gmmvi Documentation

Release 0.0.0

<anonymized>

Feb 24, 2023

CONTENTS

1 About **3**
- 1.1 Method . . . 3
- 1.2 Design Choices . . . 3
- 1.3 Naming Convention . . . 5

2 Get Started **7**
- 2.1 Installation . . . 7
- 2.2 Usage . . . 7

3 API Reference **13**
- 3.1 Models . . . 13
- 3.2 Experiments . . . 23
- 3.3 Optimization . . . 33

4 References **67**

Bibliography **69**

Python Module Index **71**

Index **73**

Welcome to the documentation for GMMVI - a framework for optimizing Gaussian mixture models for variational inference.

If you don't quite know how you ended up here, check out the *About* page, for more details on the problem setting and the underlying algorithms.

Just wanna know how do *install* and *use* the framework? *Get Started*!

For a look under the hood, you can check the *API Reference* or directly dig into the source.

CHAPTER

ONE

ABOUT

GMMVI (Gaussian Mixture Model Variational Inference) is a framework for optimizing a Gaussian mixture model $q(\mathbf{x})$ with Gaussian components $q(\mathbf{x}|o)$ and weights $q(o)$,

$$q(\mathbf{x}) = \sum_o q(o)q(\mathbf{x}|o),$$

with respect to the optimization problem,

$$\max_{q(\mathbf{x})} \mathbb{E}_{q(\mathbf{x})}\left[r(\mathbf{x})\right] + \mathrm{H}(q(\mathbf{x})),$$

where $\mathrm{H}(q(\mathbf{x}))$ denotes the mixture's entropy and $r(\mathbf{x})$ assigns a reward to the sample $\mathbf{x}$. If $r(\mathbf{x})$ corresponds to the energy of a Gibbs-Boltzmann distribution $p(\mathbf{x}) \propto \exp(r(\mathbf{x}))$, the learned GMM will approximate the target distribution $p(\mathbf{x})$ by minimizing the reverse Kullback-Leibler divergence $\mathrm{KL}\left(q(\mathbf{x})||p(\mathbf{x})\right)$.

The optimization is performed with respect to the weights, means and covariance matrices, and if desired the number of components. The framework is build on Tensorflow 2, however, the reward function can also be implemented using different libraries, such as PyTorch.

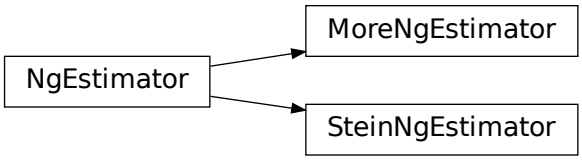
1.1 Method

The optimization is performed iteratively, where at every iteration an independent natural gradient descent step is performed to the categorical distribution over weights, $q(o)$, and to each individual component $q(\mathbf{x}|o)$. This procedure was concurrently proposed by Arenz *et al.* [AZN18] and Lin *et al.* [LKS19a]. However, both approaches differ quite significantly in several design choices (e.g. how the natural gradients are estimated) and derived the procedure from different perspectives with different theoretical guarantees, and therefore the equivalence of both approaches was initially not understand. This framework is published along with the article that first established the close connection between both approaches, and was used to systematically evaluate the effects of the different design choices.

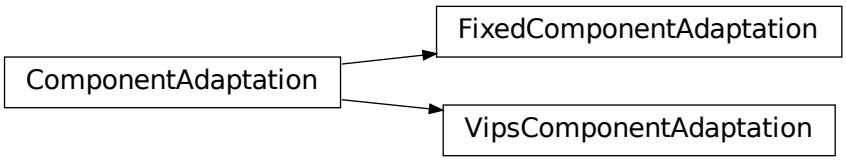
1.2 Design Choices

We distinguish design choices for seven different *modules* corresponding to the abstract classes, where for each design choice, there are two to three option implemented as concrete classes.

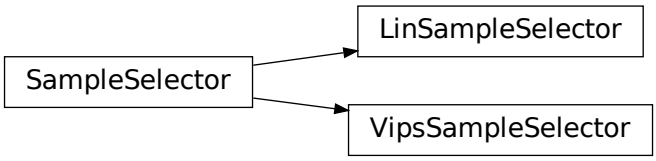
1. ***NgEstimator***



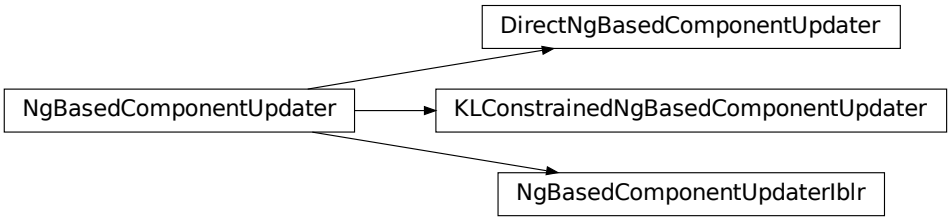
2. *ComponentAdaptation*



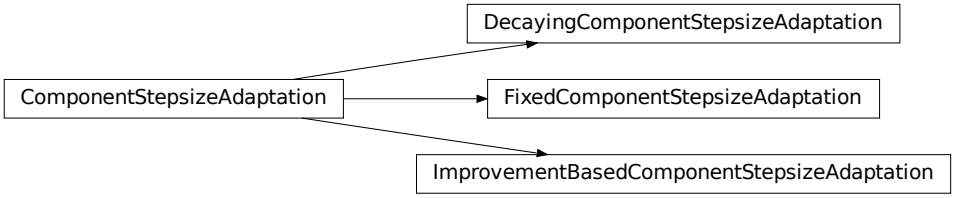
3. *SampleSelector*



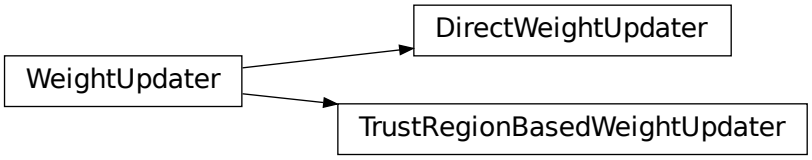
4. *NgBasedComponentUpdater*



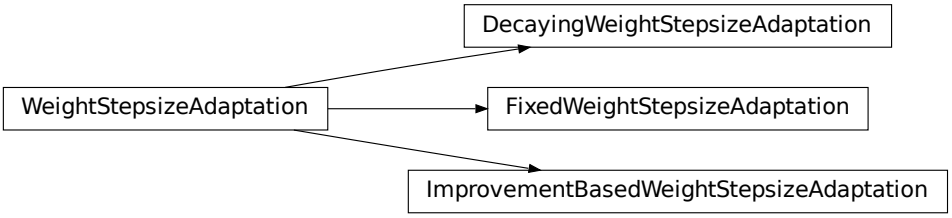
5. *ComponentStepsizeAdaptation*



6. ***WeightUpdater***



7. ***WeightStepsizeAdaptation***



1.3 Naming Convention

Depending on which option is used for each design choice, there are currently 432 different instantiation supported by GMMVI. When referring to a specific instantiation, we use 7-letter codewords, where the presence of a letter implies, that the corresponding option was chosen. The mapping from letter to option is given in the following table:

Module	Options		
NgEstimator	*MORE (Z)*	*Stein (S)*	
ComponentAdaptation	*Fixed (E)*	*VIPS (A)*	
SampleSelector	*Lin (P)*	*VIPS (M)*	
NgBasedComponentUpdater	*Direct (I)*	*iBLR (Y)*	*Trust-Region (T)*
ComponentStepsizeAdaptation	*Fixed (F)*	*Decaying (D)*	*Adaptive (R)*
WeightUpdater	*Direct (U)*	*Trust-Region (O)*	
WeightStepsizeAdaptation	*Fixed (X)*	*Decaying (G)*	*Adaptive (N)*

Using this naming convention, ZAMTRUX refers to VIPS [AZN20], and SEPIFUX refers to the method by Lin *et al.* [LKS19a]. The recommended setting, however, is SAMTRON.

CHAPTER

TWO

GET STARTED

2.1 Installation

To install GMMVI (optionally) create a virtual environment and run

```
(.venv) $ pip install .
```

2.2 Usage

For performing the optimization, you can directly instantiate a *GMMVI* and run *GMMVI.train_iter()* in a loop, or, for adding basic logging capability and easier integration, for example with WandB, you can instantiate a *GmmviRunner* and run *GmmviRunner.iterate_and_log(n)* in a loop.

2.2.1 Directly Using GMMVI

Before instantiating the *GMMVI*, we need to create several other objects, namely:

1. A *wrapped model* which stores the parameters of the GMM, as well as component-specific meta-information (reward histories, learning-rates, etc.)
2. A *SampleDB* for storing samples.
3. One object for each of the seven *Design Choices*.

Fortunately, each of these classes and also *GMMVI* itself, have a static method called build_from_config(), which allows to create the object from a common config dictionary (which can be created from a YAML file). Using a common dictionary is recommended, to ensure that the parameters passed to the different constructors are consistent (e.g. the sample dimensions needs to be the same).

It is easiest to directly use *GMMVI.build_from_config*, which will automatically construct most of the required objects. However, you still need to pass

1. the dictionary containing the hyperparameters,
2. the *target distribution*,
3. and the *initial model*.

The following example script directly uses *GMMVI* using the hyperparameters from the following YAML file: examples/example_config.yml.

```
import os
import logging
# Tensorflow may give warnings when the Cholesky decomposition fails.
# However, these warning can usually be ignored because the NgBasedOptimizer
# will handle them by rejecting the update and decreasing the stepsize for
# the failing component. To keep the console uncluttered, we suppress warnings.
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'  # ERROR
logging.getLogger('tensorflow').setLevel(logging.ERROR)
import tensorflow as tf

from gmmvi.optimization.gmmvi import GMMVI
from gmmvi.configs import load_yaml
from gmmvi.experiments.target_distributions.logistic_regression import make_breast_cancer
from gmmvi.models.full_cov_gmm import FullCovGMM
from gmmvi.models.gmm_wrapper import GmmWrapper
from gmmvi.experiments.setup_experiment import construct_initial_mixture

#For creating a GMMVI object using GMMVI.build_from_config, we need:
# 1. A dictionary containing the hyperparameters
my_path = os.path.dirname(os.path.realpath(__file__))
config = load_yaml(os.path.join(my_path, "example_config.yml"))

# 2. A target distribution
target_distribution = make_breast_cancer()

# 3. An (wrapped) initial model
dims = target_distribution.get_num_dimensions()
initial_weights = tf.ones(1, tf.float32)
initial_means = tf.zeros((1, dims), tf.float32)
initial_covs = tf.reshape(100 * tf.eye(dims), [1, dims, dims])
model = FullCovGMM(initial_weights, initial_means, initial_covs)
# Above config contains a section model_initialization, and, therefore,
# we could also create the initial model using:
# model = construct_initial_mixture(dims, **config["model_initialization"])
wrapped_model = GmmWrapper.build_from_config(model=model, config=config)


# Now we can create the GMMVI object and start optimizing
gmmvi = GMMVI.build_from_config(config=config,
                                target_distribution=target_distribution,
                                model=wrapped_model)
max_iter = 1001
for n in range(max_iter):
    gmmvi.train_iter()

    if n % 100  == 0:
        samples = gmmvi.model.sample(1000)[0]
        elbo = tf.reduce_mean(target_distribution.log_density(samples)
                              - model.log_density(samples))
        print(f"{n}/{max_iter}: "
              f"The model now has {gmmvi.model.num_components} components "
              f"and an elbo of {elbo}.")
```

The script can be found under examples/1_directly_using_gmmvi.py.

