CATENets

Alicia Curth

Dec 05, 2022

CONTENTS

CHAPTER

ONE

CATENETS - CONDITIONAL AVERAGE TREATMENT EFFECT ESTIMATION USING NEURAL NETWORKS

Code Author: Alicia Curth (amc253@cam.ac.uk)

This repo contains Jax-based, sklearn-style implementations of Neural Network-based Conditional Average Treatment Effect (CATE) Estimators, which were used in the AISTATS21 paper 'Nonparametric Estimation of Heterogeneous Treatment Effects: From Theory to Learning Algorithms' (Curth & vd Schaar, 2021a) as well as the follow up NeurIPS21 paper "On Inductive Biases for Heterogeneous Treatment Effect Estimation" (Curth & vd Schaar, 2021b) and the NeurIPS21 Datasets & Benchmarks track paper "Really Doing Great at Estimating CATE? A Critical Look at ML Benchmarking Practices in Treatment Effect Estimation" (Curth et al, 2021).

We implement the SNet-class we introduce in Curth & vd Schaar (2021a), as well as FlexTENet and OffsetNet as discussed in Curth & vd Schaar (2021b), and re-implement a number of NN-based algorithms from existing literature (Shalit et al (2017), Shi et al (2019), Hassanpour & Greiner (2020)). We also provide Neural Network (NN)-based instantiations of a number of so-called meta-learners for CATE estimation, including two-step pseudo-outcome regression estimators (the DR-learner (Kennedy, 2020) and single-robust propensity-weighted (PW) and regression-adjusted (RA) learners), Nie & Wager (2017)'s R-learner and Kuenzel et al (2019)'s X-learner. The jax implementations in `catenets.models.jax` were used in all papers listed; additionally, pytorch versions of some models (`catenets.models.torch`) were contributed by Bogdan Cebere.

1.1 Interface

The repo contains a package `catenets`, which contains all general code used for modeling and evaluation, and a folder `experiments`, in which the code for replicating experimental results is contained. All implemented learning algorithms in `catenets` (`SNet, FlexTENet, OffsetNet, TNet, SNet1 (TARNet), SNet2 (DragonNet), SNet3, DRNet, RANet, PWNet, RNet, XNet`) come with a sklearn-style wrapper, implementing a `.fit(X, y, w)` and a `.predict(X)` method, where predict returns CATE by default. All hyperparameters are documented in detail in the respective files in `catenets.models` folder.

Example usage:

```
from catenets.models.jax import TNet, SNet
from catenets.experiment_utils.simulation_utils import simulate_treatment_setup

# simulate some data (here: unconfounded, 10 prognostic variables and 5 predictive␣
↪variables)
X, y, w, p, cate = simulate_treatment_setup(n=2000, n_o=10, n_t=5, n_c=0)
```

(continues on next page)

(continued from previous page)

```
# estimate CATE using TNet
t = TNet()
t.fit(X, y, w)
cate_pred_t = t.predict(X)  # without potential outcomes
cate_pred_t, po0_pred_t, po1_pred_t = t.predict(X, return_po=True)  # predict potential␣
↪outcomes too

# estimate CATE using SNet
s = SNet(penalty_orthogonal=0.01)
s.fit(X, y, w)
cate_pred_s = s.predict(X)
```

All experiments in Curth & vd Schaar (2021a) can be replicated using this repository; the necessary code is in `experiments.experiments_AISTATS21`. To do so from shell, clone the repo, create a new virtual environment and run

```
pip install -r requirements.txt #install requirements
python run_experiments_AISTATS.py
```

```
Options:
--experiment # defaults to 'simulation', 'ihdp' will run ihdp experiments
--setting # different simulation settings in synthetic experiments (can be 1-5)
--models # defaults to None which will train all models considered in paper,
         # can be string of model name (e.g 'TNet'), 'plug' for all plugin models,
         # 'pseudo' for all pseudo-outcome regression models

--file_name # base file name to write to, defaults to 'results'
--n_repeats # number of experiments to run for each configuration, defaults to 10␣
↪(should be set to 100 for IHDP)
```

Similarly, the experiments in Curth & vd Schaar (2021b) can be replicated using the code in `experiments.experiments_inductivebias_NeurIPS21` (or from shell using `python run_experiments_inductive_bias_NeurIPS.py`) and the experiments in Curth et al (2021) can be replicated using the code in `experiments.experiments_benchmarks_NeurIPS21` (the catenets experiments can also be run from shell using `python run_experiments_benchmarks_NeurIPS`).

The code can also be installed as a python package (`catenets`). From a local copy of the repo, run `python setup.py install`.

Note: jax is currently only supported on macOS and linux, but can be run from windows using WSL (the windows subsystem for linux).

1.2 Citing

If you use this software please cite the corresponding paper(s):

```
@inproceedings{curth2021nonparametric,
  title={Nonparametric Estimation of Heterogeneous Treatment Effects: From Theory to␣
↪Learning Algorithms},
  author={Curth, Alicia and van der Schaar, Mihaela},
    year={2021},
```

(continues on next page)

(continued from previous page)

```
  booktitle={Proceedings of the 24th International Conference on Artificial
  Intelligence and Statistics (AISTATS)},
  organization={PMLR}
}

@article{curth2021inductive,
  title={On Inductive Biases for Heterogeneous Treatment Effect Estimation},
  author={Curth, Alicia and van der Schaar, Mihaela},
  booktitle={Proceedings of the Thirty-Fifth Conference on Neural Information Processing␣
↪Systems},
  year={2021}
}


@article{curth2021really,
  title={Really Doing Great at Estimating CATE? A Critical Look at ML Benchmarking␣
↪Practices in Treatment Effect Estimation},
  author={Curth, Alicia and Svensson, David and Weatherall, James and van der Schaar,␣
↪Mihaela},
  booktitle={Proceedings of the Neural Information Processing Systems Track on Datasets␣
↪and Benchmarks},
  year={2021}
}
```

CHAPTER

TWO

API DOCUMENTATION

CHAPTER

THREE

JAX MODELS

3.1 JAX models

JAX-based CATE estimators

3.1.1 catenets.models.jax.tnet module

Implements a T-Net: T-learner for CATE based on a dense NN

class TNet(*binary_y: bool = False*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *penalty_l2: float = 0.0001*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *train_separate: bool = True*, *penalty_diff: float = 0.0001*, *nonlin: str = 'elu'*)

Bases: `catenets.models.jax.base.BaseCATENet`

TNet class – two separate functions learned for each Potential Outcome function

Parameters

- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_layers_r** (*int*) – Number of representation layers before hypothesis layers (distinction between hypothesis layers and representation layers is made to match TARNet & SNets)
- **n_units_r** (*int*) – Number of hidden units in each representation layer
- **penalty_l2** (*float*) – l2 (ridge) penalty
- **step_size** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss

- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **train_separate** (*bool, default True*) – Whether to train the two output heads completely separately or whether to regularize their difference
- **penalty_diff** (*float*) – l2-penalty for regularizing the difference between output heads. used only if train_separate=False
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in NN

