SGMCMC Documentation

Release

Moritz Freidank

Aug 25, 2017

Contents:

1 Sampling **3**
- 1.1 Base Classes for MCMC Methods . . . 3
- 1.2 Stochastic Gradient Hamiltonian Monte Carlo (SGHMC) . . . 8
- 1.3 Stochastic Gradient Langevin Dynamics (SGLD) . . . 9
- 1.4 Relativistic Stochastic Gradient Hamiltonian Monte Carlo (RelSGHMC) . . . 11

2 Bayesian Neural Network **13**

3 Tensor Utils **17**

4 Indices and tables **23**

Python Module Index **25**

This package provides out-of-the-box implementations of various state-of-the-art Stochastic Gradient Markov Chain Monte Carlo sampling methods.

CHAPTER 1

Sampling

This module contains implementations of various SGMCMC samplers which are well-suited for Bayesian Deep Learning. For instance, all samplers implemented in this module can be easily used to learn a BayesianNeuralNetwork. .. XXX: Cross reference to BNN doku (and to samplers?)

Base Classes for MCMC Methods

This module provides abstract base classes for our SGMCMC sampling methods. All subclasses inheriting from any of these base classes automatically conform to the iterator protocol.

This means that extracting the next sample with corresponding costs from *any of our samplers* is as simple as:

```
sample, cost = next(sampler)
```

class mcmc_base_classes.**MCMCSampler**(*params*, *seed=None*, *batch_generator=None*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*)

Generic base class for all MCMC samplers.

__init__(*params*, *seed=None*, *batch_generator=None*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*)

Initialize the sampler base class. Sets up member variables and initializes uninitialized target parameters in the current *tensorflow.Graph*.

Parameters **params** : list of *tensorflow.Variable* objects

Target parameters for which we want to sample new values.

seed : int, optional

Random seed to use. Defaults to *None*.

batch_generator : *BatchGenerator*, optional

Iterable which returns dictionaries to feed into tensorflow.Session.run() calls to evaluate the cost function. Defaults to *None* which indicates that no batches shall be fed.

dtype : tensorflow.DType, optional

Type of elements of *tensorflow.Tensor* objects used in this sampler. Defaults to *tensorflow.float64*.

session : *tensorflow.Session*, optional

Session object which knows about the external part of the graph (which defines *Cost*, and possibly batches). Used internally to evaluate (burn-in/sample) the sampler.

See also:

tensorflow_mcmc.sampling.BurnInMCMCSampler Abstract base class for samplers that perform a burn-in phase to tune their own hyperparameters. Inherits from *sampling.MCMCSampler*.

__iter__()
Allows using samplers as iterators.

Examples

Extract the first three thousand samples (with costs) from a sampler:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> from itertools import islice
>>> from tensorflow_mcmc.sampling.sghmc import SGHMCSampler
>>> session = tf.Session()
>>> x = tf.Variable(1.0)
>>> dist = tf.contrib.distributions.Normal(loc=0., scale=1.)
>>> n_burn_in, n_samples = 1000, 2000
>>> sampler = SGHMCSampler(params=[x], burn_in_steps=n_burn_in, cost_
↪fun=lambda x: -dist.log_prob(x), session=session, dtype=tf.float32)
>>> session.run(tf.global_variables_initializer())
>>> burn_in_samples = list(islice(sampler, n_burn_in))  # perform all burn_in␣
↪steps
>>> samples = list(islice(sampler, n_samples))
>>> len(burn_in_samples), len(samples)
(1000, 2000)
>>> session.close()
>>> tf.reset_default_graph()  # to avoid polluting test environment
```

__metaclass__
alias of ABCMeta

__next__(*feed_vals={}*)

Compute and return the next sample and next cost values for this sampler.

Returns sample: list of numpy.ndarray objects

Sampled values are a *numpy.ndarray* for each target parameter.

cost: numpy.ndarray (1,)

Current cost value of the last evaluated target parameter values.

Examples

Extract the next sample (with costs) from a sampler:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> from itertools import islice
>>> from tensorflow_mcmc.sampling.sghmc import SGHMCSampler
>>> session = tf.Session()
>>> x = tf.Variable(1.0)
>>> dist = tf.contrib.distributions.Normal(loc=0., scale=1.)
>>> n_burn_in = 1000
>>> sampler = SGHMCSampler(params=[x], burn_in_steps=n_burn_in, cost_
↪fun=lambda x:-dist.log_prob(x), session=session, dtype=tf.float32)
>>> session.run(tf.global_variables_initializer())
>>> sample, cost = next(sampler)
>>> session.close()
>>> tf.reset_default_graph()  # to avoid polluting test environment
```

__weakref__
list of weak references to the object (if defined)

_draw_noise_sample(*Sigma*, *Shape*)

Generate a single random normal sample with shape *Shape* and standard deviation *Sigma*.

Parameters **Sigma** : tensorflow.Tensor

Standard deviation of the noise.

Shape : tensorflow.Tensor

Shape that the noise sample should have.