2.2.2 Using the GmmviRunner

The *GmmviRunner* wraps around *GMMVI* to add logging capabilities. Furthermore, the *GmmviRunner* takes care of initializing the model and the target distribution (when using one of the provided target distributions). Hence, we only need to provide the config to create it, as shown by the following script:

```
import os
from gmmvi.gmmvi_runner import GmmviRunner
from gmmvi.configs import load_yaml

my_path = os.path.dirname(os.path.realpath(__file__))
config = load_yaml(os.path.join(my_path, "example_config.yml"))
gmmvi_runner = GmmviRunner.build_from_config(config)

for n in range(10001):
    gmmvi_runner.iterate_and_log(n)
```

The script can be found under examples/2_using_the_gmmvi_runner.py.

2.2.3 Using the GmmviRunner with Default Configs

We can also directly create a default config based on the 7-letter Codeword to specify the design choices, thereby, not requiring an external YAML file:

```
from gmmvi.gmmvi_runner import GmmviRunner
import gmmvi.configs

# In this example, we will create the config for a GmmviRunner using default configs
# for a given Codename (we weill use SAMYROX) and an and an environment name
# (we will use GMM20).
# Let's first get the default config for SAMYROX
algorithm_config = gmmvi.configs.get_default_algorithm_config("SAMYROX")

# Internally, this loaded the yaml files in gmmvi/configs/module_configs corresponding
# to the chosen design choices and stored them in a single dict "algorithm_config".
# Note that these default values were chosen independently for every design choice,
# and, thus, may not always be sensible. For example, the initial_stepsize defined in
# gmmvi/configs/module_configs/component_stepsize_adaptation/improvement_based.yml
# (Codeletter "R") is suitable if the stepsize is treated as a trust-region
# (Codeletter "T"), but not if it directly corresponds to the stepsize
# (Codeletter "I" or "Y")! Hence, we will overwrite the stepsize to something more
# suitable for SAMYROX:
better_stepsize_config = {
   'initial_stepsize': 0.0001,
   'min_stepsize': 0.0001,
   'max_stepsize': 0.001
}
algorithm_config = gmmvi.configs.update_config(algorithm_config, better_stepsize_config)
```

(continues on next page)

(continued from previous page)

```
# We will use a target distribution that was shipped with the framework, namely "gmm20":
environment_config = gmmvi.configs.get_default_experiment_config("gmm20")

# The last call searched configs/experiment_configs for a corresponding yml-file and␣
↪found
# gmm20.yml and stored the config in the dictionary "environment_config". We now just␣
↪need
# to merge both config files:
config = gmmvi.configs.update_config(algorithm_config, environment_config)

# Create the GmmviRunner and start optimizing.
gmmvi_runner = GmmviRunner.build_from_config(config=config)
for n in range(1500):
    gmmvi_runner.iterate_and_log(n)
```

The script can be found under examples/3_gmmvi_runner_with_default_configs.py.

2.2.4 Using the GmmviRunner with Custom Environments

We can still use the *GmmviRunner* with custom environments, but we need to store the *target distribution object* in the config:

```
from gmmvi.gmmvi_runner import GmmviRunner
from gmmvi.configs import get_default_algorithm_config, update_config
import tensorflow as tf
import numpy as np
import matplotlib
matplotlib.use("tkAgg")
import matplotlib.pyplot as plt

# For creating a custom environment, we need to extend
# gmmvi.experiments.target_distributions.lnpdf.LNPDF:
from gmmvi.experiments.target_distributions.lnpdf import LNPDF
class Rosenbrock(LNPDF):
    """ We treat the negative Rosenbrock function as unnormalized target distribution.
    We implement it in numpy and do not allow GMMVI to backpropagate through log_
↪density().
    As we want to use Stein's Lemma for estimating the natural gradient (Codeletter "S"),
    we need to implement the gradient ourselves, and, therefore, we set
    use_log_density_and_grad=True and implement the corresponding method.
    """
    def __init__(self):
        super(Rosenbrock, self).__init__(use_log_density_and_grad=True,
                                         safe_for_tf_graph=False)
        self.a = 1
        self.b = 100

    def get_num_dimensions(self) -> int:
        return 2

    def log_density(self, samples: tf.Tensor) -> tf.Tensor:
```

(continues on next page)

(continued from previous page)

```
        x = samples[:, 0].numpy().astype(np.float32)
        y = samples[:, 1].numpy().astype(np.float32)
        my_log_density = -((self.a - x)**2 + self.b * (y - x**2)**2)
        return tf.convert_to_tensor(my_log_density, dtype=tf.float32)

    def log_density_and_grad(self, samples: tf.Tensor) -> tf.Tensor:
        x = samples[:, 0].numpy().astype(np.float32)
        y = samples[:, 1].numpy().astype(np.float32)
        my_log_density = -((self.a - x)**2 + self.b * (y - x**2)**2)
        my_grad_x = -(-2 * (self.a - x) - 4 * self.b * (y - x**2) * x)
        my_grad_y = -(2 * self.b * (y - x**2))
        my_grad = np.vstack((my_grad_x, my_grad_y)).T
        return [tf.convert_to_tensor(my_log_density, dtype=tf.float32),
                tf.convert_to_tensor(my_grad, dtype=tf.float32)]

# We can also use the GmmviRunner, when using custom environments, but we have
# to put the LNPDF object into the dict. Furthermore, we need to define the other
# environment-specific settings that would otherwise be defined in
# the corresponding config in gmmvi/config/experiment_configs:
environment_config = {
    "target_fn": Rosenbrock(),
    "start_seed": 0,
    "environment_name": "Rosenbrock",
    "model_initialization": {
        "use_diagonal_covs": False,
        "num_initial_components": 1,
        "prior_mean": 0.,
        "prior_scale": 1.,
        "initial_cov": 1.,
    },
    "gmmvi_runner_config": {
        "log_metrics_interval": 100
    },
    "use_sample_database": True,
    "max_database_size": int(1e6),
    "temperature": 1.
}

# We will again use the automatically generated config for the algorithm,
# but this time, we will use "SAMTRUX". The default settings are reasonable for
# SAMTRUX, so we do not make any modifications to the hyperparameters.
algorithm_config = get_default_algorithm_config("SAMTRUX")

# Now we just need to merge the configs and use GmmviRunner as before:
merged_config = update_config(algorithm_config, environment_config)
gmmvi_runner = GmmviRunner.build_from_config(merged_config)

for n in range(500):
    gmmvi_runner.iterate_and_log(n)

# Plot samples from our "Rosenbrock-distribution"
test_samples = gmmvi_runner.gmmvi.model.sample(10000)[0]
```

(continues on next page)

(continued from previous page)

```
plt.plot(test_samples[:, 0], test_samples[:, 1], 'x')
plt.show()
plt.pause(0.1)
```

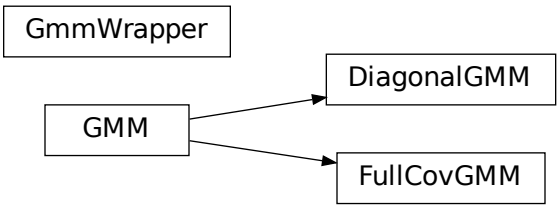
The script can be found under examples/4_gmmvi_runner_with_custom_environments.py.

CHAPTER

THREE

API REFERENCE

3.1 Models



class gmmvi.models.gmm.**GMM**(*log_weights: Variable*, *means: Variable*, *chol_covs: Variable*)

An abstract class for Gaussian mixture models (GMMs).

This class stores the parameters of a GMM (weights, means and Cholesky matrices) and provides functionality that is common for different types of GMMs (e.g., GMMs with full covariance matrices, and those with diagonal covariance matrices). For example, this class provides methods for sampling the *GMM*, evaluating its probability density function, and entropy, while relying on the subclass for sampling the *components*, etc.

Parameters

- **log_weights** – tf.Variable(float) A tensorflow Variable for storing the log-probabilities of the component weights.
- **means** – tf.Variable(float) A tensorflow Variable for storing the component means (number of components X number of dimensions)
- **chol_covar** – tf.Variable(float) A tensorflow Variable for storing the Cholesky matrix of the component's covariance matrix. The first dimension specifies the index of the components. The rank may vary depending on the subclass. For example, when storing the Cholesky matrix for a diagonal covariance matrix, it is possible to use a rank-2 Tensor (number of components X number of dimensions) for better memory efficiency.

add_component(*initial_weight: Tensor*, *initial_mean: Tensor*, *initial_cov: Tensor*)

Add a component to the mixture model. The weights will be automatically normalized.

Parameters

- **initial_weight** – tf.Tensor The weight of the new component (before re-normalization)

- **initial_mean** – tf.Tensor The mean of the new component
- **initial_cov** – tf.Tensor The covariance matrix of the new component

component_entropies() → Tensor

Computes the entropy of each component.

Returns
a one-dimensional tensor of shape num_components containing the component entropies.

Return type
tf.Tensor

component_log_densities(*samples: Tensor*) → Tensor

Evaluate the log densities for each mixture component on the given samples.

Parameters
samples – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate.

Returns
a two-dimensional tensor of size number_of_components x number_of_samples, containing the log-densities for each component.

Return type
tf.Tensor

component_log_density(*index: int*, *samples: Tensor*) → Tensor

Use the specified component to evaluate the Gaussian log-density at the given samples.

Parameters

- **index** – int The index of the component of which we want to compute the log densities.
- **samples** – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate.

Returns
a one-dimensional tensor of size number_of_samples, containing the log-densities.

Return type
tf.Tensor

component_log_density_and_grad(*index: int*, *samples: Tensor*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Evaluates for the given component the log-density and its gradient.

Parameters

- **samples** – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate.
- **Returns** – **component_log_densities** - a one-dimensional tf.Tensor of shape num_samples containing the log-densities of the given component.

  component_log_density_grads - a two-dimensional tf.Tensor of shape num_samples x num_dimensions containing the gradients of the component's log-densities.

component_marginal_log_densities(*samples: Tensor*, *dimension: int*) → Tensor

Evaluate the marginal log densities for each mixture component along the given dimension for each sample.

Parameters

- **samples** – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate. Note that for providing an easier interface, each sample has the number of dimensions compatible with this GMM, although only a single entry is actually used for evaluating the marginal density.
- **dimension** – int The dimension of interest.

Returns
a two-dimensional tensor of size number_of_components x number_of_samples, containing the marginal log-densities for each component.

Return type
tf.Tensor

property covs: Tensor

Returns: tf.Tensor: the covariance matrices as a 3-dimensional tensor of shape num_components x num_dimensions x num_dimensions

density(*samples: Tensor*) → Tensor

Evaluates the given samples on this GMM.

Parameters
samples – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate

Returns
a one-dimensional tensor of shape num_samples containing the model densities.

Return type
tf.Tensor

gaussian_entropy(*chol: Tensor*) → Tensor

Computes the entropy of Gaussian distribution with the given Cholesky matrix.

Parameters
chol – tf.Tensor A two-dimensional tensor of shape number_of_dimensions x number_of_dimensions specifying the Cholesky matrix

Returns
The entropy

Return type
tf.float32

get_average_entropy() → tf.float32

Averages the entropies of the individual components based on their respective weights.

Returns
the average component entropy

Return type
tf.float32

log_densities_also_individual(*samples: Tensor*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Evaluates the given samples on this GMM, but also returns individual log-densities for each Gaussian component.

Parameters
samples – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate

Returns

model_log_densities - a one-dimensional tnsor of shape num_samples containing the model log-densities.

component_log_densities - a two-dimensional tensor of shape num_components x num_samples containing the component log-densities

Return type
tuple(tf.Tensor, tf.Tensor)

`log_density`(*samples: Tensor*) → Tensor

Evaluates the given samples on this GMM.

Parameters
samples – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate.

Returns
a one-dimensional tensor of shape num_samples containing the model log-densities.

