pydelfi Documentation

Release 0.6

justinalsing

Nov 17, 2020

Table of Contents

CHAPTER 1

Introduction to PyDelfi

PyDelfi implements Density Estimation Likelihood-Free Inference with neural density estimators and adaptive acquisition of simulations. The implemented methods are described in detail in [Alsing, Charnock, Feeney and Wandelt 2019](https://arxiv.org/abs/1903.00007), and are based closely on [Papamakarios, Sterratt and Murray 2018](https://arxiv.org/pdf/1805.07226.pdf), [Lueckmann et al 2018](https://arxiv.org/abs/1805.09294) and [Alsing, Wandelt and Feeney, 2018](https://academic.oup.com/mnras/article-abstract/477/3/2874/4956055?redirectedFrom=fulltext). Please cite these papers if you use this code!

1.1 Quick start

Once everything is installed, try out either *cosmic_shear.ipynb* or *jla_sne.ipynb* as example templates for how to use the code; plugging in your own simulator and letting pydelfi do it's thing.

If you have a set of pre-run simulations you'd like to throw in rather than allowing pydelfi to run simulations on-the-fly, look at *cosmic_shear_prerun_sims.ipynb* as a template for how to do this.

If you are interested in using pydelfi with nuisance hardened data compression to project out nuisances ([Alsing & Wandelt 2019](https://arxiv.org/abs/1903.01473v1)), take a look at *jla_sne_marginalized.ipynb*.

CHAPTER 2

Advanced Example

In this example we'll create an ensemble of Neural Density Estimators to do an LFI (likelihood-free inference) estimate of the density of

```
import numpy as np
import matplotlib.pyplot as plt
import getdist
from getdist import plots, MCSamples
import tensorflow as tf
import pickle
```

Now, let's load up the pydelfi library.

```
import pydelfi.distributions.priors as priors
import pydelfi.ndes.ndes as ndes
import pydelfi as delfi
import pydelfi.simulators.cosmic_shear.cosmic_shear as cosmic_shear
import pydelfi.compression.score.score as score
tf.logging.set_verbosity(tf.logging.ERROR)
```

Here, we will focus on the simulator for cosmic shear. First, let's generate some power spectrum between l=10 and 1000. Here, pz is the photo-z distribution. Here, we'll set up **:python:'simulator'** as a function that calls the power spectrum simulator.

```
# Set up the tomography simulations
CosmicShearSimulator = cosmic_shear.TomographicCosmicShear(
    pz = pickle.load(open('pydelfi2/pydelfi/simulators/cosmic_shear/pz_5bin.pkl', 'rb
↪')),
    lmin = 10, lmax = 1000, n_ell_bins = 5,
    sigma_e = 0.3, nbar = 30, Area = 15000)

# Simulator function: This must be of the form simulator(theta, seed, args) ->␣
↪simulated data vector
def simulator(theta, seed, simulator_args, batch=1):
    return CosmicShearSimulator.simulate(theta, seed)
```

(continues on next page)

(continued from previous page)

```
# Simulator arguments
simulator_args = None
```

Generally you would like a weakly informated prior. For our NDE, we define truncated Gaussian priors. The following variables declare our truncation bounds as well as the prior mean and convariance, over the following Cosmological parameters: Omega_M, S_8, Omega_b, h, and n_S.

```
# Define the priors parameters
lower = np.array([0, 0.4, 0, 0.4, 0.7])
upper = np.array([1, 1.2, 0.1, 1.0, 1.3])
prior_mean = np.array([0.3, 0.8, 0.05, 0.70, 0.96])
prior_covariance = np.eye(5)*np.array([0.1, 0.1, 0.05, 0.3, 0.3])**2

# Prior
prior = priors.TruncatedGaussian(prior_mean, prior_covariance, lower, upper)
```