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

_train_tnet_jointly(*X: jax._src.basearray.Array, y: jax._src.basearray.Array, w: jax._src.basearray.Array, binary_y: bool = False, n_layers_out: int = 2, n_units_out: int = 100, n_layers_r: int = 3, n_units_r: int = 200, penalty_l2: float = 0.0001, step_size: float = 0.0001, n_iter: int = 10000, batch_size: int = 100, val_split_prop: float = 0.3, early_stopping: bool = True, patience: int = 10, n_iter_min: int = 200, n_iter_print: int = 50, seed: int = 42, return_val_loss: bool = False, same_init: bool = True, penalty_diff: float = 0.0001, nonlin: str = 'elu', avg_objective: bool = True*) → jax._src.basearray.Array

predict_t_net(*X: jax._src.basearray.Array, trained_params: dict, predict_funs: list, return_po: bool = False, return_prop: bool = False*) → jax._src.basearray.Array

train_tnet(*X: jax._src.basearray.Array, y: jax._src.basearray.Array, w: jax._src.basearray.Array, binary_y: bool = False, n_layers_out: int = 2, n_units_out: int = 100, n_layers_r: int = 3, n_units_r: int = 200, penalty_l2: float = 0.0001, step_size: float = 0.0001, n_iter: int = 10000, batch_size: int = 100, val_split_prop: float = 0.3, early_stopping: bool = True, patience: int = 10, n_iter_min: int = 200, n_iter_print: int = 50, seed: int = 42, return_val_loss: bool = False, train_separate: bool = True, penalty_diff: float = 0.0001, nonlin: str = 'elu', avg_objective: bool = True*) → Any

3.1.2 catenets.models.jax.rnet module

Implements NN based on R-learner and U-learner (as discussed in Nie & Wager (2017))

class RNet(*second_stage_strategy: str = 'R', data_split: bool = False, cross_fit: bool = False, n_cf_folds: int = 2, n_layers_out: int = 2, n_layers_r: int = 3, n_layers_out_t: int = 2, n_layers_r_t: int = 3, n_units_out: int = 100, n_units_r: int = 200, n_units_out_t: int = 100, n_units_r_t: int = 200, penalty_l2: float = 0.0001, penalty_l2_t: float = 0.0001, step_size: float = 0.0001, step_size_t: float = 0.0001, n_iter: int = 10000, batch_size: int = 100, n_iter_min: int = 200, val_split_prop: float = 0.3, early_stopping: bool = True, patience: int = 10, n_iter_print: int = 50, seed: int = 42, nonlin: str = 'elu', binary_y: bool = False*)

Bases: `catenets.models.jax.base.BaseCATENet`

Class implements R-learner and U-learner using NNs

Parameters

- **second_stage_strategy** (*str, default 'R'*) – Which strategy to use in the second stage ('R' for R-learner, 'U' for U-learner)
- **data_split** (*bool, default False*) – Whether to split the data in two folds for estimation

- **cross_fit** (*bool, default False*) – Whether to perform cross fitting
- **n_cf_folds** (*int*) – Number of crossfitting folds to use
- **n_layers_out** (*int*) – First stage Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out** (*int*) – First stage Number of hidden units in each hypothesis layer
- **n_layers_r** (*int*) – First stage Number of representation layers before hypothesis layers (distinction between hypothesis layers and representation layers is made to match TARNet & SNets)
- **n_units_r** (*int*) – First stage Number of hidden units in each representation layer
- **n_layers_out_t** (*int*) – Second stage Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out_t** (*int*) – Second stage Number of hidden units in each hypothesis layer
- **n_layers_r_t** (*int*) – Second stage Number of representation layers before hypothesis layers (distinction between hypothesis layers and representation layers is made to match TARNet & SNets)
- **n_units_r_t** (*int*) – Second stage Number of hidden units in each representation layer
- **penalty_l2** (*float*) – First stage l2 (ridge) penalty
- **penalty_l2_t** (*float*) – Second stage l2 (ridge) penalty
- **step_size** (*float*) – First stage learning rate for optimizer
- **step_size_t** (*float*) – Second stage learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in NN

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

fit(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *p: Optional[jax._src.basearray.Array] = None*) → *catenets.models.jax.rnet.RNet*

Fit method for a CATENet. Takes covariates, outcome variable and treatment indicator as input

Parameters

- **X** (*pd.DataFrame or np.array*) – Covariate matrix

- **y** (*np.array*) – Outcome vector
- **w** (*np.array*) – Treatment indicator
- **p** (*np.array*) – Vector of (known) treatment propensities. Currently only supported for TwoStepNets.

predict(*X: jax._src.basearray.Array*, *return_po: bool = False*, *return_prop: bool = False*) → jax._src.basearray.Array

Predict treatment effect estimates using a CATENet. Depending on method, can also return potential outcome estimate and propensity score estimate.

Parameters

- **X** (*pd.DataFrame or np.array*) – Covariate matrix
- **return_po** (*bool, default False*) – Whether to return potential outcome estimate
- **return_prop** (*bool, default False*) – Whether to return propensity estimate

Returns

Return type array of CATE estimates, optionally also potential outcomes and propensity

_train_and_predict_r_stage1(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *fit_mask: jax._src.basearray.Array*, *pred_mask: jax._src.basearray.Array*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *penalty_l2: float = 0.0001*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *nonlin: str = 'elu'*, *binary_y: bool = False*) → Any

train_r_net(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *p: Optional[jax._src.basearray.Array] = None*, *second_stage_strategy: str = 'R'*, *data_split: bool = False*, *cross_fit: bool = False*, *n_cf_folds: int = 2*, *n_layers_out: int = 2*, *n_layers_r: int = 3*, *n_layers_r_t: int = 3*, *n_layers_out_t: int = 2*, *n_units_out: int = 100*, *n_units_r: int = 200*, *n_units_out_t: int = 100*, *n_units_r_t: int = 200*, *penalty_l2: float = 0.0001*, *penalty_l2_t: float = 0.0001*, *step_size: float = 0.0001*, *step_size_t: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *nonlin: str = 'elu'*, *binary_y: bool = False*) → Any

train_r_stage2(*X: jax._src.basearray.Array*, *y_ortho: jax._src.basearray.Array*, *w_ortho: jax._src.basearray.Array*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *n_layers_r: int = 0*, *n_units_r: int = 200*, *penalty_l2: float = 0.0001*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *nonlin: str = 'elu'*, *avg_objective: bool = True*) → Any

3.1.3 catenets.models.jax.xnet module

Module implements X-learner from Kuenzel et al (2019) using NNs

class XNet(*weight_strategy: Optional[int] = None, first_stage_strategy: str = 'T', first_stage_args: Optional[dict] = None, binary_y: bool = False, n_layers_out: int = 2, n_layers_r: int = 3, n_layers_out_t: int = 2, n_layers_r_t: int = 3, n_units_out: int = 100, n_units_r: int = 200, n_units_out_t: int = 100, n_units_r_t: int = 200, penalty_l2: float = 0.0001, penalty_l2_t: float = 0.0001, step_size: float = 0.0001, step_size_t: float = 0.0001, n_iter: int = 10000, batch_size: int = 100, n_iter_min: int = 200, val_split_prop: float = 0.3, early_stopping: bool = True, patience: int = 10, n_iter_print: int = 50, seed: int = 42, nonlin: str = 'elu'*)

Bases: `catenets.models.jax.base.BaseCATENet`

Class implements X-learner using NNs.