Return type
tf.Tensor

`log_density_and_grad`(*samples: Tensor*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Evaluates the given samples on this GMM, returns the log-densities of the whole model, their gradients, and also the individual log-densities for each Gaussian component.

Parameters
samples – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate.

Returns

model_log_densities - a one-dimensional tensor of shape num_samples containing the model log-densities.

model_log_density_grads - a two-dimensional tf.Tensor of shape num_samples x num_dimensions containing the gradients of the model log-densities.

component_log_densities - a two-dimensional tf.Tensor of shape num_components x num_samples containing the component log-densities.

Return type
tuple(tf.Tensor, tf.Tensor)

`marginal_log_density`(*samples: Tensor*, *dimension: int*) → Tensor

Evaluates this GMM on the given samples with respect to the marginal log-density along the given dimensions

Parameters

- **samples** – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate
- **dimension** – int The dimension of interest

Returns
a one-dimensional tensor of shape num_samples containing the marginal log-densities

Return type
tf.Tensor

property num_components: int

Returns: int: the number of components of this GMM

remove_component(*idx: int*)

Removes the specified component, and renormalizes the weights afterwards.

Parameters
idx – int The idx of the component to be removed.

replace_components(*new_means: Tensor*, *new_chols: Tensor*)

Updates the means and covariances matrices (Cholesky) of the GMM. The weights and, therefore, the number of components can not be changed with this method.

Parameters

- **new_means** – tf.Tensor a two-dimensional tensor of shape current_number_of_components x dimensions, specifying the updated means.
- **new_chols** – tf.Tensor a three-dimensional tensor of shape current_number_of_components x dimensions x dimensions, specifying the updated Cholesky matrix.

replace_weights(*new_log_weights: Tensor*)

Overwrites the component log(weights). This method will take care of normalization.

Parameters
new_log_weights – tf.Tensor a one-dimensional tensor of size num_components, containing the new log(weights)

sample(*num_samples: int*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Draw samples from this GMM, also returns for every sample, the index of the component that was used for sampling it.

Parameters
num_samples – int The number of samples to be drawn

Returns

drawn_samples - a two-dimensional tensor of shape num_samples x num_dimensions, containing the drawn samples.

component_indices - a one-dimensional tensor of int, containing the component indices.

Return type
tuple(tf.Tensor, tf.Tensor)

sample_categorical(*num_samples: int*) → Tensor

Sample components according to the weights

Parameters
num_samples – int The number of components to be drawn

Returns
a one-dimensional tensor of int, containing the component indices.

Return type
tf.Tensor

sample_from_component(*index: int*, *num_samples: int*) → Tensor

draw samples from the specified components

Parameters

- **index** – int The index of the component from which we want to sample.
- **num_samples** – int The number of samples to be drawn.

Returns
The drawn samples, tensor of size num_samples x dimensions.

Return type
tf.Tensor

sample_from_components(*samples_per_component: Tensor*) → Tensor

Draws from each component the corresponding number of samples (provided as a one-dimensional tensor).

Parameters
samples_per_component – tf.Tensor a one-dimensional tensor of size number_of_component, containint for each component the number of samples to be drawn.

Returns
a tensor of shape sum(samples_per_component) x num_dimensions containing the samples (shuffled).

Return type
tf.Tensor

sample_from_components_no_shuffle(*samples_per_component: Tensor*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Draws from each component the corresponding number of samples (provided as a one-dimensional tensor). Similar to *sample_from_components*, but the returned samples are not shuffled.

Parameters
samples_per_component – tf.Tensor a one-dimensional tensor of size number_of_component, containint for each component the number of samples to be drawn.

Returns
a tensor of shape sum(samples_per_component) x num_dimensions containing the samples (not shuffled).

Return type
tf.Tensor

property weights: Tensor

Returns: tf.Tensor: a one-dimensional tensor of size num_components, containing the component weights.

class gmmvi.models.full_cov_gmm.**FullCovGMM**(*weights: Tensor*, *means: Tensor*, *covs: Tensor*)

A Gaussian mixture model with full covariance matrices.

Parameters

- **weights** – tf.Tensor a one-dimensional tensor containing the initial weights of the GMM.
- **means** – tf.Tensor a two-dimensional tensor containing the component means.
- **covs** – tf.Tensor a three-dimensional tensor containing the component covariance matrices.

add_component(*initial_weight: Tensor*, *initial_mean: Tensor*, *initial_cov: Tensor*)

Add a component to the mixture model. The weights will be automatically normalized.

Parameters

- **initial_weight** – tf.Tensor The weight of the new component (before re-normalization)
- **initial_mean** – tf.Tensor The mean of the new component
- **initial_cov** – tf.Tensor The covariance matrix of the new component

component_log_densities(*samples: Tensor*) → Tensor

Evaluate the log densities for each mixture component on the given samples.

Parameters
samples – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate.

Returns
a two-dimensional tensor of size number_of_components x number_of_samples, containing the log-densities for each component.

Return type
tf.Tensor

component_log_density(*index: int*, *samples: Tensor*) → Tensor

Use the specified component to evaluate the Gaussian log-density at the given samples.

Parameters

- **index** – int The index of the component of which we want to compute the log densities.
- **samples** – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate.

Returns
a one-dimensional tensor of size number_of_samples, containing the log-densities.

Return type
tf.Tensor

component_marginal_log_densities(*samples: Tensor*, *dim: int*) → Tensor

Evaluate the marginal log densities for each mixture component along the given dimension for each sample.

Parameters

- **samples** – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate. Note that for providing an easier interface, each sample has the number of dimensions compatible with this GMM, although only a single entry is actually used for evaluating the marginal density.
- **dimension** – int The dimension of interest.

Returns
a two-dimensional tensor of size number_of_components x number_of_samples, containing the marginal log-densities for each component.

Return type
tf.Tensor

property covs: Tensor

Returns: tf.Tensor: the covariance matrices as a 3-dimensional tensor of shape num_components x num_dimensions x num_dimensions

gaussian_entropy(*chol: Tensor*) → Tensor

Computes the entropy of Gaussian distribution with the given Cholesky matrix.

Parameters
chol – tf.Tensor A two-dimensional tensor of shape number_of_dimensions x number_of_dimensions specifying the Cholesky matrix

Returns
The entropy

Return type
tf.float32

sample_from_component(*index: int*, *num_samples: int*) → Tensor

draw samples from the specified components

Parameters

- **index** – int The index of the component from which we want to sample.
- **num_samples** – int The number of samples to be drawn.

Returns
The drawn samples, tensor of size num_samples x dimensions.

Return type
tf.Tensor

class gmmvi.models.diagonal_gmm.**DiagonalGMM**(*weights: Tensor*, *means: Tensor*, *covs: Tensor*)

A Gaussian mixture model with diagonal covariance matrices.

Parameters

- **weights** – tf.Tensor a one-dimensional tensor containing the initial weights of the GMM.
- **means** – tf.Tensor a two-dimensional tensor containing the component means.
- **covs** – tf.Tensor a two-dimensional tensor containing the diagonal entries of the component covariances.

add_component(*initial_weight: Tensor*, *initial_mean: Tensor*, *initial_cov: Tensor*)

Add a component to the mixture model. The weights will be automatically normalized.

Parameters

- **initial_weight** – tf.Tensor The weight of the new component (before re-normalization)
- **initial_mean** – tf.Tensor The mean of the new component
- **initial_cov** – tf.Tensor The covariance matrix of the new component

component_log_densities(*samples: Tensor*) → Tensor

Evaluate the log densities for each mixture component on the given samples.

Parameters
samples – tf.Tensor A two-dimensional tensor of shape number_of_samples x num_dimensions, which we want to evaluate.

Returns
a two-dimensional tensor of size number_of_components x number_of_samples, containing the log-densities for each component.

Return type
tf.Tensor

property covs: Tensor

Returns: tf.Tensor: the covariance matrices as a 3-dimensional tensor of shape num_components x num_dimensions x num_dimensions

static diagonal_gaussian_log_pdf(*dim: int*, *mean: Tensor*, *chol: Tensor*, *x: Tensor*) → Tensor

gaussian_entropy(*chol: Tensor*) → Tensor

Computes the entropy of Gaussian distribution with the given Cholesky matrix.

Parameters
chol – tf.Tensor A two-dimensional tensor of shape number_of_dimensions x number_of_dimensions specifying the Cholesky matrix

Returns
The entropy

Return type
tf.float32

sample_from_component(*index: int*, *num_samples: int*) → Tensor

draw samples from the specified components

Parameters

- **index** – int The index of the component from which we want to sample.
- **num_samples** – int The number of samples to be drawn.

Returns
The drawn samples, tensor of size num_samples x dimensions.

Return type
tf.Tensor

class gmmvi.models.gmm_wrapper.**GmmWrapper**(*model: GMM*, *initial_stepsize: float*, *initial_regularizer: float*, *max_reward_history_length: int*)

This method wraps around the *model* to keep track of component-specific meta-information used by the learner (e.g. component-specific stepsizes). This class can be used just like a *model*, because any methods not implemented within the GmmWrapper are forwarded to the encapuslated model. However, some functions have slightly different behavior, for example, when removing a component, not only the component in the encapuslated model will be removed, but also the meta-information, stored in this GmmWrapper. Hence, the model should always be accessed through the GmmWrapper.

Whenever adding a new component (via py:meth:*gmmvi.models.gmm.GmmWrapper.add_component*), the GmmWrapper will initialize the meta-information (stepsize and l2-regularizer) with the provided initial_values.

Parameters

- **model** – *gmmvi.models.gmm.GMM* The model to be encapuslated.
- **initial_stepsize** – float The stepsize, that is assigned to a newly added component
- **initial_regularizer** – float The l2 regularizer, that is assigned to a newly added component (only used when using a py:class:*MoreNgEstimator<gmmvi.optimization.gmmvi_modules.ng_estimator.MoreNgEstimator>* for estimating the natural gradients).
- **max_reward_history_length** – int The GmmWrapper also keeps track how much reward each component obtained at the previous iterations. This parameter controls after how many iterations the component rewards are forgotten (to save memory).

add_component(*initial_weight: tf.float32*, *initial_mean: Tensor*, *initial_cov: Tensor*, *adding_threshold: Tensor*, *initial_entropy: Tensor*)

Adds a new component to the encapuslated model (see *GMM*, but also stores / intializes meta-information.

Parameters

- **initial_weight** – tf.Tensor The weight of the new component (before re-normalization)
- **initial_mean** – tf.Tensor The mean of the new component
- **initial_cov** – tf.Tensor The covariance matrix of the new component
- **adding_threshold** – tf.Tensor The threshold used by *VipsComponentAdaptation*, stored for debugging.
- **initial_entropy** – tf.Tensor The initial entropy of the new component (can be computed from initial_cov).

static build_from_config(*model:* GMM, *config: dict*)

Create a *GmmWrapper* instance from a configuration dictionary.

This static method provides a convenient way to create a *GmmWrapper* instance, based on an initial *GMM*, and a dictionary containing the parameters.

Parameters

- **config** – dict The dictionary is typically read from YAML a file, and holds all hyperparameters. The max_reward_history_length, which is needed for instantiating the *GmmWrapper* is typically not provided directly, but chosen depending on whether *VipsComponentAdaptation* is used or not.
- **model** – *GMM* The model that we want to encapsulate.

remove_component(*idx: int*)

Deletes the given component in the encapuslated model (see *GMM*), but also deletes the corresponding meta-information.

Parameters

idx – int The idx of the component to be removed.

replace_weights(*new_log_weights: Tensor*)

Overwrites the weights of the encapuslated model, see (see *GMM*), but also keeps track of each component's weight from previous iterations.

Parameters

new_log_weights – tf.Tensor A one dimensional tensor of size number_of_components, containing the log of the new weight for each component.

store_rewards(*rewards: Tensor*)

Store the provided reward of each component.