Returns noise_sample: tensorflow.Tensor

Random normal sample with shape *Shape* and standard deviation *Sigma*.

_next_batch()

Get a dictionary mapping *tensorflow.Placeholder* onto their corresponding feedable minibatch data. Each dictionary can directly be fed into *tensorflow.Session*.

Returns an empty dictionary if *self.batch_generator* is *None*, i.e. if no batches are needed to compute the cost function. (e.g. the cost function depends only on the target parameters).

Returns batch:

Dictionary that maps *tensorflow.Placeholder* objects onto *ndarray* objects that can be fed for them. Returns an empty *dict* if *self.batch_generator* is *None*, i.e. if no batches are needed to compute the cost function (e.g. the cost function depends only on the target parameters).

Examples

Extracting batches without any *batch_generator* function simply returns an empty *dict*: >>> import tensorflow as tf >>> import numpy as np >>> from itertools import islice >>> from tensorflow_mcmc.sampling.sghmc import SGHMCSampler >>> session = tf.Session() >>> x = tf.Variable(1.0) >>> dist = tf.contrib.distributions.Normal(loc=0., scale=1.) >>> sampler = SGHMCSampler(params=[x],

cost_fun=lambda x: -dist.log_prob(x), session=session, dtype=tf.float32) >>> session.close() >>> sampler._next_batch() {}

A simple case with batches would look like this: >>> import tensorflow as tf >>> from tensorflow_mcmc.bayesian_neural_network import generate_batches >>> from tensorflow_mcmc.sampling.sghmc import SGHMCSampler >>> session = tf.Session() >>> N, D = 100, 3 # 100 datapoints with 3 features each >>> X = np.asarray([np.random.uniform(-10, 10, D) for _ in range(N)]) >>> y = np.asarray([np.random.choice([0., 1.]) for _ in range(N)]) >>> x_placeholder, y_placeholder = tf.placeholder(dtype=tf.float64), tf.placeholder(dtype=tf.float64) >>> batch_size = 10 >>> batch_generator = generate_batches(X=X, y=y, x_placeholder=x_placeholder, y_placeholder=y_placeholder, batch_size=batch_size) >>> sampler = SGHMCSampler(params=[x], cost_fun=lambda x: x, session=session, dtype=tf.float32, batch_generator=batch_generator) # cost function is just a dummy >>> batch_dict = sampler._next_batch() >>> session.close() >>> set(batch_dict.keys()) == set((x_placeholder, y_placeholder)) True >>> batch_dict[x_placeholder].shape, batch_dict[y_placeholder].shape ((10, 3), (10, 1))

_uninitialized_params(*params*)

Determine a list of *tensorflow.Variable* objects in iterable *params* that are not yet initialized.

Parameters **params** : list of tensorflow.Variable objects

List of target parameters that we want to sample values for.

Returns params_uninitialized: list of tensorflow.Variable objects

All target parameters in *params* that were not initialized yet in the current graph.

For some applications (e.g. Bayesian Optimization), it is important that samplers come with as few design choices as possible. To reduce the number of such design choices, a recent contribution in the literature proposes an on-line *burn-in* procedure.

class mcmc_base_classes.**BurnInMCMCSampler**(*params*, *burn_in_steps*, *seed=None*, *batch_generator=None*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*)

Bases: *mcmc_base_classes.MCMCSampler*

Base class for MCMC samplers that use a burn-in procedure to estimate their mass matrix. Details of how this burn-in is performed are left to be specified in the individual samplers that inherit from this class.

__init__(*params*, *burn_in_steps*, *seed=None*, *batch_generator=None*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*)

Initializes the corresponding MCMCSampler super object and sets member variables.

Parameters **params** : list of *tensorflow.Variable* objects

Target parameters for which we want to sample new values.

burn_in_steps: int

Number of burn-in steps to perform. In each burn-in step, this sampler will adapt its own internal parameters to decrease its error. For reference see: TODO ADD PAPER REFERENCE HERE

seed : int, optional

Random seed to use. Defaults to *None*.

batch_generator : *BatchGenerator*, optional

Iterable which returns dictionaries to feed into tensorflow.Session.run() calls to evaluate the cost function. Defaults to *None* which indicates that no batches shall be fed.

dtype : tensorflow.DType, optional

Type of elements of *tensorflow.Tensor* objects used in this sampler. Defaults to *tensorflow.float64*.

session : *tensorflow.Session*, optional

Session object which knows about the external part of the graph (which defines *Cost*, and possibly batches). Used internally to evaluate (burn-in/sample) the sampler.

See also:

tensorflow_mcmc.sampling.mcmc_base_classes.MCMCSampler Super class of this class. Has generic methods shared by all MCMC samplers implemented as part of this framework.

tensorflow_mcmc.sampling.sghmc.SGHMCSampler Instantiation of this class. Uses SGHMC to sample from the target distribution after burn-in.

tensorflow_mcmc.sampling.sgld.SGLDSampler Instantiation of this class. Uses SGLD to sample from the target distribution after burn-in.

