activations. During the backward pass, it first recomputes the activations and then applies the vector-Jacobian product (vjp) to calculate gradients with respect to the inputs.

The function automatically tracks gradients for the parameters and input tensors that require gradients. It is particularly useful for creating computational graphs with constant memory complexity, i.e., $\mathcal{O}(1)$ memory.

Parameters

- **func** (*torch.nn.Module*) – Pytorch Module for which gradients will be computed.
- **in_args** (*tuple, optional*) – Input arguments for the function. Default None.

Returns

The output of the *func* Module.

Return type

tuple

8.3 Init

torchdeq.utils.**mixed_init**(*z_shape*, *device=None*)

Initializes a tensor with a shape of *z_shape* with half Gaussian random values and hald zeros.

Proposed in the paper, Path Independent Equilibrium Models Can Better Exploit Test-Time Computation, for better path independence.

Parameters

- **z_shape** (*tuple*) – Shape of the tensor to be initialized.
- **device** (*torch.device, optional*) – The desired device of returned tensor. Default None.

Returns

A tensor of shape *z_shape* with values randomly initialized and zero masked.

Return type

torch.Tensor

CHAPTER

NINE

MODELS IN DEQ ZOO

deq-zoo currently supports six implicit models via TorchDEQ. For each project, we provide a README doc for data preparation and launching instructions.

9.1 DEQ

The first Deep Equilibrium Model is a sequence model that takes advantage of transformers in its model design. Given the injection $U(\mathbf{x}_{0:T})$ from the input sequence and the past context $\mathbf{z}^\star_{0:t}$, DEQ transformer predicts the next tokens via the fixed points $\mathbf{z}^\star_{t:T}$ of a transformer block,

$$\begin{aligned} \mathbf{q}, \mathbf{k}, \mathbf{v} &= \mathbf{w}\mathbf{z}^\star_{0:T} + U(\mathbf{x}_{0:T}) \\ \tilde{\mathbf{z}} &= \mathbf{z}^\star_{t:T} + \text{Attention}\left(\mathbf{q}, \mathbf{k}, \mathbf{v}\right) \\ \mathbf{z}^\star_{t:T} &= \tilde{\mathbf{z}} + \text{FFN}\left(\tilde{\mathbf{z}}\right) \end{aligned}$$

where Attention is MultiHead Decoder Attention, FFN is a 2-layer feed-forward network.

In DEQ Zoo, we implement the DEQ transformer and benchmark it through the word-level language modeling on WikiText-103~\cite{wiki}. The model details and training protocols are redesigned based on TorchDEQ.

- deq-seq: Language modeling on WikiText-103. Implementation using Pytorch DataParallel.
- deq-lm: Faster & updated implementation using PyTorch Distributed Data Parallel (DDP) framework. This is the recommended version.

9.2 MDEQ

This directory contains the code for Multiscale Deep Equilibrium Models(MDEQ) proposed in the paper Multiscale Deep Equilibrium Models.

- mdeq: Code for training MDEQs on CIFAR10 and ImageNet (DDP).

9.3 IGNN

This directory contains the code for Implicit Graph Neural Networks (IGNN) proposed in the paper Implicit Graph Neural Networks.

- ignn: Code for conducting graph and node classification tasks, using datasets like PPI.

9.4 DEQ-Flow

Deep Equilibrium Optical Flow Estimation

- `deq-flow`: Code for training and evaluating optical flow models.

9.5 DEQ-INR

$(\text{Implicit})^2$: Implicit Layers for Implicit Representations.

- `deq-inr`: Code for converting and compressing image, audio, and video data into implicit layers for implicit representations.

9.6 DEQ-DDIM

Deep Equilibrium Approaches to Diffusion Models

- `deq-ddim`: Code for performing parallel diffusion sampling & inversion using the joint equilibrium of the sampling trajectory.

CHAPTER

TEN

TASKS IN DEQ ZOO

To be continued.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

S

T

V

W