Parameters

- **rewards** – tf.Tensor
- **number_of_components** (*A one dimensional tensor of size*) –
- **component** (*containing the reward for each*) –

update_stepsizes(*new_stepsizes: Tensor*)

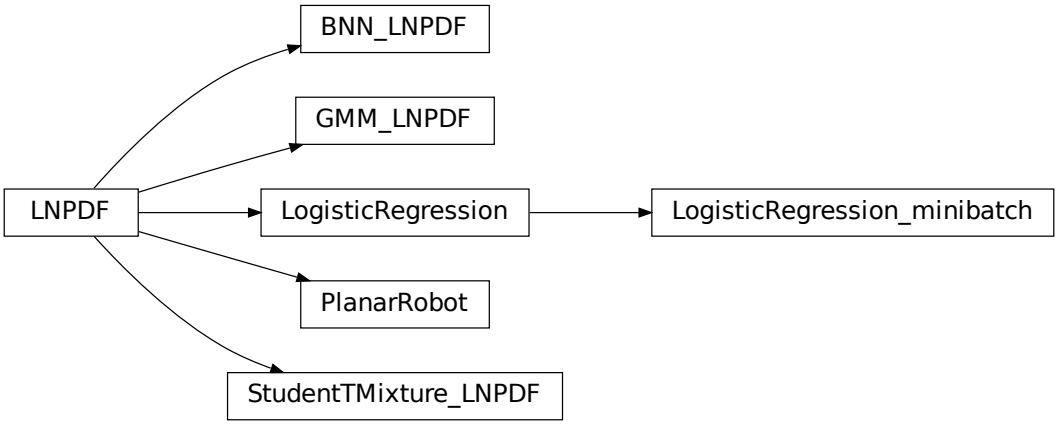
This method updates the stepsize for each component.

Parameters

- **new_stepsizes** – tf.Tensor
- **number_of_components** (*A one dimensional tensor of size*) –
- **component** (*containing the new stepsize for each*) –

3.2 Experiments

3.2.1 Target Distributions



class gmmvi.experiments.target_distributions.lnpdf.**LNPDF**(*use_log_density_and_grad: bool = False*, *safe_for_tf_graph: bool = True*)

This class defines the interface for target distributions. Every target distribution needs to implement the method log_density(x) for computing the unnormalized log-density of the target distribution.

This function can be also used to wrap target distributions that are not implemented in Tensorflow, by setting use_log_density_and_grad to True and safe_for_tf_graph to False.

Parameters

- **use_log_density_and_grad** – bool if False, user is allowed to backprob through self.log_density(), otherwise the method *log_density_and_grad* should be used (and needs to be implemented when using first-order estimates of the NG).
- **safe_for_tf_graph** – bool if True, we can call *log_density* and *log_density_and_grad* within a tf.function().

can_sample() → bool

If the target distribution can be sampled, and the respective *method* you can overwrite this method to return True.

Returns

is it is safe to call *sample*?

Return type
bool

expensive_metrics(*model:* GmmWrapper, *samples: Tensor*) → dict

(May not be implemented) This method can be used for computing environment-specific metrics or plots that we want to log. It is called by the *GmmviRunner*.

Parameters

- **model** – *GmmWrapper* The learned model that we want to evaluate for this target distribution.
- **samples** – tf.Tensor Samples that have been drawn from the model, which can be used for evaluations.

Returns
a dictionary containing the name and value for each item we wish to log.

Return type
dict

get_num_dimensions() → int

Returns
the number of dimensions

Return type
int

log_density(*x: Tensor*) → Tensor

Returns the unnormalized log-density for each sample in x, $\log p(\mathbf{x})$.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns
A one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

Return type
tf.Tensor

log_density_and_grad(*x: Tensor*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

(May not be implemented) Returns the unnormalized log-density and its gradient for each sample in x.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns

target_log_densities - a one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

log_density_grads - a two-dimensional tensor of shape number_of_samples x dimensions containing the gradients of the log-densities with respect to the respective sample.

Return type
tuple(tf.Tensor, tf.Tensor)

property safe_for_tf_graph: bool

if True, we can call *log_density* and *log_density_and_grad* within a tf.function().

sample(*n: int*) → Tensor

(May not be implemented) If we can sample from the target distribution, this functionality can be implemented here. However, it is not used by the learning algorithm.

Parameters
n – int The number of samples we want to draw

Returns
the sample, a Tensor of shape n x dimensions

Return type
tf.Tensor

property use_log_density_and_grad: bool

if False, user is allowed to backprob through self.log_density(), otherwise the method *log_density_and_grad* should be used (and needs to be implemented when using first-order estimates of the NG).

class gmmvi.experiments.target_distributions.gmm.**GMM_LNPDF**(*target_weights: Tensor*, *target_means: Tensor*, *target_covs: Tensor*)

Implements a target distribution that is given by a Gaussian mixture model.

Parameters

- **target_weights** (*tf.Tensor of tf.float32*) – a one-dimensional vector of size number_of_components containing the mixture weights.
- **target_means** (*tf.Tensor of tf.float32*) – a two-dimensional vector of size number_of_components x dimensions containing the mixture means.
- **target_covs** (*tf.Tensor of tf.float32*) – a three-dimensional vector of size number_of_components x dimensions x dimensions containing the covariance matrices.

can_sample()

Returns
We can sample from a GMM, so this method will return True.

Return type
bool

expensive_metrics(*model:* GmmWrapper, *samples: Tensor*) → dict

This method computed the number of detected modes (by testing how many modes of this target distribution are close to a component in the learned model) and a figure that shows plots comparing the marginal distributions of the model with the true marginals of this target distribution.

Parameters

- **model** – *GmmWrapper* The learned model that we want to evaluate for this target distribution.
- **samples** – tf.Tensor Samples that have been drawn from the model, which can be used for evaluations.

Returns
a dictionary containing two items (the number of detected modes, and a figure showing the plots of marginals.

Return type
dict

`get_num_dimensions`()

Returns
the number of dimensions

Return type
int

`log_density`(*x*)

Returns the unnormalized log-density for each sample in x, $\log p(\mathbf{x})$.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns
A one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

Return type
tf.Tensor

`marginal_log_density`(*x*, *dim*)

Computes the marginal distribution along the given dimensions.

Parameters

- **x** (*tf.Tensor of tf.float32*) – a one-dimensional vector of size number_of_samples containing the samples we want to evaluate
- **dim** (*an int*) – Specifies the dimension used for constructing the marginal GMM.
- **Returns** – tf.Tensor - a one-dimensional Tensor of shape number_of_samples containing the marginal log densities.

`sample`(*n*)

Draws n samples from this GMM.

Parameters
n – int The number of samples we want to draw.

Returns
The sample, a tensor of size n x dimensions.

Return type
tf.Tensor

class gmmvi.experiments.target_distributions.student_t_mixture.**StudentTMixture_LNPDF**(*target_weights: Tensor*, *target_means: Tensor*, *target_covs: Tensor*, *alpha=2*)

Implements a target distribution that is given by a mixture of Student-T distributions.

Parameters

- **target_weights** (*tf.Tensor of tf.float32*) – a one-dimensional vector of size number_of_components containing the mixture weights.
- **target_means** (*tf.Tensor of tf.float32*) – a two-dimensional vector of size number_of_components x dimensions containing the mixture means.
- **target_covs** (*tf.Tensor of tf.float32*) – a three-dimensional vector of size number_of_components x dimensions x dimensions containing the covariance matrices.
- **alpha** (*int*) – The number of degrees of freedom.

can_sample()

Returns
We can sample from a mixture of Student-T, so this method will return True.

Return type
bool

expensive_metrics(*model:* GmmWrapper, *samples: Tensor*) → dict

This method computed the number of detected modes (by testing how many modes of this mixture of Student-T are close to a component in the learned model) and a figure that shows plots comparing the marginal distributions of the model with the true marginals of this mixture of Student-T.

Parameters

- **model** – *GmmWrapper* The learned model that we want to evaluate for this target distribution.
- **samples** – tf.Tensor Samples that have been drawn from the model, which can be used for evaluations.

Returns
a dictionary containing two items (the number of detected modes, and a figure showing the plots of marginals.

Return type
dict

get_num_dimensions()

Returns
the number of dimensions

Return type
int

`log_density`(*x*)

Returns the unnormalized log-density for each sample in x, $\log p(\mathbf{x})$.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns
A one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

Return type
tf.Tensor

`marginal_log_density`(*x*, *dim*)

Computes the marginal distribution along the given dimensions.

Parameters

- **x** – tf.Tensor of tf.float32 a one-dimensional vector of size number_of_samples containing the samples we want to evaluate
- **dim** – an int Specifies the dimension used for constructing the marginal mixture of Student-Ts.

Returns
a one-dimensional tensor of shape number_of_samples containing the marginal log densities.

Return type
tf.Tensor

`sample`(*n*)

Draws n samples from this mixture of Student-T.

Parameters
n – int The number of samples we want to draw.

Returns
The sample, a tensor of size n x dimensions.

Return type
tf.Tensor

class gmmvi.experiments.target_distributions.logistic_regression.**LogisticRegression**(*dataset_id*)

This class is used for implementing the logistic regression experiments based on the BreastCancer and GermanCredit dataset [Lic13], reimplementing the experiments used by Arenz *et al.* [AZN20].

Parameters
dataset_id – a string Should be either "breast_cancer" or "german_credit"

`get_num_dimensions`()

Returns
the number of dimensions

Return type
int

log_density(*x*)

Returns the unnormalized log-density for each sample in x, $\log p(\mathbf{x})$.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns
A one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

Return type
tf.Tensor

log_likelihood(*x*)

property prior_std

class gmmvi.experiments.target_distributions.logistic_regression.**LogisticRegression_minibatch**(*dataset_id*, *batch-size*, *size_test_se* *use_own_ba*

This class is used for implementing minibatch-variants of the GermanCredit and BreastCancer *experiments*

Parameters

- **dataset_id** – str Should be either "breast_cancer" or "german_credit"
- **batchsize** – int batchsize for evaluating the likelihood.
- **size_test_set** – int number of training data that should be held out.
- **use_own_batch_per_samples** – bool if True, a different minibatch is used for every sample for which we want to evaluate the target log-density, which reduces the variance (local reparameterization).

expensive_metrics(*model:* GmmWrapper, *samples: Tensor*) → dict

As target-distribution specific metric, we estimate the full-batch ELBO.

Parameters

- **model** – *GmmWrapper* The learned model that we want to evaluate for this target distribution.
- **samples** – tf.Tensor Samples that have been drawn from the model and that are used for estimating the full-batch ELBO.

Returns
a dictionary with a single item containing the full-batch elbo.

Return type
dict

likelihood_batch(*x*, *data*, *labels*)

log_density(*x*)

Returns the unnormalized log-density for each sample in x, $\log p(\mathbf{x})$.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns
A one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

Return type
tf.Tensor

`log_density_fb`(*x*)

Evaluate the log-density on the full data set (used for evaluation). If size_test_set=0, this function is equivalent to *gmmvi.experiments.target_distributions.logistic_regression.LogisticRegression.log_density()*.

`shuffle_data`()

class gmmvi.experiments.target_distributions.planar_robot.**PlanarRobot**(*num_links*, *num_goals*, *prior_std=0.2*, *likelihood_std=0.01*)

This class reimplements the "PlanarRobot" experiments used by Arenz *et al.* [AZN20].

Parameters

- **num_links** – int The number of links of the robot
- **num_goals** – int The number of goal positions, must be either 1 or 4
- **prior_std** – float The standard deviation of the (zero-mean) prior on the joint angles. The first value is ignored, as the first link always has a standard deviation of 1.
- **likelihood_std** – float The std-deviation used for penalizing the distance in X-Y between the robot endeffective and the goal position.