__metaclass__
alias of ABCMeta

__next__()

Perform a sampler step: Compute and return the next sample and next cost values for this sampler.

While *self.is_burning_in* returns *True* (while the sampler has not yet performed *self.burn_in_steps* steps) this will also adapt the samplers mass matrix in a sampler-specific way to improve performance.

Returns sample: list of numpy.ndarray objects

Sampled values are a *numpy.ndarray* for each target parameter.

cost: numpy.ndarray (1,)

Current cost value of the last evaluated target parameter values.

is_burning_in

Check if this sampler is still in burn-in phase. Used during graph construction to insert conditionals into the graph that will make the sampler skip all burn-in operations after the burn-in phase is over.

Returns is_burning_in: boolean

True if *self.n_iterations* <= *self.burn_in_steps*, otherwise *False*.

Stochastic Gradient Hamiltonian Monte Carlo (SGHMC)

class sghmc.**SGHMCSampler**(*params*, *cost_fun*, *seed=None*, *batch_generator=None*, *epsilon=0.01*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*, *burn_in_steps=3000*, *scale_grad=1.0*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *mdecay=0.05*)

Bases: tensorflow_mcmc.sampling.mcmc_base_classes.BurnInMCMCSampler

Stochastic Gradient Hamiltonian Monte-Carlo Sampler that uses a burn-in procedure to adapt its own hyperparameters during the initial stages of sampling.

See [1] for more details on this burn-in procedure. See [2] for more details on Stochastic Gradient Hamiltonian Monte-Carlo.

[1] J. T. Springenberg, A. Klein, S. Falkner, F. Hutter Bayesian Optimization with Robust Bayesian Neural Networks. In Advances in Neural Information Processing Systems 29 (2016).

[2] T. Chen, E. B. Fox, C. Guestrin Stochastic Gradient Hamiltonian Monte Carlo In Proceedings of Machine Learning Research 32 (2014).

__init__(*params*, *cost_fun*, *seed=None*, *batch_generator=None*, *epsilon=0.01*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*, *burn_in_steps=3000*, *scale_grad=1.0*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *mdecay=0.05*)

Initialize the sampler parameters and set up a tensorflow.Graph for later queries.

Parameters **params** : list of tensorflow.Variable objects

Target parameters for which we want to sample new values.

cost_fun : callable

Function that takes *params* as input and returns a 1-d *tensorflow.Tensor* that contains the cost-value. Frequently denoted with *U* in literature.

seed : int, optional

Random seed to use. Defaults to *None*.

batch_generator : BatchGenerator, optional

Iterable which returns dictionaries to feed into tensorflow.Session.run() calls to evaluate the cost function. Defaults to *None* which indicates that no batches shall be fed.

epsilon : float, optional

Value that is used as learning rate parameter for the sampler, also denoted as discretization parameter in literature. Defaults to *0.01*.

session : tensorflow.Session, optional

Session object which knows about the external part of the graph (which defines *Cost*, and possibly batches). Used internally to evaluate (burn-in/sample) the sampler.

burn_in_steps: int, optional

Number of burn-in steps to perform. In each burn-in step, this sampler will adapt its own internal parameters to decrease its error. For reference see: TODO ADD PAPER REFERENCE HERE

scale_grad : float, optional

Value that is used to scale the magnitude of the noise used during sampling. In a typical batches-of-data setting this usually corresponds to the number of examples in the entire dataset. Defaults to *1.0* which corresponds to no scaling.

dtype : tensorflow.DType, optional

Type of elements of *tensorflow.Tensor* objects used in this sampler. Defaults to *tensorflow.float64*.

mdecay : float, optional

(Constant) momentum decay per time-step. Defaults to *0.05*.

See also:

tensorflow_mcmc.sampling.mcmc_base_classes.BurnInMCMCSampler Base class for *SGHMCSampler* that specifies how actual sampling is performed (using iterator protocol, e.g. *next(sampler)*).

Examples

Simple, plain example: TODO: Add 2D Gaussian Case here

Simple example that uses batches: TODO: Add simplified batch example here

Stochastic Gradient Langevin Dynamics (SGLD)

class sgld.**SGLDSampler**(*params*, *cost_fun*, *seed=None*, *batch_generator=None*, *epsilon=0.01*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*, *burn_in_steps=3000*, *scale_grad=1.0*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *A=1.0*)

Bases: tensorflow_mcmc.sampling.mcmc_base_classes.BurnInMCMCSampler

Stochastic Gradient Langevin Dynamics Sampler that uses a burn-in procedure to adapt its own hyperparameters during the initial stages of sampling.

See [1] for more details on this burn-in procedure. See [2] for more details on Stochastic Gradient Langevin Dynamics.

[1] J. T. Springenberg, A. Klein, S. Falkner, F. Hutter Bayesian Optimization with Robust Bayesian Neural Networks. In Advances in Neural Information Processing Systems 29 (2016).