We work in the idealized case where sampling distribution for each l bin in our power spectra is Wishart. In other words, we compress our data as if it were a Wishart distributed. (Each l-bin is a Wishart distributed.)

```
# Fiducial parameters
theta_fiducial = np.array([0.3, 0.8, 0.05, 0.70, 0.96])

# Expected support of Wishart likelihood (fiducial inverse power spectrum)
C = CosmicShearSimulator.power_spectrum(theta_fiducial)
Cinv = np.array([np.linalg.inv(C[l,:,:]) for l in range(CosmicShearSimulator.n_ell_
↪bins)])

# Degrees of freedom (effective number of modes per band power)
nl = CosmicShearSimulator.nl

# Calculate derivatives of the expected power spectrum
step = np.array(abs(theta_fiducial)*np.array([0.05, 0.05, 0.05, 0.05, 0.05]))
dCdt = CosmicShearSimulator.compute_derivatives(theta_fiducial, step)

# Define compression as score-MLE of a Wishart likelihood
Compressor = score.Wishart(theta_fiducial, nl, Cinv, dCdt, prior_mean=prior_mean,␣
↪prior_covariance=prior_covariance)

# Pull out Fisher matrix inverse
Finv = Compressor.Finv

# Compressor function: This must have the form compressor(data, args) -> compressed␣
↪summaries (pseudoMLE)
def compressor(d, compressor_args):
    return Compressor.scoreMLE(d)
compressor_args = None
```

Next

```
seed = 0
data = simulator(theta_fiducial, seed, simulator_args)
compressed_data = compressor(data, compressor_args)
```

Next

```
    # Create an ensemble of NDEs
NDEs = [ndes.ConditionalMaskedAutoregressiveFlow(n_parameters=5, n_data=5, n_
↪hiddens=[50,50], n_mades=5, act_fun=tf.tanh, index=0),
        ndes.MixtureDensityNetwork(n_parameters=5, n_data=5, n_components=1, n_
↪hidden=[30,30], activations=[tf.tanh, tf.tanh], index=1),
        ndes.MixtureDensityNetwork(n_parameters=5, n_data=5, n_components=2, n_
↪hidden=[30,30], activations=[tf.tanh, tf.tanh], index=2),
        ndes.MixtureDensityNetwork(n_parameters=5, n_data=5, n_components=3, n_
↪hidden=[30,30], activations=[tf.tanh, tf.tanh], index=3),
        ndes.MixtureDensityNetwork(n_parameters=5, n_data=5, n_components=4, n_
↪hidden=[30,30], activations=[tf.tanh, tf.tanh], index=4),
        ndes.MixtureDensityNetwork(n_parameters=5, n_data=5, n_components=5, n_
↪hidden=[30,30], activations=[tf.tanh, tf.tanh], index=5)]

# Create the DELFI object
DelfiEnsemble = delfi.Delfi(compressed_data, prior, NDEs, Finv=Finv, theta_
↪fiducial=theta_fiducial,
                      param_limits = [lower, upper],
                      param_names = ['\Omega_m', 'S_8', '\Omega_b', 'h', 'n_s'],
                      results_dir = "pydelfi2/pydelfi/simulators/cosmic_shear/
↪results/",
                      input_normalization='fisher')
```

Next

```
# Do the Fisher pre-training
DelfiEnsemble.fisher_pretraining()
```

Next

```
# Initial samples, batch size for population samples, number of populations
n_initial = 200
n_batch = 200
n_populations = 39

# Do the SNL training
DelfiEnsemble.sequential_training(simulator, compressor, n_initial, n_batch, n_
↪populations, patience=10, save_intermediate_posteriors=True)
```

CHAPTER 3

pydelfi package

3.1 Submodules

3.2 pydelfi.delfi module

class pydelfi.delfi.**Delfi**(*data*, *prior*, *nde*, *Finv=None*, *theta_fiducial=None*, *param_limits=None*, *param_names=None*, *nwalkers=100*, *posterior_chain_length=1000*, *proposal_chain_length=100*, *rank=0*, *n_procs=1*, *comm=None*, *red_op=None*, *show_plot=True*, *results_dir=''*, *progress_bar=True*, *input_normalization=None*, *graph_restore_filename='graph_checkpoint'*, *restore_filename='restore.pkl'*, *restore=False*, *save=True*)