`expensive_metrics`(*model:* GmmWrapper, *samples: Tensor*) → dict

This method computes two task-specific metrics:

1. The number of detected modes: This is course heuristic to count the different configurations used for reaching each of the goal positions (potentially misleading!)
2. Plots of the mean configurations of the learned model

Parameters

- **model** – *GmmWrapper* The learned model that we want to evaluate for this target distribution.
- **samples** – tf.Tensor Samples that have been drawn from the model, which can be used for evaluations.

Returns
a dictionary containing two items (the number of detected modes, and a figure showing the mean configurations).

Return type
dict

`forward_kinematics`(*theta*)

`get_num_dimensions`()

Returns
the number of dimensions

Return type
int

likelihood(*pos: Tensor*) → Tensor

log_density(*theta*)

Returns the unnormalized log-density for each sample in x, $\log p(\mathbf{x})$.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns
A one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

Return type
tf.Tensor

class gmmvi.experiments.target_distributions.bnn.**BNN_LNPDF**(*likelihood_scaling*, *dataset_seed*, *prior_std*, *batch_size*, *hidden_units*, *loss*, *activations*)

This class is used for implementing the target distribution given by the posterior for a Bayesian Neural Network.

Parameters

- **likelihood_scaling** – float a coefficient that can be used to scale the effect of the likelihood
- **dataset_seed** – int The dataset_seed is used for reproducible train/test-splits
- **prior_std** – float The standard deviation of the (zero-mean) prior over the network weights
- **batch_size** – int size of the minibatches
- **hidden_units** – list[int] The length of the list defines the number of hidden layers, the entries define their width
- **loss** – a tf.Keras.losses The loss function used for computing the log-likelihood
- **activations** – a list of Tensorflow activation functions activations for each hidden layer and the output layer

average_loss(*x*, *dataset*)

avg_bayesian_inference_loss(*x*, *dataset*)

avg_bayesian_inference_test_loss(*x*, *num_batches*)

bayesian_inference_test_loss(*x*)

create_model()

forward_from_weight_vector(*input*, *x*)

get_num_dimensions()

Returns
the number of dimensions

Return type
int

log_density(*x*)

Returns the unnormalized log-density for each sample in x, $\log p(\mathbf{x})$.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns
A one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

Return type
tf.Tensor

log_density_and_grad(*x: Tensor*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

(May not be implemented) Returns the unnormalized log-density and its gradient for each sample in x.

Parameters
x – tf.Tensor The samples that we want to evaluate, a tf.Tensor of shape number_of_samples x dimensions.

Returns

target_log_densities - a one-dimensional tensor of shape number_of_samples containing the unnormalized log-densities.

log_density_grads - a two-dimensional tensor of shape number_of_samples x dimensions containing the gradients of the log-densities with respect to the respective sample.

Return type
tuple(tf.Tensor, tf.Tensor)

log_likelihood(*x*)

log_likelihood_and_grad(*x*)

log_likelihood_old(*x*)

log_prior(*x*, *ignore_constant=False*)

log_prior_and_grad(*x*, *ignore_constant=False*)

prepare_data()

property prior_std

3.2.2 Evaluation

class gmmvi.experiments.evaluation.mmd.**MMD**(*groundtruth*, *alpha*)

This class can be used for computing the Maximum Mean Discrepancy [GBR+12]. The MMD can be used to compute the discrepancy between a model sample and a groundtruth sample.

Note that instantiating this object can be quite slow, but computing the MMD using *compute_MMD* should be fast.

Parameters

- **groundtruth** – tf.Tensor The groundtruth sample of shape number_of_samples x dimension

- **alpha** – tf.float32 A factor for scaling the diagonal bandwidth matrix (which is automatically chosen based on the groundtruth sample using the Median trick [GBR+12]).

compute_MMD(*model_sample*)

Compute the MMD between the model_sample and the groundtruth data that was provided when instantiating this object.

Parameters
model_sample – tf.Tensor The sample from the model of shape number_of_samples x dimension

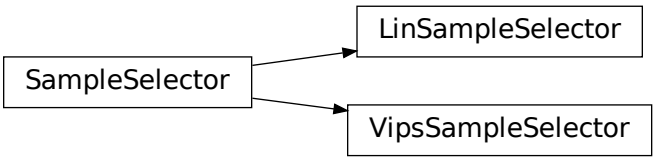
Returns
The MMD between model sample and groundtruth sample

Return type
float

3.3 Optimization

3.3.1 GMMVI Modules

SampleSelector



class gmmvi.optimization.gmmvi_modules.sample_selector.**SampleSelector**(*target_distribution:* LNPDF, *model:* GmmWrapper, *sample_db:* SampleDB)

Provides the interface for selecting samples for performing the updates at the beginning of every iteration.

The samples are evaluated on the target distribution and used for updating the weights, means and covariance of the GMM.

There are currently two options for estimating the natural gradient:

1. The *VipsSampleSelector* use the procedure described by Arenz *et al.* [AZN18], Arenz *et al.* [AZN20] to ensure that we have samples in the vicinity of every component, enabling us to perform a stable update on every component.
2. **The *LinSampleSelector* uses the procedure described by Lin *et al.* [LKS19a] which draws samples** according to the weights of the current mixture model, aiming for better sample efficiency.

Parameters

- **target_distribution** – LNPDF The target distribution is used for evaluating the newly drawn samples.
- **model** – *GmmWrapper* The wrapped model is used for drawing the samples.
- **sample_db** – SampleDB The new samples and their target_densities (and gradients) are stored in the sample database.

static build_from_config(*config*, *gmm_wrapper*, *sample_db*, *target_distribution*)

This static method provides a convenient way to create a *VipsSampleSelector*, or *LinSampleSelector* depending on the provided config.

Parameters

- **config** – dict The dictionary is typically read from YAML a file, and holds all hyperparameters.
- **gmm_wrapper** – *GmmWrapper* The wrapped model is used for drawing the samples.
- **sample_db** – SampleDB The new samples and their target_densities (and gradients) are stored in the sample database.
- **target_distribution** – LNPDF The target distribution is used for evaluating the newly drawn samples.

select_samples() → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Select the samples for current learning iteration and stores the data in the sample database.

Returns

samples - a tensor of shape number_of_selected_samples x number_of_dimensions

old_samples_pdf - a tensor of shape number_of_selected_samples, containing the log-densities of the distribution that was effectively used to obtain the selected samples. Needed for importance weighting.

target_lnpdfs - a tensor of shape number_of_selected_samples, containing the log-densities of the target distrbution for each selected sample, $\log p(\mathbf{x})$.

target_grads - a tensor of shape number_of_selected_samples x num_dimensions, containing the gradients of the log-densities of the target distrbution for each selected sample, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.

Return type
tuple(tf.Tensor, tf.Tensor, tf.Tensor, tf.Tensor)

LinSampleSelector

class gmmvi.optimization.gmmvi_modules.sample_selector.**LinSampleSelector**(*target_distribution: LNPDF*, *model: GmmWrapper*, *sample_db: SampleDB*, *desired_samples_per_component: int*, *ratio_reused_samples_to_desired: float*)

Selects the samples according to the procedure described by Lin *et al.* [LKS19a].

This class uses the procedure described by Lin *et al.* [LKS19a] by drawing new samples for the current mixture model. We also implemented the two-phase procedure of the *VipsSampleSelector* to reuse samples from the database and redraw samples based on a desired number of samples. However, in contrast to the *VipsSampleSelector*, we compute the effective sample size not per component, but for the whole mixture, and redraw samples n_eff - *desired_samples_per_component* new samples from the mixture model. The exact procedure of Lin *et al.* [LKS19a] can be reproduced, when choosing *ratio_reused_samples_to_desired* = 0, where always a fixed number of new samples is drawn from the mixture model.

Parameters

- **target_distribution** – LNPDF The target distribution is used for evaluating the newly drawn samples.
- **model** – *GmmWrapper* The wrapped model is used for drawing the samples.
- **sample_db** – SampleDB The database is used for reusing samples from previous iterations and for storing the new samples and their target_densities (and gradients).
- **desired_samples_per_component** – int The desired number for the mixture update.
- **ratio_reused_samples_to_desired** – float In the first pass, we reuse the *ratio_reused_samples_to_desired* * *desired_samples_per_component* freshest samples from the database.

get_effective_samples(*model_densities: Tensor*, *oldsamples_pdf: Tensor*) → Tensor

Computes the effective sample size of the mixture model based on the log-densities of the target distribution and the log-densities of the background distribution.

Parameters

- **model_densities** – tf.Tensor The log-densities of the mixture model, $\log q(\mathbf{x})$.
- **oldsamples_pdf** – tf.Tensor The log-densities of the distribution that was effectively used for obtaining the selected samples

Returns
the effective number of samples

Return type
float

sample_where_needed() → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'int'>]

Computes the mixture model's effective sample size for the given set of samples and draws $n_{\text{des}} - n_{\text{eff}}$ new samples from the mixture model.

Parameters

- **samples** – tf.Tensor the samples that were chosen during the first pass
- **oldsamples_pdf** – tf.Tensor The log-densities of the distribution that was effectively used for obtaining the selected samples
- **num_desired_samples** – int The number of desired samples per component

Returns

new_samples - a tensor containing the newly drawn samples

new_target_lnpdfs - a tensor containing the log-densities of the target distribution on the newly drawn samples, $\log p(\mathbf{x})$.

new_target_grads - a tensor containing the gradients of the log-densities for the newly drawn samples.

mapping - a tensor containing for every sample the one-dimensional tensor contains the index of the component that was used for drawing that sample.

Return type
tuple(tf.Tensor, tf.Tensor, tf.Tensor, tf.Tensor)

`select_samples()` → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Select the samples for current learning iteration and stores the data in the sample database.

Returns

samples - a tensor of shape number_of_selected_samples x number_of_dimensions

old_samples_pdf - a tensor of shape number_of_selected_samples, containing the log-densities of the distribution that was effectively used to obtain the selected samples. Needed for importance weighting.

target_lnpdfs - a tensor of shape number_of_selected_samples, containing the log-densities of the target distrbution for each selected sample, $\log p(\mathbf{x})$.

target_grads - a tensor of shape number_of_selected_samples x num_dimensions, containing the gradients of the log-densities of the target distrbution for each selected sample, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.

Return type
tuple(tf.Tensor, tf.Tensor, tf.Tensor, tf.Tensor)

VipsSampleSelector

class gmmvi.optimization.gmmvi_modules.sample_selector.**VipsSampleSelector**(*target_distribution: LNPDF*, *model: GmmWrapper*, *sample_db: SampleDB*, *desired_samples_per_component: int*, *ratio_reused_samples_to_desired: float*)

Selects the samples according to the procedure described by Arenz *et al.* [AZN18], Arenz *et al.* [AZN20].

This class uses the procedure described by Arenz *et al.* [AZN18], Arenz *et al.* [AZN20] to ensure that we have samples in the vicinity of every component. It uses two passes. In the first pass, it selects a given number of samples from the sample database. In the second pass, it computes the effective sample size for every component (based on the importance weights) and compares the effective sample size with a given desired number of samples. It then draws from every component the respective missing number of samples.

Parameters

- **target_distribution** – LNPDF The target distribution is used for evaluating the newly drawn samples.
- **model** – *GmmWrapper* The wrapped model is used for drawing the samples.

- **sample_db** – SampleDB The database is used for reusing samples from previous iterations and for storing the new samples and their target_densities (and gradients).
- **desired_samples_per_component** – int The desired number of samples for every component.
- **ratio_reused_samples_to_desired** – float In the first pass, we reuse the number_of_components * *ratio_reused_samples_to_desired* * *desired_samples_per_component* freshest samples from the database.

get_effective_samples(*model_densities: Tensor*, *oldsamples_pdf: Tensor*) → Tensor

Computes the effective sample size based on the log-densities of the target distribution and the log-densities of the background distribution.