[2] M.Welling, Y. W. Teh Bayesian Learning via Stochastic Gradient Langevin Dynamics

__init__(*params*, *cost_fun*, *seed=None*, *batch_generator=None*, *epsilon=0.01*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*, *burn_in_steps=3000*, *scale_grad=1.0*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *A=1.0*)

Initialize the sampler parameters and set up a tensorflow.Graph for later queries.

Parameters **params** : list of tensorflow.Variable objects

Target parameters for which we want to sample new values.

cost_fun : callable

Function that takes *params* as input and returns a 1-d *tensorflow.Tensor* that contains the cost-value. Frequently denoted with *U* in literature.

seed : int, optional

Random seed to use. Defaults to *None*.

batch_generator : BatchGenerator, optional

Iterable which returns dictionaries to feed into tensorflow.Session.run() calls to evaluate the cost function. Defaults to *None* which indicates that no batches shall be fed.

epsilon : float, optional

Value that is used as learning rate parameter for the sampler, also denoted as discretization parameter in literature. Defaults to *0.01*.

session : tensorflow.Session, optional

Session object which knows about the external part of the graph (which defines *Cost*, and possibly batches). Used internally to evaluate (burn-in/sample) the sampler.

burn_in_steps: int, optional

Number of burn-in steps to perform. In each burn-in step, this sampler will adapt its own internal parameters to decrease its error. For reference see: TODO ADD PAPER REFERENCE HERE

scale_grad : float, optional

Value that is used to scale the magnitude of the noise used during sampling. In a typical batches-of-data setting this usually corresponds to the number of examples in the entire dataset.

A : float, optional

TODO XXX Doku Defaults to *1.0*.

See also:

`tensorflow_mcmc.sampling.mcmc_base_classes.BurnInMCMCSampler` Base class for *SGLDSampler* that specifies how actual sampling is performed (using iterator protocol, e.g. *next(sampler)*).

Examples

Simple, plain example: >>> import tensorflow as tf >>> session = tf.Session() >>> sampler = SGHMCSampler(params=[x], cost_fun=None, session=session) >>> first_sample = next(sampler) TODO: Add more samples

Simple example that uses batches: TODO: Add simplified batch example here

Relativistic Stochastic Gradient Hamiltonian Monte Carlo (RelSGHMC)

class relativistic_sghmc.**RelativisticSGHMCSampler**(*params*, *Cost*, *seed=None*, *epsilon=0.01*, *m=1.0*, *c=0.6*, *D=1.0*, *scale_grad=1.0*, *batch_generator=None*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*)

Bases: tensorflow_mcmc.sampling.mcmc_base_classes.MCMCSampler

__init__(*params*, *Cost*, *seed=None*, *epsilon=0.01*, *m=1.0*, *c=0.6*, *D=1.0*, *scale_grad=1.0*, *batch_generator=None*, *dtype=<Mock name='tensorflow.float64' id='139961115850736'>*, *session=<Mock name='tensorflow.get_default_session()' id='139961115851016'>*)

Relativistic Stochastic Gradient Hamiltonian Monte-Carlo Sampler.

See [1] for more details on Relativistic SGHMC.

[1] X. Lu, V. Perrone, L. Hasenclever, Y. W. Teh, S. J. Vollmer Relativistic Monte Carlo

Parameters **params** : list of tensorflow.Variable objects

Target parameters for which we want to sample new values.

Cost : tensorflow.Tensor

1-d Cost tensor that depends on *params*. Frequently denoted as U(theta) in literature.

seed : int, optional

Random seed to use. Defaults to *None*.

epsilon : float, optional

Value that is used as learning rate parameter for the sampler, also denoted as discretization parameter in literature. Defaults to *0.01*.

m : float, optional

mass constant. Defaults to *1.0*.

c : float, optional

"Speed of light constant" Defaults to *0.6*.

D : float, optional

Diffusion constant Defaults to *1.0*.

scale_grad : float, optional

Value that is used to scale the magnitude of the noise used during sampling. In a typical batches-of-data setting this usually corresponds to the number of examples in the entire dataset. Defaults to *1.0* which corresponds to no scaling.

batch_generator : BatchGenerator, optional

Iterable which returns dictionaries to feed into tensorflow.Session.run() calls to evaluate the cost function. Defaults to *None* which indicates that no batches shall be fed.

dtype : tensorflow.DType, optional

Type of elements of *tensorflow.Tensor* objects used in this sampler. Defaults to *tensorflow.float64*.

session : tensorflow.Session, optional

Session object which knows about the external part of the graph (which defines *Cost*, and possibly batches). Used internally to evaluate (burn-in/sample) the sampler.