Parameters

- **weight_strategy** (*int, default None*) – Which strategy to use to weight the two CATE estimators in the second stage. weight_strategy is coded as follows: for tau(x)=g(x)tau_0(x) + (1-g(x))tau_1(x) [eq 9, kuenzel et al (2019)] weight_strategy=0 sets g(x)=0, weight_strategy=1 sets g(x)=1, weight_strategy=None sets g(x)=pi(x) [propensity score],

  weight_strategy=-1 sets g(x)=(1-pi(x))
- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **n_layers_out** (*int*) – First stage Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out** (*int*) – First stage Number of hidden units in each hypothesis layer
- **n_layers_r** (*int*) – First stage Number of representation layers before hypothesis layers (distinction between hypothesis layers and representation layers is made to match TARNet & SNets)
- **n_units_r** (*int*) – First stage Number of hidden units in each representation layer
- **n_layers_out_t** (*int*) – Second stage Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out_t** (*int*) – Second stage Number of hidden units in each hypothesis layer
- **n_layers_r_t** (*int*) – Second stage Number of representation layers before hypothesis layers (distinction between hypothesis layers and representation layers is made to match TARNet & SNets)
- **n_units_r_t** (*int*) – Second stage Number of hidden units in each representation layer
- **penalty_l2** (*float*) – First stage l2 (ridge) penalty
- **penalty_l2_t** (*float*) – Second stage l2 (ridge) penalty
- **step_size** (*float*) – First stage learning rate for optimizer
- **step_size_t** (*float*) – Second stage learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping

- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in NN

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

predict(*X: jax._src.basearray.Array*, *return_po: bool = False*, *return_prop: bool = False*) → jax._src.basearray.Array

Predict treatment effect estimates using a CATENet. Depending on method, can also return potential outcome estimate and propensity score estimate.

Parameters

- **X** (*pd.DataFrame or np.array*) – Covariate matrix
- **return_po** (*bool, default False*) – Whether to return potential outcome estimate
- **return_prop** (*bool, default False*) – Whether to return propensity estimate

Returns

Return type array of CATE estimates, optionally also potential outcomes and propensity

_get_first_stage_pos(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *first_stage_strategy: str = 'T'*, *first_stage_args: Optional[dict] = None*, *binary_y: bool = False*, *n_layers_out: int = 2*, *n_layers_r: int = 3*, *n_units_out: int = 100*, *n_units_r: int = 200*, *penalty_l2: float = 0.0001*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *n_iter_min: int = 200*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_print: int = 50*, *seed: int = 42*, *nonlin: str = 'elu'*, *avg_objective: bool = True*) → Tuple[jax._src.basearray.Array, jax._src.basearray.Array]

predict_x_net(*X: jax._src.basearray.Array*, *trained_params: dict*, *predict_funs: list*, *return_po: bool = False*, *return_prop: bool = False*, *weight_strategy: Optional[int] = None*) → jax._src.basearray.Array

train_x_net(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *weight_strategy: Optional[int] = None*, *first_stage_strategy: str = 'T'*, *first_stage_args: Optional[dict] = None*, *binary_y: bool = False*, *n_layers_out: int = 2*, *n_layers_r: int = 3*, *n_layers_out_t: int = 2*, *n_layers_r_t: int = 3*, *n_units_out: int = 100*, *n_units_r: int = 200*, *n_units_out_t: int = 100*, *n_units_r_t: int = 200*, *penalty_l2: float = 0.0001*, *penalty_l2_t: float = 0.0001*, *step_size: float = 0.0001*, *step_size_t: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *n_iter_min: int = 200*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_print: int = 50*, *seed: int = 42*, *nonlin: str = 'elu'*, *return_val_loss: bool = False*, *avg_objective: bool = True*) → Tuple

3.1.4 catenets.models.jax.representation_nets module

Module implements SNet1 and SNet2, which are based on CFRNet/TARNet from Shalit et al (2017) and DragonNet from Shi et al (2019), respectively.

class DragonNet(*binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *penalty_l2: float = 0.0001*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 0*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *reg_diff: bool = False*, *same_init: bool = False*, *penalty_diff: float = 0.0001*, *nonlin: str = 'elu'*)

Bases: `catenets.models.jax.representation_nets.SNet2`

Wrapper for DragonNet

_abc_impl = <_abc_data object>

class SNet1(*binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *penalty_l2: float = 0.0001*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *reg_diff: bool = False*, *penalty_diff: float = 0.0001*, *same_init: bool = False*, *nonlin: str = 'elu'*, *penalty_disc: float = 0*)

Bases: `catenets.models.jax.base.BaseCATENet`

Class implements Shalit et al (2017)'s TARNet & CFR (discrepancy regularization is NOT TESTED). Also referred to as SNet-1 in our paper.

Parameters

- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_layers_r** (*int*) – Number of shared representation layers before hypothesis layers
- **n_units_r** (*int*) – Number of hidden units in each representation layer
- **penalty_l2** (*float*) – l2 (ridge) penalty
- **step_size** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **reg_diff** (*bool, default False*) – Whether to regularize the difference between the two potential outcome heads

- **penalty_diff** (*float*) – l2-penalty for regularizing the difference between output heads. used only if train_separate=False
- **same_init** (*bool, False*) – Whether to initialise the two output heads with same values
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in NN
- **penalty_disc** (*float, default zero*) – Discrepancy penalty. Defaults to zero as this feature is not tested.

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

class **SNet2**(*binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *penalty_l2: float = 0.0001*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *reg_diff: bool = False*, *same_init: bool = False*, *penalty_diff: float = 0.0001*, *nonlin: str = 'elu'*)

Bases: `catenets.models.jax.base.BaseCATENet`

Class implements SNet-2, which is based on Shi et al (2019)'s DragonNet (this version does NOT use targeted regularization and has a (possibly deeper) propensity head.

Parameters

- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_layers_out_prop** (*int*) – Number of hypothesis layers for propensity score(n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_units_out_prop** (*int*) – Number of hidden units in each propensity score hypothesis layer
- **n_layers_r** (*int*) – Number of shared representation layers before hypothesis layers
- **n_units_r** (*int*) – Number of hidden units in each representation layer
- **penalty_l2** (*float*) – l2 (ridge) penalty
- **step_size** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used

- **reg_diff** (`bool, default False`) – Whether to regularize the difference between the two potential outcome heads
- **penalty_diff** (`float`) – l2-penalty for regularizing the difference between output heads. used only if train_separate=False
- **same_init** (`bool, False`) – Whether to initialise the two output heads with same values
- **nonlin** (`string, default 'elu'`) – Nonlinearity to use in NN

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

class TARNet(*binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *penalty_l2: float = 0.0001*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *reg_diff: bool = False*, *penalty_diff: float = 0.0001*, *same_init: bool = False*, *nonlin: str = 'elu'*)

Bases: `catenets.models.jax.representation_nets.SNet1`

Wrapper for TARNet

_abc_impl = <_abc_data object>

mmd2_lin(*X: jax._src.basearray.Array*, *w: jax._src.basearray.Array*) → jax._src.basearray.Array

predict_snet1(*X: jax._src.basearray.Array*, *trained_params: dict*, *predict_funs: list*, *return_po: bool = False*, *return_prop: bool = False*) → jax._src.basearray.Array

predict_snet2(*X: jax._src.basearray.Array*, *trained_params: dict*, *predict_funs: list*, *return_po: bool = False*, *return_prop: bool = False*) → jax._src.basearray.Array

train_snet1(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *penalty_l2: float = 0.0001*, *penalty_disc: int = 0*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *reg_diff: bool = False*, *same_init: bool = False*, *penalty_diff: float = 0.0001*, *nonlin: str = 'elu'*, *avg_objective: bool = True*) → Any

train_snet2(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *penalty_l2: float = 0.0001*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *reg_diff: bool = False*, *penalty_diff: float = 0.0001*, *nonlin: str = 'elu'*, *avg_objective: bool = True*, *same_init: bool = False*) → Any

SNet2 corresponds to DragonNet (Shi et al, 2019) [without TMLE regularisation term].