Parameters

- **model_densities** – tf.Tensor The log-densities of the individual components, $\log q(\mathbf{x}|o)$
- **oldsamples_pdf** – tf.Tensor The log-densities of the distribution that was effectively used for obtaining the selected samples

Returns
the effective number of samples

Return type
float

sample_where_needed(*samples: Tensor*, *oldsamples_pdf: Tensor*, *num_desired_samples: Optional[int] = None*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Computes the components' effective sample sizes for the given set of samples and draws, for every component i, $n_{\text{des}} - n_{\text{eff,i}}$ new samples.

Parameters

- **samples** – tf.Tensor the samples that were chosen during the first pass
- **oldsamples_pdf** – tf.Tensor The log-densities of the distribution that was effectively used for obtaining the selected samples
- **num_desired_samples** – int The number of desired samples per component

Returns

new_samples - a tf.Tensor, the newly drawn samples

new_target_lnpdfs - a tf.Tensor, the log-densities of the target distribution on the newly drawn samples, $\log p(\mathbf{x})$.

new_target_grads - a tf.Tensor, the gradients of the log-densities for the newly drawn samples, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.

mapping - a tf.Tensor, for every sample the one-dimensional tensor contains the index of the component that was used for drawing that sample.

Return type
tuple(tf.Tensor, tf.Tensor, tf.Tensor, tf.Tensor)

select_samples() → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Select the samples for current learning iteration and stores the data in the sample database.

Returns

samples - a tensor of shape number_of_selected_samples x number_of_dimensions

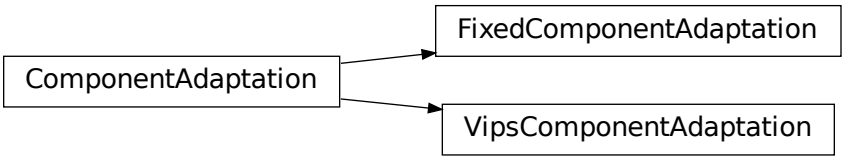
old_samples_pdf - a tensor of shape number_of_selected_samples, containing the log-densities of the distribution that was effectively used to obtain the selected samples. Needed for importance weighting.

target_lnpdfs - a tensor of shape number_of_selected_samples, containing the log-densities of the target distrbution for each selected sample, $\log p(\mathbf{x})$.

target_grads - a tensor of shape number_of_selected_samples x num_dimensions, containing the gradients of the log-densities of the target distrbution for each selected sample, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.

Return type
tuple(tf.Tensor, tf.Tensor, tf.Tensor, tf.Tensor)

ComponentAdaptation



class gmmvi.optimization.gmmvi_modules.component_adaptation.**ComponentAdaptation**

This class provides a common interface for adapting the number of components.

There are currently only two options:

1. The *FixedComponentAdaptation* a dummy-class, that does not do anything.
2. The *VipsComponentAdaptation* uses the procedure of VIPS [AZN20] to add and delete components.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to adapt the number of components.
- **sample_db** – *SampleDB* The sample database can be used to select candidate locations for adding a new component, without having to perform additional queries to the target distribution.
- **target_distribution** – *LNPDF* The target distribution can be used to evaluate candidate locations for adding a new component.
- **prior_mean** – tf.Tensor A one dimensional tensor of size num_dimensions, specifying the mean of the Gaussian that we can use to sample candidate locations for adding a new component.

- **initial_cov** – tf.Tensor A two-dimensional tensor of size num_dimensions x num_dimensions, specifying the covariance of the Gaussian that we can use to sample candidate locations for adding a new component.

static build_from_config(*config*, *gmm_wrapper*, *sample_db*, *target_distribution*, *prior_mean*, *initial_cov*)

This static method provides a convenient way to create a *FixedComponentAdaptation* or *VipsComponentAdaptation* depending on the provided config.

Parameters

- **config** – dict The dictionary is typically read from YAML a file, and holds all hyperparameters.
- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to adapt the number of components.
- **sample_db** – *SampleDB* The sample database can be used to select candidate locations for adding a new component, without having to perform additional queries to the target distribution.
- **target_distribution** – *LNPDF* The target distribution can be used to evaluate candidate locations for adding a new component.
- **prior_mean** – tf.Tensor A one dimensional tensor of size num_dimensions, specifying the mean of the Gaussian that we can use to sample candidate locations for adding a new component.
- **initial_cov** – tf.Tensor A two-dimensional tensor of size num_dimensions x num_dimensions, specifying the covariance of the Gaussian that we can use to sample candidate locations for adding a new component.

FixedComponentAdaptation

class gmmvi.optimization.gmmvi_modules.component_adaptation.**FixedComponentAdaptation**

This is a dummy class, used when we do not want to adapt the number of components during learning.

adapt_number_of_components(*iteration*)

As we do not want to change the number of components, this method does not do anything.

Parameters

iteration – int The current iteration (ignored).

VipsComponentAdaptation

class gmmvi.optimization.gmmvi_modules.component_adaptation.**VipsComponentAdaptation**(*model: GmmWrapper, sample_db: SampleDB, target_lnpdf: LNPDF, prior_mean: Union[float, Tensor], initial_cov: Union[float, Tensor], del_iters: int, add_iters: int, max_components: int, thresholds_for_add_heuristic: float, min_weight_for_del_heu float, num_database_samples: int, num_prior_samples: int*)

This class implements the component adaptation procedure used by VIPS.

See [AZN20].

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to adapt the number of components.
- **sample_db** – *SampleDB* The sample database can be used to select candidate locations for adding a new component, without having to perform additional queries to the target distribution.
- **target_distribution** – *LNPDF* The target distribution can be used to evaluate candidate locations for adding a new component.
- **prior_mean** – tf.Tensor A one dimensional tensor of size num_dimensions, specifying the mean of the Gaussian that we can use to sample candidate locations for adding a new component.
- **initial_cov** – tf.Tensor A two-dimensional tensor of size num_dimensions x num_dimensions, specifying the covariance of the Gaussian that we can use to sample candidate locations for adding a new component.

- **del_iters** – int minimum number of updates a component needs to have received, before it is considered as candidate for deletion.
- **add_iters** – int a new component will be added every *add_iters* iterations
- **max_components** – int do not add components, if the model has at least *max_components* components
- **num_database_samples** – int number of samples from the *SampleDB* that are used for selecting a good initial mean when adding a new component.
- **num_prior_samples** – int number of samples from the prior distribution that are used for selecting a good initial mean when adding a new component.

adapt_number_of_components(*iteration: int*)

This method may change the number of components, either by deleting bad components that have low weights, or by adding new components.

Parameters
iteration – int The current iteration, used to decide whether a new component should be added.

add_at_best_location(*samples*, *target_lnpdfs*)

Find the most promising [AZN20] location among the provided samples for adding a new component, that is, a new component will be added with mean given by one of the provided samples.

Parameters

- **samples** – tf.Tensor candidate locations for initializing the mean of the new component
- **target_lnpdfs** – tf.Tensor for each candidate location, this tensor contains the log-density under the (unnormalized) target distribution.

add_new_component()

This method adds a new component by first selecting a set of candidate locations and the choosing the most promising one using the procedure of VIPS [AZN20].

delete_bad_components()

Components are deleted, if all the following criteria are met received:

1. It must have received at least *del_iters* updates
2. It must not have improved significantly during the last iterations. In contrast to VIPS, we use a Gaussian filter to smooth the rewards of the component, to be more robust with respect to noisy target distributions.
3. It must have very low weight, such that the effects on the model are negligible.

select_samples_for_adding_heuristic()

Select a set of samples used as candidates for initializing the mean of the new component.

Returns

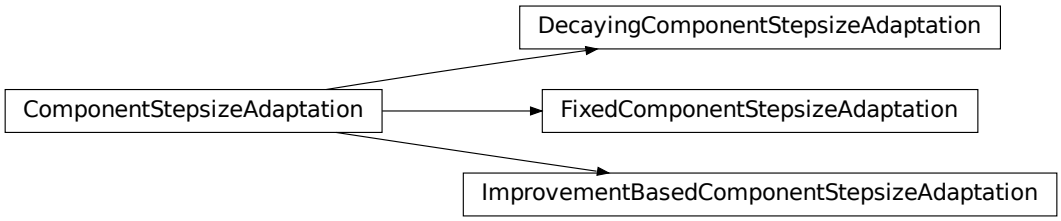
samples - the selected candidate locations

target_lnpdfs - log-densities of the *samples* under the unnormalized target distribution

prior_samples - additional samples drawn from a prior, which have not yet been evaluated on the target distribution.

Return type
tuple(tf.Tensor, tf.Tensor, tf.Tensor)

ComponentStepsizeAdaptation



class gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.**ComponentStepsizeAdaptation**(*gmm_wr* GmmWrapper, *initial_step* *float*)

This class provides a common interface for adapting the individual stepsizes for the component updates.

There are currently three options for component stepsize adpatation:

1. The *FixedComponentStepsizeAdaptation* is a dummy-class, that does not do anything.
2. The *DecayingComponentStepsizeAdaptation* uses exponential decay.
3. The *ImprovementBasedComponentStepsizeAdaptation* uses the procedure of VIPS [AZN20] to increase the stepsize if a component improved during the last updates, and to decrease it otherwise.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to adapt the number of components.
- **initial_stepsize** – float The stepsize used when the component receives its first update

static build_from_config(*config*, *gmm_wrapper*)

This static method provides a convenient way to create a *FixedComponentStepsizeAdaptation*, *DecayingComponentStepsizeAdaptation* or *ImprovementBasedComponentStepsizeAdaptation* depending on the provided config.

Parameters

- **config** – dict The dictionary is typically read from YAML a file, and holds all hyperparameters.
- **gmm_wrapper** – *GmmWrapper* The wrapped model.

update_stepsize(*current_stepsizes: Tensor*) → Tensor

Update the stepsizes, according to the chosen procedure.

Parameters

current_stepsizes – tf.Tensor A tensor that contains the stepsize of each component

Returns
a tensor of same size as *current_stepsizes* that contains the updates stepsizes.

Return type
tf.Tensor

DecayingComponentStepsizeAdaptation

class gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.**DecayingComponentStepsizeAdaptatio**

This class implements an exponentially decaying stepsize schedule. See [KNT+18].

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model.
- **annealing_exponent** – float controls how fast the stepsize decays
- **initial_stepsize** – float The stepsize used for all component updates

update_stepsize(*current_stepsizes: Tensor*) → Tensor

Updates the stepsize using exponential decay. More specifically, the new stepsize is given by $\frac{\text{initial\_stepsize}}{1+\text{num\_received\_updates}^{\text{annealing\_exponent}}}$.

Parameters
current_stepsizes – tf.Tensor A tensor that contains the stepsize of each component

Returns
a tensor of same size as *current_stepsizes* that contains the updates stepsizes.

Return type
tf.Tensor

FixedComponentStepsizeAdaptation

class gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.**FixedComponentStepsizeAdaptation**(*g*
G
p
in
ti
fl

This class is a dummy class, that can be used when we want to keep the stepsizes constant.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to adapt the number of components.
- **initial_stepsize** – float The stepsize used for all component updates

update_stepsize(*current_stepsizes: Tensor*) → Tensor

This dummy function does nothing.

Parameters
current_stepsizes – tf.Tensor A tensor that contains the stepsize of each component

Returns
the same *current_stepsizes* tensor, that it was called with.

Return type
tf.Tensor

ImprovementBasedComponentStepsizeAdaptation

class gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.**ImprovementBasedComponentStepsizeA**

Increases the stepsize if the last component update increased its reward, decreases it otherwise. See [AZN20].