CHAPTER 2

Bayesian Neural Network

An implementation of a **Bayesian Neural Network** that is trained using our SGMCMC sampling methods.

class bayesian_neural_network.**BayesianNeuralNetwork**(*sampling_method=<SamplingMethod.SGHMC: 'SGHMC'>*, *n_nets=100*, *learning_rate=0.001*, *mdecay=0.05*, *n_iters=50000*, *batch_size=20*, *burn_in_steps=1000*, *sample_steps=100*, *normalize_input=True*, *normalize_output=True*, *seed=None*, *get_net=<function get_default_net>*, *session=None*)

__init__(*sampling_method=<SamplingMethod.SGHMC: 'SGHMC'>*, *n_nets=100*, *learning_rate=0.001*, *mdecay=0.05*, *n_iters=50000*, *batch_size=20*, *burn_in_steps=1000*, *sample_steps=100*, *normalize_input=True*, *normalize_output=True*, *seed=None*, *get_net=<function get_default_net>*, *session=None*)

Bayesian Neural Networks use Bayesian methods to estimate the posterior distribution of a neural network's weights. This allows to also predict uncertainties for test points and thus makes Bayesian Neural Networks suitable for Bayesian optimization.

This module uses stochastic gradient MCMC methods to sample from the posterior distribution.

See [1] for more details.

[1] J. T. Springenberg, A. Klein, S. Falkner, F. Hutter Bayesian Optimization with Robust Bayesian Neural Networks. In Advances in Neural Information Processing Systems 29 (2016).

Parameters **sampling_method** : SamplingMethod, optional

Method used to sample networks for this BNN. Defaults to *SamplingMethod.SGHMC*.

n_nets: int, optional

Number of nets to sample during training (and use to predict). Defaults to *100*.

learning_rate: float, optional

Learning rate to use during sampling. Defaults to *1e-3*.

mdecay: float, optional

Momentum decay per time-step (parameter for SGHMCSampler). Defaults to *0.05*.

n_iters: int, optional

Total number of iterations of the sampler to perform. Defaults to *50000*

batch_size: int, optional

Number of datapoints to include in each minibatch. Defaults to *20* datapoints per minibatch.

burn_in_steps: int, optional

Number of burn-in steps to perform Defaults to *1000*.

sample_steps: int, optional

Number of sample steps to perform. Defaults to *100*.

normalize_input: bool, optional

Specifies whether or not input data should be normalized. Defaults to *True*

normalize_output: bool, optional

Specifies whether or not outputs should be normalized. Defaults to *True*

seed: int, optional

Random seed to use in this BNN. Defaults to *None*.

get_net: callable, optional

Callable that returns a network specification. Expected inputs are a *tensorflow.Placeholder* object that serves as feedable input to the network and an integer random seed. Expected return value is the networks final output. Defaults to *get_default_net*.

session: tensorflow.Session, optional

A *tensorflow.Session* object used to delegate computations performed in this network over to *tensorflow*. Defaults to *None* which indicates we should start a fresh *tensorflow.Session*.

__weakref__
list of weak references to the object (if defined)

compute_network_output(*params*, *input_data*)

Compute and return the output of the network when parameterized with *params* on *input_data*.

Parameters **params** : list of ndarray objects

List of parameter values (ndarray) for each tensorflow.Variable parameter of our network.

input_data : ndarray (N, D)

Input points to compute the network output for.

Returns network_output: ndarray (N,)

Output of the network parameterized with *params* on the given *input_data* points.

negative_log_likelihood(*X*, *Y*)

Compute the negative log likelihood of the current network parameters with respect to inputs *X* with labels *Y*.

Parameters **X** : tensorflow.Placeholder

Placeholder for input datapoints.

Y : tensorflow.Placeholder

Placeholder for input labels.

Returns neg_log_like: tensorflow.Tensor

Negative log likelihood of the current network parameters with respect to inputs *X* with labels *Y*.

mse: tensorflow.Tensor

Mean squared error of the current network parameters with respect to inputs *X* with labels *Y*.

To discretize possible user choices for the method used to train a Bayesian Neural Network, we maintain an Enum class called **SamplingMethod**.

The Enum class also provides facilities to obtain a supported sampler directly. To obtain a sampler, it is enough to call *SamplingMethod.get_sampler(sampling_method, **sampler_args)* with a supported *sampling_method* and corresponding keyword arguments in *sampler_args*.

class bayesian_neural_network.**SamplingMethod**

Enumeration type for all sampling methods we support.

static get_sampler(*sampling_method*, ***sampler_args*)

TODO: Docstring for get_sampler.

Parameters **sampling_method** : SamplingMethod

Enum corresponding to sampling method to return a sampler for.

****sampler_args** : dict

Keyword arguments that contain all input arguments to the desired the constructor of the sampler for the specified *sampling_method*.

Returns **sampler** : Subclass of *sampling.MCMCSampler*

A sampler instance that implements the specified *sampling_method* and is initialized with inputs *sampler_args*.

static is_supported(*sampling_method*)

Static method that returns true if *val* is a supported sampling method (e.g. there is an entry for it in *SamplingMethod* enum).

Examples

Supported sampling methods give *True*:

```
>>> SamplingMethod.is_supported(SamplingMethod.SGHMC)
True
```

Other input types give *False*:

```
>>> SamplingMethod.is_supported(0)
False
>>> SamplingMethod.is_supported("test")
False
```

This module contains our implementation of priors for the weights and log variance of our **Bayesian Neural Network**.

class bnn_priors.**LogVariancePrior**(*mean=0.01*, *var=2*)
Prior on the log predicted variance.