3.1.5 catenets.models.jax.disentangled_nets module

Class implements SNet-3, a variation on DR-CFR discussed in Hassanpour and Greiner (2020) and Wu et al (2020).

class SNet3(*binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 150*, *n_layers_out: int = 2*, *n_units_r_small: int = 50*, *n_units_out: int = 100*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *penalty_l2: float = 0.0001*, *penalty_orthogonal: float = 0.01*, *penalty_disc: float = 0*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *nonlin: str = 'elu'*, *reg_diff: bool = False*, *penalty_diff: float = 0.0001*, *same_init: bool = False*)

Bases: `catenets.models.jax.base.BaseCATENet`

Class implements SNet-3, which is based on Hassanpour & Greiner (2020)'s DR-CFR (Without propensity weighting), using an orthogonal regularizer to enforce decomposition similar to Wu et al (2020).

Parameters

- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_layers_out_prop** (*int*) – Number of hypothesis layers for propensity score(n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_units_out_prop** (*int*) – Number of hidden units in each propensity score hypothesis layer
- **n_layers_r** (*int*) – Number of shared & private representation layers before hypothesis layers
- **n_units_r** (*int*) – Number of hidden units in representation layer shared by propensity score and outcome function (the 'confounding factor')
- **n_units_r_small** (*int*) – Number of hidden units in representation layer NOT shared by propensity score and outcome functions (the 'outcome factor' and the 'instrumental factor')
- **penalty_l2** (*float*) – l2 (ridge) penalty
- **step_size** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **reg_diff** (*bool, default False*) – Whether to regularize the difference between the two potential outcome heads

- **penalty_diff** (*float*) – l2-penalty for regularizing the difference between output heads. used only if train_separate=False
- **same_init** (*bool, False*) – Whether to initialise the two output heads with same values
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in NN
- **penalty_disc** (*float, default zero*) – Discrepancy penalty. Defaults to zero as this feature is not tested.

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

_concatenate_representations(*reps: jax._src.basearray.Array*) → jax._src.basearray.Array

_get_absolute_rowsums(*mat: jax._src.basearray.Array*) → jax._src.basearray.Array

predict_snet3(*X: jax._src.basearray.Array*, *trained_params: dict*, *predict_funs: list*, *return_po: bool = False*, *return_prop: bool = False*) → jax._src.basearray.Array

train_snet3(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 150*, *n_units_r_small: int = 50*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *penalty_l2: float = 0.0001*, *penalty_disc: float = 0*, *penalty_orthogonal: float = 0.01*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *n_iter_min: int = 200*, *patience: int = 10*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *reg_diff: bool = False*, *penalty_diff: float = 0.0001*, *nonlin: str = 'elu'*, *avg_objective: bool = True*, *same_init: bool = False*) → Any

SNet-3, based on the decompostion used in Hassanpour and Greiner (2020)

3.1.6 catenets.models.jax.snet module

Module implements SNet class as discussed in Curth & van der Schaar (2021)

class SNet(*with_prop: bool = True*, *binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 100*, *n_layers_out: int = 2*, *n_units_r_small: int = 50*, *n_units_out: int = 100*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *penalty_l2: float = 0.0001*, *penalty_orthogonal: float = 0.01*, *penalty_disc: float = 0*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *reg_diff: bool = False*, *penalty_diff: float = 0.0001*, *seed: int = 42*, *nonlin: str = 'elu'*, *same_init: bool = False*, *ortho_reg_type: str = 'abs'*)

Bases: `catenets.models.jax.base.BaseCATENet`

Class implements SNet as discussed in Curth & van der Schaar (2021). Additionally to the version implemented in the AISTATS paper, we also include an implementation that does not have propensity heads (set with_prop=False)

Parameters

- **with_prop** (*bool, True*) – Whether to include propensity head
- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_layers_out_prop** (*int*) – Number of hypothesis layers for propensity score(n_layers_out x n_units_out + 1 x Dense layer)

- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_units_out_prop** (*int*) – Number of hidden units in each propensity score hypothesis layer
- **n_layers_r** (*int*) – Number of shared & private representation layers before hypothesis layers
- **n_units_r** (*int*) – If withprop=True: Number of hidden units in representation layer shared by propensity score and outcome function (the 'confounding factor') and in the ('instrumental factor') If withprop=False: Number of hidden units in representation shared across PO function
- **n_units_r_small** (*int*) – If withprop=True: Number of hidden units in representation layer of the 'outcome factor' and each PO functions private representation if withprop=False: Number of hidden units in each PO functions private representation
- **penalty_l2** (*float*) – l2 (ridge) penalty
- **step_size** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **reg_diff** (*bool, default False*) – Whether to regularize the difference between the two potential outcome heads
- **penalty_diff** (*float*) – l2-penalty for regularizing the difference between output heads. used only if train_separate=False
- **same_init** (*bool, False*) – Whether to initialise the two output heads with same values
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in NN
- **penalty_disc** (*float, default zero*) – Discrepancy penalty. Defaults to zero as this feature is not tested.
- **ortho_reg_type** (*str, 'abs'*) – Which type of orthogonalization to use. 'abs' uses the (hard) disentanglement described in AISTATS paper, 'fro' uses frobenius norm as in FlexTENet

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

predict_snet(*X: jax._src.basearray.Array*, *trained_params: jax._src.basearray.Array*, *predict_funs: list*, *return_po: bool = False*, *return_prop: bool = False*) → jax._src.basearray.Array

predict_snet_noprop(*X: jax._src.basearray.Array*, *trained_params: jax._src.basearray.Array*, *predict_funs: list*, *return_po: bool = False*, *return_prop: bool = False*) → jax._src.basearray.Array

train_snet(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 100*, *n_units_r_small: int = 50*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *penalty_l2: float = 0.0001*, *penalty_disc: float = 0*, *penalty_orthogonal: float = 0.01*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *reg_diff: bool = False*, *penalty_diff: float = 0.0001*, *nonlin: str = 'elu'*, *avg_objective: bool = True*, *with_prop: bool = True*, *same_init: bool = False*, *ortho_reg_type: str = 'abs'*) → Tuple

train_snet_noprop(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 150*, *n_units_r_small: int = 50*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *penalty_l2: float = 0.0001*, *penalty_orthogonal: float = 0.01*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *n_iter_min: int = 200*, *patience: int = 10*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *reg_diff: bool = False*, *penalty_diff: float = 0.0001*, *nonlin: str = 'elu'*, *avg_objective: bool = True*, *with_prop: bool = False*, *same_init: bool = False*, *ortho_reg_type: str = 'abs'*) → Tuple

SNet but without the propensity head

3.1.7 catenets.models.jax.flextenet module

Module implements FlexTENet, also referred to as the 'flexible approach' in "On inductive biases for heterogeneous treatment effect estimation", Curth & vd Schaar (2021).

DenseW(*out_dim: int*, *W_init: Callable = <function variance_scaling.<locals>.init>*, *b_init: Callable = <function normal.<locals>.init>*) → Tuple

Layer constructor function for a dense (fully-connected) layer. Adapted to allow passing treatment indicator through layer without using it

class FlexTENet(*binary_y: bool = False*, *n_layers_out: int = 2*, *n_units_s_out: int = 50*, *n_units_p_out: int = 50*, *n_layers_r: int = 3*, *n_units_s_r: int = 100*, *n_units_p_r: int = 100*, *private_out: bool = False*, *penalty_l2: float = 0.0001*, *penalty_l2_p: float = 0.0001*, *penalty_orthogonal: float = 0.01*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *opt: str = 'adam'*, *shared_repr: bool = False*, *pretrain_shared: bool = False*, *same_init: bool = True*, *lr_scale: float = 10*, *normalize_ortho: bool = False*)

Bases: `catenets.models.jax.base.BaseCATENet`

Module implements FlexTENet, an architecture for treatment effect estimation that allows for both shared and private information in each layer of the network.