Bases: object

acquisition(*theta*)

add_simulations(*xs_batch*, *ps_batch*)

allocate_jobs(*n_jobs*)

bayesian_optimization_training(*simulator*, *compressor*, *n_batch*, *n_populations*, *n_optimizations=10*, *simulator_args=None*, *compressor_args=None*, *plot=False*, *batch_size=100*, *validation_split=0.1*, *epochs=300*, *patience=20*, *seed_generator=None*, *save_intermediate_posteriors=False*, *sub_batch=1*)

complete_array(*target_distrib*)

emcee_sample(*log_target=None*, *x0=None*, *burn_in_chain=100*, *main_chain=1000*)

fisher_pretraining(*n_batch=5000*, *plot=True*, *batch_size=100*, *validation_split=0.1*, *epochs=1000*, *patience=20*, *mode='regression'*)

load_simulations(*xs_batch*, *ps_batch*)

log_geometric_mean_proposal_stacked(*x*, *data*)

log_likelihood_individual(*i*, *theta*, *data*)

log_likelihood_stacked(*theta*, *data*)

log_posterior_individual(*i*, *theta*, *data*)

log_posterior_stacked(*theta*, *data*)

run_simulation_batch(*n_batch*, *ps*, *simulator*, *compressor*, *simulator_args*, *compressor_args*, *seed_generator=None*, *sub_batch=1*)

saver()

sequential_training(*simulator*, *compressor*, *n_initial*, *n_batch*, *n_populations*, *proposal=None*, *simulator_args=None*, *compressor_args=None*, *safety=5*, *plot=True*, *batch_size=100*, *validation_split=0.1*, *epochs=300*, *patience=20*, *seed_generator=None*, *save_intermediate_posteriors=True*, *sub_batch=1*)

sequential_training_plot(*savefig=False*, *filename=None*)

train_ndes(*training_data=None*, *batch_size=100*, *validation_split=0.1*, *epochs=500*, *patience=20*, *mode='samples'*)

triangle_plot(*samples=None*, *weights=None*, *savefig=False*, *filename=None*)

3.3 pydelfi.ndes module

class pydelfi.ndes.**ConditionalGaussianMade**(*n_parameters*, *n_data*, *n_hiddens*, *act_fun*, *output_order='sequential'*, *mode='sequential'*, *input_parameters=None*, *input_data=None*, *logpdf=None*)

Bases: object

Implements a Made, where each conditional probability is modelled by a single gaussian component.

create_degrees(*input_order*)

Generates a degree for each hidden and input unit. A unit with degree d can only receive input from units with degree less than d. :param n_hiddens: a list with the number of hidden units :param input_order: the order of the inputs; can be 'random', 'sequential', or an array of an explicit order :param mode: the strategy for assigning degrees to hidden nodes: can be 'random' or 'sequential' :return: list of degrees

create_masks(*degrees*)

Creates the binary masks that make the connectivity autoregressive. :param degrees: a list of degrees for every layer :return: list of all masks, as theano shared variables

create_weights_conditional(*n_comps*)

Creates all learnable weight matrices and bias vectors. :param n_comps: number of gaussian components :return: weights and biases, as tensorflow variables

eval(*xy*, *sess*, *log=True*)

Evaluate log probabilities for given input-output pairs. :param xy: a pair (x, y) where x rows are inputs and y rows are outputs :param sess: tensorflow session where the graph is run :param log: whether to return probabilities in the log domain :return: log probabilities: log p(y|x)

class pydelfi.ndes.**ConditionalMaskedAutoregressiveFlow**(*n_parameters*, *n_data*, *n_hiddens*, *act_fun*, *n_mades*, *output_order='sequential'*, *mode='sequential'*, *input_parameters=None*, *input_data=None*, *logpdf=None*, *index=1*)