__init__(*mean=0.01*, *var=2*)
Initialize prior for a given *mean* and *variance*.

Parameters **mean** : float, optional

Actual mean on a linear scale. Default value is '10e-3'.

var : float, optional

Variance on a log scale. Default value is '2'.

__weakref__
list of weak references to the object (if defined)

log_like(*log_var*)
Compute the log likelihood of this prior for a given input.

Parameters **log_var: tensorflow.Tensor**

Returns log_like_output: tensorflow.Tensor

class bnn_priors.**WeightPrior**
Prior on the weights.

__init__()
Initialize weight prior with weight decay initialized to *1*.

__weakref__
list of weak references to the object (if defined)

log_like(*params*)
Compute the log log likelihood of this prior for a given input.

Parameters **params** : list of tensorflow.Variable objects

Returns log_like: tensorflow.Tensor

CHAPTER 3

Tensor Utils

This module contains a couple of util functions to facilitate working with tensors in numerical computations.

`tensor_utils.`**`pdist`**(*tensor*, *metric='euclidean'*)

Pairwise distances between observations in n-dimensional space. Ported from *scipy.spatial.distance.pdist* @2f5aa264724099c03772ed784e7a947d2bea8398 for cherry-picked distance metrics.

Parameters **tensor** : *tensorflow.Tensor*

metric : *DistanceMetric*, optional

Pairwise metric to apply. Defaults to *DistanceMetric.Euclidean*.

Returns **Y** : *tensorflow.Tensor*

Returns a condensed distance matrix *Y* as *tensorflow.Tensor*. For each i and j (where $i < j < m$), where m is the number of original observations. The metric `dist(u=X[i], v=X[j])` is computed and stored in entry `j` of subtensor `Y[j]`.

Examples

Gives equivalent results to *scipy.spatial.distance.pdist* but uses *tensorflow.Tensor* objects: >>> import tensorflow as tf >>> import numpy as np >>> from scipy.spatial.distance import pdist as pdist_scipy >>> input_scipy = np.array([[0.77228064, 0.09543156], [0.3918973 , 0.96806584], [0.66008144, 0.22163063]]) >>> result_scipy = pdist_scipy(input_scipy, metric="euclidean") >>> session = tf.Session() >>> input_tensorflow = tf.constant(input_scipy) >>> result_tensorflow = session.run(pdist(input_tensorflow, metric="euclidean")) >>> np.allclose(result_scipy, result_tensorflow) True

Will raise a *NotImplementedError* for unsupported metric choices: >>> import tensorflow as tf >>> import numpy as np >>> input_scipy = np.array([[0.77228064, 0.09543156], [0.3918973 , 0.96806584], [0.66008144, 0.22163063]]) >>> session = tf.Session() >>> input_tensorflow = tf.constant(input_scipy) >>> session.run(pdist(input_tensorflow, metric="lengthy_metric")) Traceback (most recent call last):

...

NotImplementedError: tensor_utils.pdist: Metric 'lengthy_metric' currently not supported!

Like *scipy.spatial.distance.pdist*, we fail for input that is not 2-d: >>> import tensorflow as tf >>> import numpy as np >>> input_scipy = np.random.rand(2, 2, 1) >>> session = tf.Session() >>> input_tensorflow = tf.constant(input_scipy) >>> session.run(pdist(input_tensorflow, metric="lengthy_metric")) Traceback (most recent call last):

...

ValueError: tensor_utils.pdist: A 2-d tensor must be passed.

`tensor_utils.`**`safe_divide`**(*x*, *y*, *small_constant=1e-16*, *name=None*)

tf.divide(x, y)* after adding a small appropriate constant to *y in a smart way so that we can avoid division-by-zero artefacts.

Parameters **x** : *tensorflow.Tensor*

Left-side operand of *tensorflow.divide*

y : *tensorflow.Tensor*

Right-side operand of *tensorflow.divide*

small_constant : *tensorflow.Tensor*

Small constant tensor to add to/subtract from *y* before computing *x / y* to avoid division-by-zero.

name : *string* or *NoneType*, optional

Name of the resulting node in a *tensorflow.Graph*. Defaults to *None*.

Returns **division_result** : *tensorflow.Tensor*

Result of division *tf.divide(x, y)* after applying clipping to *y*.