Parameters

- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_s_out** (*int*) – Number of hidden units in each shared hypothesis layer
- **n_units_p_out** (*int*) – Number of hidden units in each private hypothesis layer
- **n_layers_r** (*int*) – Number of representation layers before hypothesis layers (distinction between hypothesis layers and representation layers is made to match TARNet & SNets)

- **n_units_s_r** (*int*) – Number of hidden units in each shared representation layer
- **n_units_s_r** – Number of hidden units in each private representation layer
- **private_out** (*bool, False*) – Whether the final prediction layer should be fully private, or retain a shared component.
- **penalty_l2** (*float*) – l2 (ridge) penalty
- **penalty_l2_p** (*float*) – l2 (ridge) penalty for private layers
- **penalty_orthogonal** (*float*) – orthogonalisation penalty
- **step_size** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **opt** (*str, default 'adam'*) – Optimizer to use, accepts 'adam' and 'sgd'
- **shared_repr** (*bool, False*) – Whether to use a shared representation block as TARNet
- **pretrain_shared** (*bool, False*) – Whether to pretrain the shared component of the network while freezing the private parameters
- **same_init** (*bool, True*) – Whether to use the same initialisation for all private spaces
- **lr_scale** (*float*) – Whether to scale down the learning rate after unfreezing the private components of the network (only used if pretrain_shared=True)
- **normalize_ortho** (*bool, False*) – Whether to normalize the orthogonality penalty (by depth of network)

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

FlexTENetArchitecture(*n_layers_out: int = 2*, *n_units_s_out: int = 50*, *n_units_p_out: int = 50*, *n_layers_r: int = 3*, *n_units_s_r: int = 100*, *n_units_p_r: int = 100*, *private_out: bool = False*, *binary_y: bool = False*, *shared_repr: bool = False*, *same_init: bool = True*) → Any

SplitLayerAsymmetric(*n_units_s: int*, *n_units_p: int*, *first_layer: bool = False*, *same_init: bool = True*) → Tuple

TEOutputLayerAsymmetric(*private: bool = True*, *same_init: bool = True*) → Tuple

_compute_ortho_penalty_asymmetric(*params: jax._src.basearray.Array*, *n_layers_out: int*, *n_layers_r: int*, *private_out: int*, *penalty_orthogonal: float*, *shared_repr: bool*, *normalize_ortho: bool*, *mode: int = 1*) → float

_compute_penalty(*params: jax._src.basearray.Array*, *n_layers_out: int*, *n_layers_r: int*, *private_out: int*, *penalty_l2: float*, *penalty_l2_p: float*, *penalty_orthogonal: float*, *shared_repr: bool*, *normalize_ortho: bool*, *mode: int = 1*) → jax._src.basearray.Array

_compute_penalty_l2(*params: jax._src.basearray.Array*, *n_layers_out: int*, *n_layers_r: int*, *private_out: int*, *penalty_l2: float*, *penalty_l2_p: float*, *shared_repr: bool*, *mode: int = 1*) → jax._src.basearray.Array

_get_cos_reg(*params_0: jax._src.basearray.Array*, *params_1: jax._src.basearray.Array*, *normalize: bool*) → jax._src.basearray.Array

elementwise_parallel(*fun: Callable*, ***fun_kwargs: Any*) → Tuple

Layer that applies a scalar function elementwise on its inputs. Adapted from original jax.stax to allow three inputs and to skip treatment indicator.

Input looks like: X_s, X_p0, X_p1, t = inputs

elementwise_split(*fun: Callable*, ***fun_kwargs: Any*) → Tuple

Layer that applies a scalar function elementwise on its inputs. Adapted from original jax.stax to skip treatment indicator.

Input looks like: X, t = inputs

predict_flextenet(*X: jax._src.basearray.Array*, *trained_params: jax._src.basearray.Array*, *predict_funs: Callable*, *return_po: bool = False*, *return_prop: bool = False*) → Any

train_flextenet(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *binary_y: bool = False*, *n_layers_out: int = 2*, *n_units_s_out: int = 50*, *n_units_p_out: int = 50*, *n_layers_r: int = 3*, *n_units_s_r: int = 100*, *n_units_p_r: int = 100*, *private_out: bool = False*, *penalty_l2: float = 0.0001*, *penalty_l2_p: float = 0.0001*, *penalty_orthogonal: float = 0.01*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *avg_objective: bool = True*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *opt: str = 'adam'*, *shared_repr: bool = False*, *pretrain_shared: bool = False*, *same_init: bool = True*, *lr_scale: float = 10*, *normalize_ortho: bool = False*, *nonlin: str = 'elu'*, *n_units_r: Optional[int] = None*, *n_units_out: Optional[int] = None*) → Tuple

3.1.8 catenets.models.jax.offsetnet module

Module implements OffsetNet, also referred to as the 'reparametrization approach' and 'hard approach' in "On inductive biases for heterogeneous treatment effect estimation", Curth & vd Schaar (2021); modeling the POs using a shared prognostic function and an offset (treatment effect)

class OffsetNet(*binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *penalty_l2: float = 0.0001*, *penalty_l2_p: float = 0.0001*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *nonlin: str = 'elu'*)

Bases: `catenets.models.jax.base.BaseCATENet`

Module implements OffsetNet, also referred to as the 'reparametrization approach' and 'hard approach' in Curth & vd Schaar (2021); modeling the POs using a shared prognostic function and an offset (treatment effect).

Parameters

- **binary_y** (`bool, default False`) – Whether the outcome is binary
- **n_layers_out** (`int`) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)

- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_layers_r** (*int*) – Number of representation layers before hypothesis layers (distinction between hypothesis layers and representation layers is made to match TARNet & SNets)
- **n_units_r** (*int*) – Number of hidden units in each representation layer
- **penalty_l2** (*float*) – l2 (ridge) penalty
- **step_size** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **early_stopping** (*bool, default True*) – Whether to use early stopping
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **penalty_l2_p** (*float*) – l2-penalty for regularizing the offset
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in NN

_abc_impl = <_abc_data object>

_get_predict_function() → Callable

_get_train_function() → Callable

predict_offsetnet(*X: jax._src.basearray.Array*, *trained_params: jax._src.basearray.Array*, *predict_funs: List[Any]*, *return_po: bool = False*, *return_prop: bool = False*) → jax._src.basearray.Array

train_offsetnet(*X: jax._src.basearray.Array*, *y: jax._src.basearray.Array*, *w: jax._src.basearray.Array*, *binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 200*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *penalty_l2: float = 0.0001*, *penalty_l2_p: float = 0.0001*, *step_size: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *early_stopping: bool = True*, *patience: int = 10*, *n_iter_min: int = 200*, *n_iter_print: int = 50*, *seed: int = 42*, *return_val_loss: bool = False*, *nonlin: str = 'elu'*, *avg_objective: bool = True*) → Tuple

CHAPTER

FOUR

PYTORCH MODELS

4.1 PyTorch models

PyTorch-based CATE estimators

4.1.1 catenets.models.torch.tlearner module

class TLearner(*n_unit_in: int*, *binary_y: bool*, *po_estimator: Optional[Any] = None*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *weight_decay: float = 0.0001*, *lr: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *n_iter_print: int = 50*, *seed: int = 42*, *nonlin: str = 'elu'*, *batch_norm: bool = True*, *early_stopping: bool = True*, *dropout: bool = False*, *dropout_prob: float = 0.2*)

Bases: `catenets.models.torch.base.BaseCATEEstimator`

TLearner class – two separate functions learned for each Potential Outcome function

Parameters

- **n_unit_in** (*int*) – Number of features
- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **po_estimator** (*sklearn/PyTorch model, default: None*) – Custom plugin model. If this parameter is set, the rest of the parameters are ignored.
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Linear layer)
- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **weight_decay** (*float*) – l2 (ridge) penalty
- **lr** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in the neural net. Cat be ‘elu’, ‘relu’, ‘selu’ or ‘leaky_relu’.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[torch.Tensor]]

_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]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[torch.nn.parameter.Parameter]]

_plug_in: Any

_state_dict_hooks: Dict[int, Callable]

fit(*X: torch.Tensor*, *y: torch.Tensor*, *w: torch.Tensor*) → *catenets.models.torch.tlearner.TLearner*

Train plug-in models.