Bases: object

Conditional Masked Autoregressive Flow.

eval(*xy*, *sess*, *log=True*)

Evaluate log probabilities for given input-output pairs. :param xy: a pair (x, y) where x rows are inputs and y rows are outputs :param sess: tensorflow session where the graph is run :param log: whether to return probabilities in the log domain :return: log probabilities: log p(y|x)

class pydelfi.ndes.**MixtureDensityNetwork**(*n_parameters, n_data, n_components=3, n_hidden=[50, 50], activations=[<sphinx.ext.autodoc.importer._MockObject object>, <sphinx.ext.autodoc.importer._MockObject object>], input_parameters=None, input_data=None, logpdf=None, index=1*)

Bases: object

Implements a Mixture Density Network for modeling p(y|x)

eval(*xy*, *sess*, *log=True*)

Evaluate log probabilities for given input-output pairs. :param xy: a pair (x, y) where x rows are inputs and y rows are outputs :param sess: tensorflow session where the graph is run :param log: whether to return probabilities in the log domain :return: log probabilities: log p(y|x)

3.4 pydelfi.priors module

class pydelfi.priors.**TruncatedGaussian**(*mean*, *C*, *lower*, *upper*)

Bases: object

draw()

logpdf(*x*)

loguniform(*x*)

pdf(*x*)

uniform(*x*)

class pydelfi.priors.**Uniform**(*lower*, *upper*)

Bases: object

draw()

logpdf(*x*)

pdf(*x*)

3.5 pydelfi.score module

class pydelfi.score.**Gaussian**(*ndata*, *theta_fiducial*, *mu=None*, *Cinv=None*, *dmudt=None*, *dCdt=None*, *F=None*, *prior_mean=None*, *prior_covariance=None*, *rank=0*, *n_procs=1*, *comm=None*, *red_op=None*)

Bases: object

allocate_jobs(*n_jobs*)

complete_array(*target_distrib*)

compute_derivatives(*simulator*, *nsims*, *h*, *simulator_args=None*, *seed_generator=None*, *progress_bar=True*, *sub_batch=1*)

compute_fisher()

compute_mean_covariance(*simulator*, *nsims*, *simulator_args=None*, *seed_generator=None*, *progress_bar=True*, *sub_batch=1*)

projected_scoreMLE(*d*, *nuisances*)

scoreMLE(*d*)

class pydelfi.score.**Wishart**(*theta_fiducial*, *nu*, *Cinv*, *dCdt*, *F=None*, *prior_mean=None*, *prior_covariance=None*)

Bases: object

fisher()

projected_scoreMLE(*d*, *nuisances*)

scoreMLE(*d*)

pydelfi.score.**isnotebook**()

3.6 pydelfi.train module

class pydelfi.train.**ConditionalTrainer**(*model*, *optimizer=<sphinx.ext.autodoc.importer._MockObject object>*, *optimizer_arguments={}*)

Bases: object

train(*sess*, *train_data*, *validation_split=0.1*, *epochs=1000*, *batch_size=100*, *patience=20*, *saver_name='tmp_model'*, *progress_bar=True*, *mode='samples'*)

Training function to be called with desired parameters within a tensorflow session. :param sess: tensorflow session where the graph is run. :param train_data: a tuple/list of (X,Y) with training data where Y is conditioned on X. :param validation_split: percentage of training data randomly selected to be used for validation :param epochs: maximum number of epochs for training. :param batch_size: batch size of each batch within an epoch. :param early_stopping: number of epochs for early stopping criteria. :param check_every_N: check every N iterations if model has improved and saves if so. :param saver_name: string of name (with or without folder) where model is saved. If none is given,

a temporal model is used to save and restore best model, and removed afterwards.

3.7 Module contents

CHAPTER 4

Module Reference

- modindex
- genindex
- search

Python Module Index

p

Index

M

P

R

S

T

U

W