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to adapt the number of components.
- **initial_stepsize** – float The stepsize used for all component updates
- **min_stepsize** – float Do not not decrease the stepsize below this point
- **max_stepsize** – float Do not increase the stepsize above this point
- **stepsize_inc_factor** – float Factor (>1) for increasing the stepsize
- **stepsize_dec_factor** – float Factor in the range [0, 1] for decreasing the stepsize

update_stepsize(*current_stepsizes: Tensor*) → Tensor

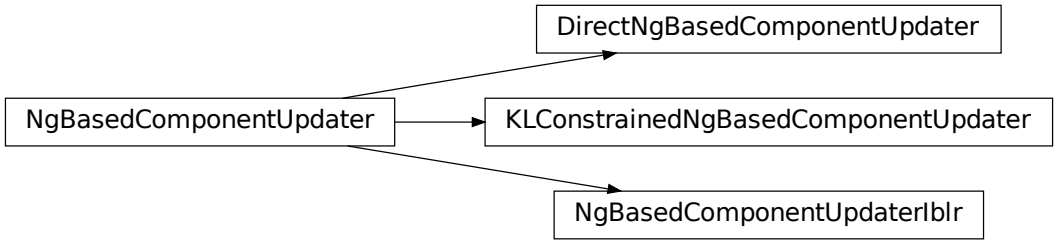
Updates the stepsize of each component based on previous reward improvements [AZN20].

Parameters
current_stepsizes – tf.Tensor A tensor that contains the stepsize of each component

Returns
a tensor of same size as *current_stepsizes* that contains the updates stepsizes.

Return type
tf.Tensor

NgBasedComponentUpdater



class gmmvi.optimization.gmmvi_modules.ng_based_component_updater.**NgBasedComponentUpdater**(*model: GmmWrapper*, *temperature: float*)

This class provides a common interface for updating the Gaussian components along the natural gradient.

The Gaussian components of the mixture model, are updated by updating their parameters (their mean and covariance) based on previously computed estimates of the natural gradient (see `NgEstimator`) and stepsizes (see `ComponentStepsizeAdaptation`).

There are currently three options for updating the components:

1. The *DirectNgBasedComponentUpdater* straightforwardly performs the natural gradient with the given stepsize.
2. The *NgBasedComponentUpdaterIblr* uses the improved Bayesian learning rate to ensure positive definite covariance matrices.
3. The *KLConstrainedNgBasedComponentUpdater* treats the stepsize as a trust-region constraint.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to update the components.
- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.

apply_NG_update(*expected_hessians_neg: Tensor*, *expected_gradients_neg: Tensor*, *stepsizes: Tensor*)

Update the components based on the estimates of the natural gradients (provided in terms of the negated expected Hessian and expected gradients) and the given component-specific stepsizes.

Parameters

- **expected_hessians_neg** – tf.Tensor A three-dimensional tensor of shape num_components x num_dimensions x num_dimensions, containing an estimate of $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{xx}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$ for each component.

- **expected_gradients_neg** – tf.Tensor A two-dimensional tensor of shape num_components x num_dimensions x num_dimensions, containing an estimate of $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{x}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$ for each component.
- **stepsizes** – tf.Tensor A one-dimensional tensor of shape num_components, containing the stepsize for each component.

static build_from_config(*config*, *gmm_wrapper*)

This static method provides a convenient way to create a *DirectNgBasedComponentUpdater*, *NgBasedComponentUpdaterIblr* or *KLConstrainedNgBasedComponentUpdater* depending on the provided config.

Parameters

- **config** – dict The dictionary is typically read from YAML a file, and holds all hyperparameters.
- **gmm_wrapper** – *GmmWrapper* The wrapped model for which we want to update the components

DirectNgBasedComponentUpdater

class gmmvi.optimization.gmmvi_modules.ng_based_component_updater.**DirectNgBasedComponentUpdater**(*model: GmmWrapper*, *temperature: float*)

This class straightforwardly performs the natural gradient update with the given stepsize.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to update the components.
- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.

apply_NG_update(*expected_hessians_neg: Tensor*, *expected_gradients_neg: Tensor*, *stepsizes: Tensor*)

Update the components based on the estimates of the natural gradients (provided in terms of the negated expected Hessian and expected gradients) and the given component-specific stepsizes.

Parameters

- **expected_hessians_neg** – tf.Tensor A three-dimensional tensor of shape num_components x num_dimensions x num_dimensions, containing an estimate of $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{xx}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$ for each component.
- **expected_gradients_neg** – tf.Tensor A two-dimensional tensor of shape num_components x num_dimensions x num_dimensions, containing an estimate of $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{x}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$ for each component.
- **stepsizes** – tf.Tensor A one-dimensional tensor of shape num_components, containing the stepsize for each component.

KLConstrainedNgBasedComponentUpdater

class gmmvi.optimization.gmmvi_modules.ng_based_component_updater.**KLConstrainedNgBasedComponentUpdater**(

Updates the component by treating the stepsize as a constraint on the KL-divergence to the current component.

This class updates the component by performing the largest update along the natural gradient direction, that confines with a trust-region constraint on the Kullback-Leibler divergence with respect to the current component [AZN20].

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to update the components.
- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.

apply_NG_update(*expected_hessians_neg: Tensor*, *expected_gradients_neg: Tensor*, *stepsizes: Tensor*)

Update the components based on the estimates of the natural gradients (provided in terms of the negated expected Hessian and expected gradients) and the given component-specific stepsizes.

Parameters

- **expected_hessians_neg** – tf.Tensor A three-dimensional tensor of shape num_components x num_dimensions x num_dimensions, containing an estimate of $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{xx}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$ for each component.
- **expected_gradients_neg** – tf.Tensor A two-dimensional tensor of shape num_components x num_dimensions x num_dimensions, containing an estimate of $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{x}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$ for each component.
- **stepsizes** – tf.Tensor A one-dimensional tensor of shape num_components, containing the stepsize for each component.

bracketing_search(*kl_bound: tf.float32*, *lower_bound: tf.float32*, *upper_bound: tf.float32*, *old_lin_term: Tensor*, *old_precision: Tensor*, *old_inv_chol: Tensor*, *reward_lin_term: Tensor*, *reward_quad_term: Tensor*, *kl_const_part: tf.float32*, *old_mean: Tensor*, *eta_in_logspace: tf.float32*) → [tf.float32, tf.float32]

This method finds the largest stepsize eta, such that the updated component stays within a KL-constrained trust-region around the current component. Whereas, [AZN20] used L-BFGS-B to solve this convex optimization problem, here we use a simple bracketing search, which seems to be more robust (by not relying on gradients, and can be efficiently performed within a Tensorflow graph. The procedure simple keeps track of a lower bound and an upper bound on the optimal stepsize and recursively evaluates the arithmetic mean of both bounds. If this mean-stepsize results in a two large KL divergence, or in a non-positive-definite covariance matrix, it becomes the new lower bound; otherwise the new upper bound.

Parameters

- **kl_bound** – tf.float32 The trust region constraint
- **lower_bound** – tf.float32 The initial lower bound on the stepsize
- **upper_bound** – The initial upper bound on the stepsize

- **old_lin_term** – tf.Tensor The linear term of the canonical Gaussian form of the current component. A one-dimensional tensor of shape num_dimensions.
- **old_precision** – The precision matrix of the current component. A two-dimensional tensor of shape num_dimensions x num_dimensions.
- **old_inv_chol** – The inverse of the Cholesky matrix of the current component. A two-dimensional tensor of shape num_dimensions x num_dimensions.
- **reward_lin** – When using MORE to estimate the natural gradient, this tensor correspond to the linear coefficient of the quadratic reward model. When using Stein's Lemma, this term can be computed from the expected gradient and expected Hessian.
- **reward_quad** – When using MORE to estimate the natural gradient, this tensor correspond to the quadratic coefficient of the quadratic reward model. When using Stein's Lemma, this term can be computed from the expected Hessian.
- **kl_const_part** – A term of the KL divergence that can be precomputed as it does not depend on the parameters of the updated component.
- **old_mean** – The mean of the current component.
- **eta_in_logspace** – if true, the bracketing search should be performed in log-space (requires fewer iterations)

Returns

new_lower_bound - The lower bound after a stopping criterion was reached.

new_upper_bound - The upper bound after a stopping criterion was reached.

Return type
tuple(tf.float32, tf.float32)

kl(*eta: tf.float32*, *old_lin_term: Tensor*, *old_precision: Tensor*, *old_inv_chol: Tensor*, *reward_lin: Tensor*, *reward_quad: Tensor*, *kl_const_part: tf.float32*, *old_mean: Tensor*, *eta_in_logspace: bool*) → [tf.float32, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Computes the Kullback-Leibler divergence between the updated component and current component, when updating with stepsize eta along the natural gradient.

Parameters

- **eta** – tf.float32 The stepsize for which the KL divergence should be computed.
- **old_lin_term** – tf.Tensor The linear term of the canonical Gaussian form of the current component. A one-dimensional tensor of shape num_dimensions.
- **old_precision** – tf.Tensor The precision matrix of the current component. A two-dimensional tensor of shape num_dimensions x num_dimensions.
- **old_inv_chol** – tf.Tensor The inverse of the Cholesky matrix of the current component. A two-dimensional tensor of shape num_dimensions x num_dimensions.
- **reward_lin** – tf.Tensor When using MORE to estimate the natural gradient, this tensor correspond to the linear coefficient of the quadratic reward model. When using Stein's Lemma, this term can be computed from the expected gradient and expected Hessian.
- **reward_quad** – tf.Tensor When using MORE to estimate the natural gradient, this tensor correspond to the quadratic coefficient of the quadratic reward model. When using Stein's Lemma, this term can be computed from the expected Hessian.
- **kl_const_part** – tf.float32 A term of the KL divergence that can be precomputed as it does not depend on the parameters of the updated component.

- **old_mean** – tf.Tensor The mean of the current component.
- **eta_in_logspace** – bool if true, the provided eta is given in log-space.

Returns

kl - a float corresponding to the KL between the updated component and the old component.

new_mean - a tensor specifying the mean of the updated component.

new_precision - a tensor specifying the precision of the updated component.

inv_chol_inv - a tensor specifying the inverse of the Cholesky of the precision matrix of the updated component.

Return type
tuple(float, tf.Tensor, tf.Tensor, tf.Tensor)

NgBasedComponentUpdaterIblr

class gmmvi.optimization.gmmvi_modules.ng_based_component_updater.**NgBasedComponentUpdaterIblr**(*model: GmmWrapper*, *temperature: float*)

This class updates the component using the improved Bayesian learning rule.

The iBLR performs Riemannian gradient descent and ensures positive definite covariance matrices [LSK20].

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to update the components.
- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.

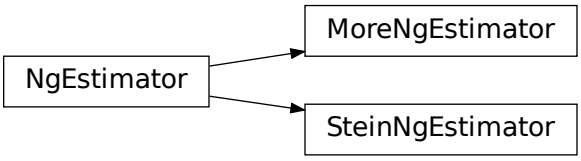
apply_NG_update(*expected_hessians_neg: Tensor*, *expected_gradients_neg: Tensor*, *stepsizes: Tensor*)

Update the components based on the estimates of the natural gradients (provided in terms of the negated expected Hessian and expected gradients) and the given component-specific stepsizes.

Parameters

- **expected_hessians_neg** – tf.Tensor A three-dimensional tensor of shape num_components x num_dimensions x num_dimensions, containing an estimate of $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{xx}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$ for each component.
- **expected_gradients_neg** – tf.Tensor A two-dimensional tensor of shape num_components x num_dimensions x num_dimensions, containing an estimate of $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{x}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$ for each component.
- **stepsizes** – tf.Tensor A one-dimensional tensor of shape num_components, containing the stepsize for each component.

NgEstimator



class gmmvi.optimization.gmmvi_modules.ng_estimator.**NgEstimator**(*temperature*, *model:* GmmWrapper, *requires_gradient:* *bool*, *only_use_own_samples:* *bool*, *use_self_normalized_importance_weights:* *bool*)

This class provides a common interface for estimating the natural gradient for a Gaussian component.