Parameters

- **X** (*torch.Tensor (n_samples, n_features)*) – The features to fit to
- **y** (*torch.Tensor (n_samples,) or (n_samples,)*) – The outcome variable
- **w** (*torch.Tensor (n_samples,)*) – The treatment indicator

predict(*X: torch.Tensor*, *return_po: bool = False*, *training: bool = False*) → torch.Tensor

Predict treatment effects and potential outcomes :param X: Test-sample features :type X: torch.Tensor of shape (n_samples, n_features) :param return_po: Return potential outcomes too :type return_po: bool

Returns y

Return type torch.Tensor of shape (n_samples,)

training: bool

4.1.2 catenets.models.torch.slearner module

class SLearner(*n_unit_in: int*, *binary_y: bool*, *po_estimator: Optional[Any] = None*, *n_layers_out: int = 2*, *n_units_out: int = 100*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *weight_decay: float = 0.0001*, *lr: float = 0.0001*, *n_iter: int = 10000*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *n_iter_print: int = 50*, *seed: int = 42*, *nonlin: str = 'elu'*, *weighting_strategy: Optional[str] = None*, *batch_norm: bool = True*, *early_stopping: bool = True*, *dropout: bool = False*, *dropout_prob: float = 0.2*)

Bases: `catenets.models.torch.base.BaseCATEEstimator`

S-learner for treatment effect estimation (single learner, treatment indicator just another feature).

Parameters

- **n_unit_in** (*int*) – Number of features
- **binary_y** (*bool*) – Whether the outcome is binary
- **po_estimator** (*sklearn/PyTorch model, default: None*) – Custom potential outcome model. If this parameter is set, the rest of the parameters are ignored.
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Linear layer)

- **n_layers_out_prop** (*int*) – Number of hypothesis layers for propensity score(n_layers_out x n_units_out + 1 x Linear layer)
- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_units_out_prop** (*int*) – Number of hidden units in each propensity score hypothesis layer
- **weight_decay** (*float*) – l2 (ridge) penalty
- **lr** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in the neural net. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **weighting_strategy** (*optional str, None*) – Whether to include propensity head and which weightening strategy to use

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[torch.Tensor]]

_create_extended_matrices(*X: torch.Tensor*) → torch.Tensor

_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]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[torch.nn.parameter.Parameter]]

_state_dict_hooks: Dict[int, Callable]

fit(*X: torch.Tensor*, *y: torch.Tensor*, *w: torch.Tensor*) → *catenets.models.torch.slearner.SLearner*

Fit treatment models.

Parameters

- **X** (*torch.Tensor of shape (n_samples, n_features)*) – The features to fit to
- **y** (*torch.Tensor of shape (n_samples,) or (n_samples,)*) – The outcome variable
- **w** (*torch.Tensor of shape (n_samples,)*) – The treatment indicator

predict(*X: torch.Tensor*, *return_po: bool = False*, *training: bool = False*) → torch.Tensor

Predict treatment effects and potential outcomes

Parameters **X** (*array-like of shape (n_samples, n_features)*) – Test-sample features

Returns y

Return type array-like of shape (n_samples,)

training: bool

4.1.3 catenets.models.torch.representation_nets module

class BasicDragonNet(*name: str, n_unit_in: int, propensity_estimator: torch.nn.modules.module.Module, binary_y: bool = False, n_layers_r: int = 3, n_units_r: int = 200, n_layers_out: int = 2, n_units_out: int = 100, weight_decay: float = 0.0001, lr: float = 0.0001, n_iter: int = 10000, batch_size: int = 100, val_split_prop: float = 0.3, n_iter_print: int = 50, seed: int = 42, nonlin: str = 'elu', weighting_strategy: Optional[str] = None, penalty_disc: float = 0, batch_norm: bool = True, early_stopping: bool = True, prop_loss_multiplier: float = 1, n_iter_min: int = 200, patience: int = 10, dropout: bool = False, dropout_prob: float = 0.2*)

Bases: `catenets.models.torch.base.BaseCATEEstimator`

Base class for TARNet and DragonNet.

Parameters

- **name** (*str*) – Estimator name
- **n_unit_in** (*int*) – Number of features
- **propensity_estimator** (*nn.Module*) – Propensity estimator
- **binary_y** (*bool, default False*) – Whether the outcome is binary
- **n_layers_out** (*int*) – Number of hypothesis layers (n_layers_out x n_units_out + 1 x Dense layer)
- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_layers_r** (*int*) – Number of shared & private representation layers before the hypothesis layers.
- **n_units_r** (*int*) – Number of hidden units in representation before the hypothesis layers.
- **weight_decay** (*float*) – l2 (ridge) penalty
- **lr** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in the neural net. Can be 'elu', 'relu', 'selu', 'leaky_relu'.
- **weighting_strategy** (*optional str, None*) – Whether to include propensity head and which weightening strategy to use
- **penalty_disc** (*float, default zero*) – Discrepancy penalty.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[torch.Tensor]]

`_forward`(*X: torch.Tensor*) → torch.Tensor

`_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]`

`_maximum_mean_discrepancy`(*X: torch.Tensor*, *w: torch.Tensor*) → torch.Tensor

`_modules: Dict[str, Optional[Module]]`

`_non_persistent_buffers_set: Set[str]`

`_parameters: Dict[str, Optional[torch.nn.parameter.Parameter]]`

`_state_dict_hooks: Dict[int, Callable]`

`abstract _step`(*X: torch.Tensor*, *w: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

`fit`(*X: torch.Tensor*, *y: torch.Tensor*, *w: torch.Tensor*) → *catenets.models.torch.representation_nets.BasicDragonNet*

Fit the treatment models.

Parameters

- **X** (*torch.Tensor of shape (n_samples, n_features)*) – The features to fit to
- **y** (*torch.Tensor of shape (n_samples,) or (n_samples,)*) – The outcome variable
- **w** (*torch.Tensor of shape (n_samples,)*) – The treatment indicator

`loss`(*po_pred: torch.Tensor*, *t_pred: torch.Tensor*, *y_true: torch.Tensor*, *t_true: torch.Tensor*, *discrepancy: torch.Tensor*) → torch.Tensor

`predict`(*X: torch.Tensor*, *return_po: bool = False*, *training: bool = False*) → torch.Tensor

Predict the treatment effects

Parameters **X** (*array-like of shape (n_samples, n_features)*) – Test-sample features

Returns y

Return type array-like of shape (n_samples,)

`training: bool`

`class DragonNet`(*n_unit_in: int*, *binary_y: bool = False*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 0*, *nonlin: str = 'elu'*, *n_units_r: int = 200*, *batch_norm: bool = True*, *dropout: bool = False*, *dropout_prob: float = 0.2*, ***kwargs: Any*)

Bases: `catenets.models.torch.representation_nets.BasicDragonNet`

Class implements a variant based on Shi et al (2019)'s DragonNet.