Examples

Will safely avoid divisions-by-zero under normal circumstances:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> session = tf.Session()
>>> x = tf.constant(1.0)
>>> nan_tensor = tf.divide(x, 0.0)  # will produce "inf" due to division-by-zero
>>> np.isinf(nan_tensor.eval(session=session))
True
>>> z = safe_divide(x, 0., small_constant=1e-16)  # will avoid "inf" due to␣
↪division-by-zero by clipping
>>> np.isinf(z.eval(session=session))
False
```

To see that simply adding a constant may fail, but this implementation handles those corner cases correctly, consider this example:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> x, y = tf.constant(1.0), tf.constant(-1e-16)
>>> small_constant = tf.constant(1e-16)
>>> v1 = x / (y + small_constant)  # without sign
```

```
>>> v2 = safe_divide(x, y, small_constant=small_constant) # with sign
>>> val1, val2 = session.run([v1, v2])
>>> np.isinf(val1) # simply adding without considering the sign can still yield
↪"inf"
True
>>> np.isinf(val2)  # our version behaves appropriately
False
```

tensor_utils.**safe_sqrt**(*x*, *clip_value_min=0.0*, *clip_value_max=inf*, *name=None*)

Computes *tf.sqrt(x)* after clipping tensor *x* using *tf.clip_by_value(x, clip_value_min, clip_value_max)* to avoid square root (e.g. of negative values) artefacts.

Parameters **x** : *tensorflow.Tensor* or *tensorflow.SparseTensor*

Operand of *tensorflow.sqrt*.

clip_value_min : 0-D (scalar) *tensorflow.Tensor*, optional

The minimum value to clip by. Defaults to *0*

clip_value_max : 0-D (scalar) *tensorflow.Tensor*, optional

The maximum value to clip by. Defaults to *float("inf")*

name : *string* or *NoneType*, optional

Name of the resulting node in a *tensorflow.Graph*. Defaults to *None*.

Returns sqrt_result: *tensorflow.Tensor*

Result of square root *tf.sqrt(x)* after applying clipping to *x*.

Examples

Will safely avoid square root of negative values:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> x = tf.constant(-1e-16)
>>> z = tf.sqrt(x)  # fails, results in 'nan'
>>> z_safe = safe_sqrt(x)  # works, results in '0'
>>> session = tf.Session()
>>> z_val, z_safe_val = session.run([z, z_safe])
>>> np.isnan(z_val)  # ordinary tensorflow computation gives 'nan'
True
>>> np.isnan(z_safe_val) # `safe_sqrt` produces '0'.
False
>>> z_safe_val
0.0
```

tensor_utils.**squareform**(*tensor*)

TODO Ported from *scipy.spatial.distance.squareform* @2f5aa264724099c03772ed784e7a947d2bea8398, but supports only 1-d (vector) input

Parameters **tensor** : *tensorflow.Tensor*

Returns **redundant_distance_tensor** : *tensorflow.Tensor*

Examples

May be used in conjunction with *tensor_utils.pdist* to obtain a redundant distance matrix: >>> import tensorflow as tf >>> import numpy as np >>> from scipy.spatial.distance import pdist as scipy_pdist, squareform as scipy_squareform >>> original_input = np.random.rand(2, 4) >>> tf_redundant_distance_tensor = squareform(pdist(tf.constant(original_input))) >>> scipy_redundant_distance_matrix = scipy_squareform(scipy_pdist(original_input)) >>> session = tf.Session() >>> tf_redundant_distance_matrix = session.run(tf_redundant_distance_tensor) >>> np.allclose(tf_redundant_distance_matrix, scipy_redundant_distance_matrix) True

Contrary to *scipy.spatial.squareform*, conversion of 2D input to a condensed distance vector is *not* supported: >>> import numpy as np >>> import tensorflow as tf >>> illegal_input = tf.constant(np.random.rand(4, 4)) >>> squareform(illegal_input) Traceback (most recent call last):

...

NotImplementedError: tensor_utils.squareform: Only 1-d (vector) input is supported!

`tensor_utils.`**`unvectorize`**(*tensor*, *original_shape*)

Reshape previously vectorized *tensor* back to its *original_shape*. Essentially the inverse transformation as the one performed by *tensor_utils.vectorize*.

Parameters **tensor** : *tensorflow.Variable* object or *tensorflow.Tensor* object

Input tensor to unvectorize.

original_shape : *tensorflow.Shape*

Original shape of *tensor* prior to its vectorization.

Returns **tensor_unvectorized** : *tensorflow.Tensor* object

Tensor with the same values as *tensor* but reshaped back to shape *original_shape*.

Examples

Function *unvectorize* undoes the work done by *vectorize*:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> t1 = tf.constant([[12.0, 14.0, -3.0], [4.0, 3.0, 1.0], [9.0, 2.0, 4.0]])
>>> t2 = unvectorize(vectorize(t1), original_shape=t1.shape)
>>> session = tf.Session()
>>> t1_array, t2_array = session.run([t1, t2])
>>> np.allclose(t1_array, t2_array)
True
```

It will also work for *tensorflow.Variable* objects, but will return *tensorflow.Tensor* as unvectorized output.

```
>>> import tensorflow as tf
>>> import numpy as np
>>> v = tf.Variable([[0.0, 1.0], [2.0, 0.0]])
>>> session = tf.Session()
>>> session.run(tf.global_variables_initializer())
>>> t = unvectorize(vectorize(v.initialized_value()), original_shape=v.shape)
>>> v_array, t_array = session.run([v, t])
>>> np.allclose(t_array, v_array)
True
```

tensor_utils.**vectorize**(*tensor*)

Turn any matrix into a long vector for the parameters by expanding it. Turn: [[a, b], [c, d]] into [a, b, c, d]

For vector inputs, this simply returns a copy of the vector.