`_backward_hooks: Dict[int, Callable]`

`_buffers: Dict[str, Optional[torch.Tensor]]`

`_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]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[torch.nn.parameter.Parameter]]

_state_dict_hooks: Dict[int, Callable]

_step(*X: torch.Tensor*, *w: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

training: bool

class **TARNet**(*n_unit_in: int*, *binary_y: bool = False*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 0*, *nonlin: str = 'elu'*, *penalty_disc: float = 0*, *batch_norm: bool = True*, *dropout: bool = False*, *dropout_prob: float = 0.2*, ***kwargs: Any*)

Bases: `catenets.models.torch.representation_nets.BasicDragonNet`

Class implements Shalit et al (2017)'s TARNet

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[torch.Tensor]]

_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]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[torch.nn.parameter.Parameter]]

_state_dict_hooks: Dict[int, Callable]

_step(*X: torch.Tensor*, *w: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

training: bool

4.1.4 catenets.models.torch.snet module

class **SNet**(*n_unit_in: int*, *binary_y: bool = False*, *n_layers_r: int = 3*, *n_units_r: int = 100*, *n_layers_out: int = 2*, *n_units_r_small: int = 50*, *n_units_out: int = 100*, *n_units_out_prop: int = 100*, *n_layers_out_prop: int = 2*, *weight_decay: float = 0.0001*, *penalty_orthogonal: float = 0.01*, *penalty_disc: float = 0*, *lr: float = 0.0001*, *n_iter: int = 10000*, *n_iter_min: int = 200*, *batch_size: int = 100*, *val_split_prop: float = 0.3*, *n_iter_print: int = 50*, *seed: int = 42*, *nonlin: str = 'elu'*, *ortho_reg_type: str = 'abs'*, *patience: int = 10*, *clipping_value: int = 1*, *batch_norm: bool = True*, *with_prop: bool = True*, *early_stopping: bool = True*, *prop_loss_multiplier: float = 1*, *dropout: bool = False*, *dropout_prob: float = 0.2*)

Bases: `catenets.models.torch.base.BaseCATEEstimator`

Class implements SNet as discussed in Curth & van der Schaar (2021). Additionally to the version implemented in the AISTATS paper, we also include an implementation that does not have propensity heads (set with_prop=False) :param n_unit_in: Number of features :type n_unit_in: int :param binary_y: Whether the outcome is binary :type binary_y: bool, default False :param n_layers_r: Number of shared & private representation layers before the hypothesis layers. :type n_layers_r: int :param n_units_r: Number of hidden units in representation shared before the hypothesis layer. :type n_units_r: int :param n_layers_out: Number of hypothesis layers (n_layers_out x n_units_out + 1 x Linear layer) :type n_layers_out: int :param n_layers_out_prop: Number of hypothesis layers for propensity score(n_layers_out x n_units_out + 1 x Linear

layer)

Parameters

- **n_units_out** (*int*) – Number of hidden units in each hypothesis layer
- **n_units_out_prop** (*int*) – Number of hidden units in each propensity score hypothesis layer
- **n_units_r_small** (*int*) – Number of hidden units in each PO functions private representation
- **weight_decay** (*float*) – l2 (ridge) penalty
- **lr** (*float*) – learning rate for optimizer
- **n_iter** (*int*) – Maximum number of iterations
- **batch_size** (*int*) – Batch size
- **val_split_prop** (*float*) – Proportion of samples used for validation split (can be 0)
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **n_iter_print** (*int*) – Number of iterations after which to print updates
- **seed** (*int*) – Seed used
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in the neural net. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **penalty_disc** (*float, default zero*) – Discrepancy penalty. Defaults to zero as this feature is not tested.
- **clipping_value** (*int, default 1*) – Gradients clipping value

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[torch.Tensor]]

_forward(*X: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

_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]

_maximum_mean_discrepancy(*X: torch.Tensor*, *w: torch.Tensor*) → torch.Tensor

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_ortho_reg() → float

_parameters: Dict[str, Optional[torch.nn.parameter.Parameter]]

`_state_dict_hooks: Dict[int, Callable]`

`_step`(*X: torch.Tensor*, *w: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

`fit`(*X: torch.Tensor*, *y: torch.Tensor*, *w: torch.Tensor*) → *catenets.models.torch.snet.SNet*

Fit treatment models.

Parameters

- **X** (*torch.Tensor of shape (n_samples, n_features)*) – The features to fit to
- **y** (*torch.Tensor of shape (n_samples,) or (n_samples,)*) – The outcome variable
- **w** (*torch.Tensor of shape (n_samples,)*) – The treatment indicator

`loss`(*y0_pred: torch.Tensor*, *y1_pred: torch.Tensor*, *t_pred: torch.Tensor*, *discrepancy: torch.Tensor*, *y_true: torch.Tensor*, *t_true: torch.Tensor*) → torch.Tensor

`predict`(*X: torch.Tensor*, *return_po: bool = False*, *training: bool = False*) → torch.Tensor

Predict treatment effects and potential outcomes

Parameters **X** (*array-like of shape (n_samples, n_features)*) – Test-sample features

Returns **y**

Return type array-like of shape (n_samples,)

`training: bool`

CHAPTER

FIVE

DATASETS

5.1 Datasets

Dataloaders for datasets used for experiments.

5.1.1 catenets.datasets.dataset_ihdp module

IHDP (Infant Health and Development Program) dataset

`get_one_data_set`(*D: dict*, *i_exp: int*, *get_po: bool = True*) → dict

Helper for getting the IHDP data for one experiment. Adapted from https://github.com/clinicalml/cfrnet

Parameters

- `D` (*dict or pd.DataFrame*) – All the experiment
- `i_exp` (*int*) – Experiment number

Returns **data** – dict with the experiment

Return type dict or pd.Dataframe

`load`(*data_path: pathlib.Path*, *exp: int = 1*, *rescale: bool = False*, ***kwargs: Any*) → Tuple

Get IHDP train/test datasets with treatments and labels.

Parameters `data_path` (*Path*) – Path to the dataset csv. If the data is missing, it will be downloaded.

Returns

- **X** (*pd.Dataframe or array*) – The training feature set
- **w** (*pd.DataFrame or array*) – Training treatment assignments.
- **y** (*pd.Dataframe or array*) – The training labels
- **training potential outcomes** (*pd.DataFrame or array.*) – Potential outcomes for the training set.
- **X_t** (*pd.DataFrame or array*) – The testing feature set
- **testing potential outcomes** (*pd.DataFrame of array*) – Potential outcomes for the testing set.

`load_data_npz`(*fname: pathlib.Path*, *get_po: bool = True*) → dict

Helper function for loading the IHDP data set (adapted from https://github.com/clinicalml/cfrnet)

Parameters `fname` (*Path*) – Dataset path

Returns **data** – Raw IHDP dict, with X, w, y and yf keys.

Return type dict

`load_raw`(*data_path: pathlib.Path*) → Tuple

Get IHDP raw train/test sets.

Parameters **`data_path`** (*`Path`*) – Path to the dataset csv. If the data is missing, it will be downloaded.

Returns

- **data_train** (*dict or pd.DataFrame*) – Training data
- **data_test** (*dict or pd.DataFrame*) – Testing data

`prepare_ihdp_data`(*data_train: dict*, *data_test: dict*, *rescale: bool = False*, *setting: str = 'C'*, *return_pos: bool = False*) → Tuple

Helper for preprocessing the IHDP dataset.