For reference see also *vec*-operator in: https://hec.unil.ch/docs/files/23/100/handout1.pdf#page=2

Parameters **tensor** : *tensorflow.Variable* object or *tensorflow.Tensor* object

Input tensor to vectorize.

Returns tensor_vectorized: *tensorflow.Variable* object or *tensorflow.Tensor* object

Vectorized result for input *tensor*.

Examples

A *tensorflow.Variable* can be vectorized: (NOTE: the returned vectorized variable must be initialized to use it in *tensorflow* computations.)

```
>>> import tensorflow as tf
>>> v1 = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
>>> v1_vectorized = vectorize(v1)
>>> session = tf.Session()
>>> session.run(tf.global_variables_initializer())
>>> session.run(v1_vectorized)
array([[ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.],
       [ 6.]], dtype=float32)
```

A normal *tensorflow.Tensor* can be vectorized:

```
>>> import tensorflow as tf
>>> t1 = tf.constant([[12.0, 14.0, -3.0], [4.0, 3.0, 1.0], [9.0, 2.0, 4.0]])
>>> t1_vectorized = vectorize(t1)
>>> session = tf.Session()
>>> session.run(t1_vectorized)
array([[ 12.],
       [ 14.],
       [ -3.],
       [  4.],
       [  3.],
       [  1.],
       [  9.],
       [  2.],
       [  4.]], dtype=float32)
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

bayesian_neural_network, 13
bnn_priors, 16

m

mcmc_base_classes, 3

r

relativistic_sghmc, 11

s

sampling, 1
sghmc, 8
sgld, 9

t

tensor_utils, 17

Index

Symbols

__init__() (bayesian_neural_network.BayesianNeuralNetwork method), 13
__init__() (bnn_priors.LogVariancePrior method), 16
__init__() (bnn_priors.WeightPrior method), 16
__init__() (mcmc_base_classes.BurnInMCMCSampler method), 6
__init__() (mcmc_base_classes.MCMCSampler method), 3
__init__() (relativistic_sghmc.RelativisticSGHMCSampler method), 11
__init__() (sghmc.SGHMCSampler method), 8
__init__() (sgld.SGLDSampler method), 9
__iter__() (mcmc_base_classes.MCMCSampler method), 4
__metaclass__ (mcmc_base_classes.BurnInMCMCSampler attribute), 7
__metaclass__ (mcmc_base_classes.MCMCSampler attribute), 4
__next__() (mcmc_base_classes.BurnInMCMCSampler method), 7
__next__() (mcmc_base_classes.MCMCSampler method), 4
__weakref__ (bayesian_neural_network.BayesianNeuralNetwork attribute), 14
__weakref__ (bnn_priors.LogVariancePrior attribute), 16
__weakref__ (bnn_priors.WeightPrior attribute), 16
__weakref__ (mcmc_base_classes.MCMCSampler attribute), 5
_draw_noise_sample() (mcmc_base_classes.MCMCSampler method), 5
_next_batch() (mcmc_base_classes.MCMCSampler method), 5
_uninitialized_params() (mcmc_base_classes.MCMCSampler method), 6

B

bayesian_neural_network (module), 13
BayesianNeuralNetwork (class in bayesian_neural_network), 13
bnn_priors (module), 16
BurnInMCMCSampler (class in mcmc_base_classes), 6

C

compute_network_output() (bayesian_neural_network.BayesianNeuralNetwork method), 14

G

get_sampler() (bayesian_neural_network.SamplingMethod static method), 15

I

is_burning_in (mcmc_base_classes.BurnInMCMCSampler attribute), 7
is_supported() (bayesian_neural_network.SamplingMethod static method), 15

L

log_like() (bnn_priors.LogVariancePrior method), 16
log_like() (bnn_priors.WeightPrior method), 16
LogVariancePrior (class in bnn_priors), 16

M

mcmc_base_classes (module), 3
MCMCSampler (class in mcmc_base_classes), 3

N

negative_log_likelihood() (bayesian_neural_network.BayesianNeuralNetwork method), 14

P

pdist() (in module tensor_utils), 17

R

relativistic_sghmc (module), 11

RelativisticSGHMCSampler (class in relativistic_sghmc), 11

S

safe_divide() (in module tensor_utils), 18
safe_sqrt() (in module tensor_utils), 19
sampling (module), 1
SamplingMethod (class in bayesian_neural_network), 15
sghmc (module), 8
SGHMCSampler (class in sghmc), 8
sgld (module), 9
SGLDSampler (class in sgld), 9
squareform() (in module tensor_utils), 19

T

tensor_utils (module), 16, 17

U

unvectorize() (in module tensor_utils), 20

V

vectorize() (in module tensor_utils), 20

W

WeightPrior (class in bnn_priors), 16