Parameters

- **`data_train`** (*`pd.DataFrame or dict`*) – Train dataset
- **`data_test`** (*`pd.DataFrame or dict`*) – Test dataset
- **`rescale`** (*`bool, default False`*) – Rescale the outcomes to have similar scale
- **`setting`** (*`str, default C`*) – Experiment setting
- **`return_pos`** (*`bool`*) – Return potential outcomes

Returns

- **X** (*dict or pd.DataFrame*) – Training Feature set
- **y** (*pd.DataFrame or list*) – Outcome list
- **t** (*pd.DataFrame or list*) – Treatment list
- **cate_true_in** (*pd.DataFrame or list*) – Average treatment effects for the training set
- **X_t** (*pd.Dataframe or list*) – Test feature set
- **cate_true_out** (*pd.DataFrame of list*) – Average treatment effects for the testing set

5.1.2 catenets.datasets.dataset_twins module

Twins dataset Load real-world individualized treatment effects estimation datasets

- Reference: http://data.nber.org/data/linked-birth-infant-death-data-vital-statistics-data.html

`load`(*data_path: pathlib.Path*, *train_ratio: float = 0.8*, *treatment_type: str = 'rand'*, *seed: int = 42*, *treat_prop: float = 0.5*) → Tuple

Twins dataset dataloader.

- Download the dataset if needed.
- Load the dataset.
- Preprocess the data.
- Return train/test split.

Parameters

- **data_path** (*Path*) – Path to the CSV. If it is missing, it will be downloaded.
- **train_ratio** (*float*) – Train/test ratio
- **treatment_type** (*str*) – Treatment generation strategy
- **seed** (*float*) – Random seed
- **treat_prop** (*float*) – Treatment proportion

Returns

- **train_x** (*array or pd.DataFrame*) – Features in training data.
- **train_t** (*array or pd.DataFrame*) – Treatments in training data.
- **train_y** (*array or pd.DataFrame*) – Observed outcomes in training data.
- **train_potential_y** (*array or pd.DataFrame*) – Potential outcomes in training data.
- **test_x** (*array or pd.DataFrame*) – Features in testing data.
- **test_potential_y** (*array or pd.DataFrame*) – Potential outcomes in testing data.

preprocess(*fn_csv: pathlib.Path*, *train_ratio: float = 0.8*, *treatment_type: str = 'rand'*, *seed: int = 42*, *treat_prop: float = 0.5*) → Tuple

Helper for preprocessing the Twins dataset.

Parameters

- **fn_csv** (*Path*) – Dataset CSV file path.
- **train_ratio** (*float*) – The ratio of training data.
- **treatment_type** (*string*) – The treatment selection strategy.
- **seed** (*float*) – Random seed.

Returns

- **train_x** (*array or pd.DataFrame*) – Features in training data.
- **train_t** (*array or pd.DataFrame*) – Treatments in training data.
- **train_y** (*array or pd.DataFrame*) – Observed outcomes in training data.
- **train_potential_y** (*array or pd.DataFrame*) – Potential outcomes in training data.
- **test_x** (*array or pd.DataFrame*) – Features in testing data.
- **test_potential_y** (*array or pd.DataFrame*) – Potential outcomes in testing data.

5.1.3 catenets.datasets.dataset_acic2016 module

ACIC2016 dataset

get_acic_covariates(*fn_csv: pathlib.Path*, *keep_categorical: bool = False*, *preprocessed: bool = True*) → numpy.ndarray

get_acic_orig_filenames(*data_path: pathlib.Path*, *simu_num: int*) → list

get_acic_orig_outcomes(*data_path: pathlib.Path*, *simu_num: int*, *i_exp: int*) → Tuple

load(*data_path: pathlib.Path*, *preprocessed: bool = True*, *original_acic_outcomes: bool = False*, ***kwargs: Any*) → Tuple

ACIC2016 dataset dataloader.

- Download the dataset if needed.
- Load the dataset.
- Preprocess the data.
- Return train/test split.

Parameters

- **data_path** (*Path*) – Path to the CSV. If it is missing, it will be downloaded.
- **preprocessed** (*bool*) – Switch between the raw and preprocessed versions of the dataset.
- **original_acic_outcomes** (*bool*) – Switch between new simulations (Inductive bias paper) and original acic outcomes

Returns

- **train_x** (*array or pd.DataFrame*) – Features in training data.
- **train_t** (*array or pd.DataFrame*) – Treatments in training data.
- **train_y** (*array or pd.DataFrame*) – Observed outcomes in training data.
- **train_potential_y** (*array or pd.DataFrame*) – Potential outcomes in training data.
- **test_x** (*array or pd.DataFrame*) – Features in testing data.
- **test_potential_y** (*array or pd.DataFrame*) – Potential outcomes in testing data.

preprocess(*fn_csv: pathlib.Path*, *data_path: pathlib.Path*, *preprocessed: bool = True*, *original_acic_outcomes: bool = False*, ***kwargs: Any*) → Tuple

preprocess_acic_orig(*fn_csv: pathlib.Path*, *data_path: pathlib.Path*, *preprocessed: bool = False*, *keep_categorical: bool = True*, *simu_num: int = 1*, *i_exp: int = 0*, *train_size: int = 4000*, *random_split: bool = False*) → Tuple

preprocess_simu(*fn_csv: pathlib.Path*, *n_0: int = 2000*, *n_1: int = 200*, *n_test: int = 500*, *error_sd: float = 1*, *sp_lin: float = 0.6*, *sp_nonlin: float = 0.3*, *prop_gamma: float = 0*, *prop_omega: float = 0*, *ate_goal: float = 0*, *inter: bool = True*, *i_exp: int = 0*, *keep_categorical: bool = False*, *preprocessed: bool = True*) → Tuple

5.1.4 catenets.datasets.network module

Utilities and helpers for retrieving the datasets

download_gdrive_if_needed(*path: pathlib.Path*, *file_id: str*) → None

Helper for downloading a file from Google Drive, if it is now already on the disk.

Parameters

- **path** (*Path*) – Where to download the file
- **file_id** (*str*) – Google Drive File ID. Details: https://developers.google.com/drive/api/v3/about-files

download_http_if_needed(*path: pathlib.Path*, *url: str*) → None

Helper for downloading a file, if it is now already on the disk.

Parameters

- **path** (*Path*) – Where to download the file.

- **url** (*URL string*) – HTTP URL for the dataset.

download_if_needed(*download_path: pathlib.Path*, *file_id: Optional[str] = None*, *http_url: Optional[str] = None*, *unarchive: bool = False*, *unarchive_folder: Optional[pathlib.Path] = None*) → None

Helper for retrieving online datasets.

Parameters

- **download_path** (*str*) – Where to download the archive
- **file_id** (*str, optional*) – Set this if you want to download from a public Google drive share
- **http_url** (*str, optional*) – Set this if you want to download from a HTTP URL
- **unarchive** (*bool*) – Set this if you want to try to unarchive the downloaded file
- **unarchive_folder** (*str*) – Mandatory if you set unarchive to True.

unarchive_if_needed(*path: pathlib.Path*, *output_folder: pathlib.Path*) → None

Helper for uncompressing archives. Supports .tar.gz and .tar.

Parameters

- **path** (*Path*) – Source archive.
- **output_folder** (*Path*) – Where to unarchive.

PYTHON MODULE INDEX

C

INDEX

Symbols

B

C

D

R

S

T

U

X