There are currently two options for estimating the natural gradient:

1. The *MoreNgEstimator* uses compatible function approximation to estimate the natural gradient from a quadratic reward surrogate [ALL+15, PTA+19, PS08, SMSM99].
2. The *SteinNgEstimator* uses Stein's Lemma to estimate the natural gradient using first-order information [LKS19b].

Parameters

- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.
- **model** – *GmmWrapper* The wrapped model where we want to update the components.
- **requires_gradient** – bool Does this object require first-order information?
- **only_use_own_samples** – bool If true, we do not use importance sampling to update one component based on samples from a different component.
- **use_self_normalized_importance_weights** – bool if true, use self-normalized importance weighting (normalizing the importance weights such they sum to one), rather than standard importance weighting.

static **build_from_config**(*config*, *temperature*, *gmm_wrapper*)

This static method provides a convenient way to create a *MoreNgEstimator*, or *SteinNgEstimator* depending on the provided config.

Parameters

- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.
- **config** – dict The dictionary is typically read from YAML a file, and holds all hyperparameters.

get_expected_hessian_and_grad(*samples: Tensor*, *mapping: Tensor*, *background_densities: Tensor*, *target_lnpdfs: Tensor*, *target_lnpdfs_grads: Tensor*)

Perform the natural gradient estimation, needs to be implemented by the deriving class.

Parameters

- **samples** – tf.Tensor a tensor of shape num_samples x num_dimension containing the samples used for the approximation
- **mapping** – tf.Tensor a one-dimensional tensor of integers, storing for every sample from which component it was sampled.
- **background_densities** – tf.Tensor the log probability density of the background distribution (which was used for sampling the provided samples). A one-dimensional tensor of size num_samples.
- **target_lnpdfs** – tf.Tensor The rewards are given by the log-densities of the target-distribution, $\log p(\mathbf{x})$.
- **target_lnpdfs_grads** – tf.Tensor The gradients of the target_lnpdfs with respect to the samples, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.

Returns

expected_hessian_neg - A tensor of shape num_components x num_dimensions x num_dimensions containing for each component an estimate of the (negated) expected Hessian $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{xx}} \log \frac{p(\mathbf{x}}{q(\mathbf{x}}\right]$

expected_gradient_neg - A tensor of shape num_components x num_dimensions containing for each component an estimate of the (negated) expected gradient $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{x}} \log \frac{p(\mathbf{x}}{q(\mathbf{x}}\right]$

Return type
tuple(tf.Tensor, tf.Tensor)

get_rewards_for_comp(*index: int*, *samples: Tensor*, *mapping: Tensor*, *component_log_densities*, *log_ratios: Tensor*, *log_ratio_grads: Tensor*, *background_densities: Tensor*) → Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]

property requires_gradients: bool

MoreNgEstimator

class gmmvi.optimization.gmmvi_modules.ng_estimator.**MoreNgEstimator**(*temperature*, *model*, *only_use_own_samples: bool*, *initial_l2_regularizer: float*, *use_self_normalized_importance_weights: bool*)

Use compatible function approximation to estimate the natural gradient from a quadratic reward surrogate. See [ALL+15, PTA+19, PS08, SMSM99].

Parameters

- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.
- **model** – *GmmWrapper* The wrapped model where we want to update the components.

- **only_use_own_samples** – bool If true, we do not use importance sampling to update one component based on samples from a different component.
- **initial_l2_regularizer** – float The l2_regularizer is as regularizer during weighted least-squares (ridge regression) for fitting the compatible surrogate.
- **use_self_normalized_importance_weights** – bool if true, use self-normalized importance weighting (normalizing the importance weights such they sum to one), rather than standard importance weighting.

get_expected_hessian_and_grad(*samples: Tensor*, *mapping: Tensor*, *background_densities: Tensor*, *target_lnpdfs: Tensor*, *target_lnpdfs_grads: Tensor*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Estimates the natural gradient using compatible function approximation. This method does not require / make use of the provided gradients, but only uses the function evaluations *target_lnpdfs* for estimating the natural gradient. The method fits a quadratic reward function $\tilde{R}(\mathbf{x}) = \mathbf{x}^T\mathbf{R}\mathbf{x} + \mathbf{x}^T\mathbf{r} + r_0$ to approximate the target distribution using importance-weighted least squares where the targets are given by *target_lnpdfs*, $\log p(\mathbf{x})$. The natural gradient estimate, can then be computed from the coefficients $\mathbf{R}$ and $\mathbf{r}$.

Parameters

- **samples** – tf.Tensor a tensor of shape num_samples x num_dimension containing the samples used for the approximation
- **mapping** – tf.Tensor a one-dimensional tensor of integers, storing for every sample from which component it was sampled.
- **background_densities** – tf.Tensor the log probability density of the background distribution (which was used for sampling the provided samples). A one-dimensional tensor of size num_samples.
- **target_lnpdfs** – tf.Tensor The rewards are given by the (unnormalized) log-densities of the target-distribution, $\log p(\mathbf{x})$.
- **target_lnpdfs_grads** – tf.Tensor The gradients of the target_lnpdfs with respect to the samples (not used), $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.

Returns

expected_hessian_neg - A tensor of shape num_components x num_dimensions x num_dimensions containing for each component an estimate of the (negated) expected Hessian $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{x}\mathbf{x}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$

expected_gradient_neg - A tensor of shape num_components x num_dimensions containing for each component an estimate of the (negated) expected gradient $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{x}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$

Return type
tuple(tf.Tensor, tf.Tensor)

SteinNgEstimator

class gmmvi.optimization.gmmvi_modules.ng_estimator.**SteinNgEstimator**(*temperature*, *model*, *only_use_own_samples: bool*, *use_self_normalized_importance_weights: bool*)

Use Stein's Lemma to estimate the natural gradient using first-order information. See [LKS19b].

Parameters

- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.
- **model** – *GmmWrapper* The wrapped model where we want to update the components.
- **only_use_own_samples** – bool If true, we do not use importance sampling to update one component based on samples from a different component.
- **use_self_normalized_importance_weights** – bool if true, use self-normalized importance weighting (normalizing the importance weights such they sum to one), rather than standard importance weighting.

get_expected_hessian_and_grad(*samples: Tensor*, *mapping: Tensor*, *background_densities: Tensor*, *target_lnpdfs: Tensor*, *target_lnpdfs_grads: Tensor*)

Estimates the natural gradient using Stein's Lemma [LKS19b]. The expected gradient is a simple importance-weighted Monte-Carlo estimate based on the provided *target_lnpdfs_grads* and the gradients of the component log-densities. The expected Hessians are estimated as $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{xx}} \log \frac{p(\mathbf{x}}{q(\mathbf{x}}\right] \approx -\sum_{\mathbf{x}_i} w_i \Sigma^{-1}(\mathbf{x}_i - \mu)\nabla_{\mathbf{x}_i} g_{\mathbf{x}_i}^T$, where $g_{\mathbf{x}_i} = \nabla_{\mathbf{x}_i} \log \frac{p(\mathbf{x}}{q(\mathbf{x}}$ is the gradient of the log-ratio with respect to the corresponding sample.

Parameters

- **samples** – tf.Tensor a tensor of shape num_samples x num_dimension containing the samples used for the approximation
- **mapping** – tf.Tensor a one-dimensional tensor of integers, storing for every sample from which component it was sampled.
- **background_densities** – tf.Tensor the log probability density of the background distribution (which was used for sampling the provided samples). A one-dimensional tensor of size num_samples.
- **target_lnpdfs** – tf.Tensor The rewards are given by the log-densities of the target-distribution, $\log p(\mathbf{x})$.
- **target_lnpdfs_grads** – tf.Tensor The gradients of the target_lnpdfs with respect to the samples, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.

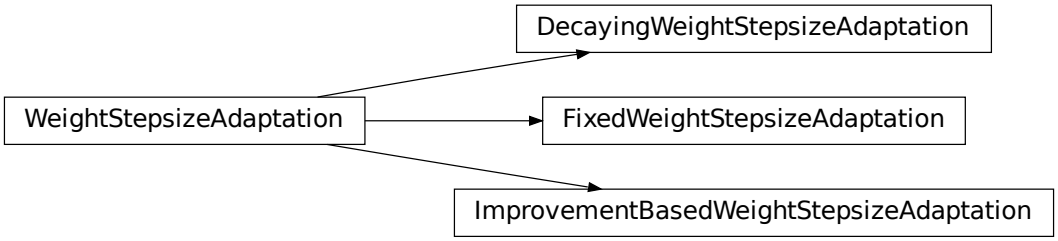
Returns

expected_hessian_neg - A tensor of shape num_components x num_dimensions x num_dimensions containing for each component an estimate of the (negated) expected Hessian $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{xx}} \log \frac{p(\mathbf{x}}{q(\mathbf{x}}\right]$

expected_gradient_neg - A tensor of shape num_components x num_dimensions containing for each component an estimate of the (negated) expected gradient $-\mathbb{E}_{q(\mathbf{x}|o)}\left[\nabla_{\mathbf{x}} \log \frac{p(\mathbf{x}}{q(\mathbf{x}}\right]$

Return type
tuple(tf.Tensor, tf.Tensor)

WeightStepsizeAdaptation



class gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation.**WeightStepsizeAdaptation**(*initial_stepsize: tf.float32*)

This class provides a common interface for adapting the stepsize for the weight update.

There are currently three options for weight stepsize adpatation:

1. The *FixedWeightStepsizeAdaptation* is a dummy-class, that does not do anything.
2. The *DecayingWeightStepsizeAdaptation* uses exponential decay.
3. The *ImprovementBasedWeightStepsizeAdaptation* uses a procedure similar to VIPS [AZN20] to increase the stepsize if the mixture improved during the last updates, and to decrease it otherwise.

Parameters
initial_stepsize – float The initial stepsize for the weight update.

static build_from_config(*config*, *gmm_wrapper*)

This static method provides a convenient way to create a *FixedWeightStepsizeAdaptation*, *DecayingWeightStepsizeAdaptation* or *ImprovementBasedWeightStepsizeAdaptation* depending on the provided config.

Parameters

- **config** – dict The dictionary is typically read from YAML a file, and holds all hyperparameters.
- **gmm_wrapper** – *GmmWrapper* The wrapped model.

update_stepsize()

Update the stepsizes, according to the chosen procedure.

Returns
the updated stepsize.

Return type
float

DecayingWeightStepsizeAdaptation

class gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation.**DecayingWeightStepsizeAdaptation**(*initial tf.floa an- neal- ing_e tf.floa*

This class implements an exponentially decaying stepsize schedule.

See [KNT+18].

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped.
- **annealing_exponent** – float controls how fast the stepsize decays
- **initial_stepsize** – float The initial stepsize for the weight update.

FixedWeightStepsizeAdaptation

class gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation.**FixedWeightStepsizeAdaptation**(*initial_ste tf.float32*

This class is a dummy class, that can be used when we want to keep the stepsize for the weight update constant.

Parameters
initial_stepsize – float The initial stepsize for the weight update.

ImprovementBasedWeightStepsizeAdaptation

class gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation.**ImprovementBasedWeightStepsizeAdaptat**

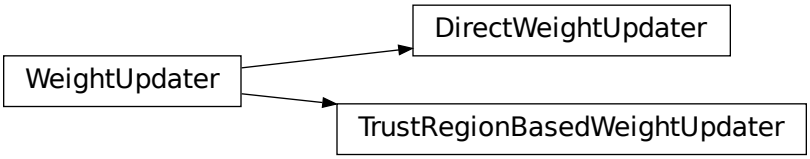
Increases the stepsize if the last weight update increased its reward, decreases it otherwise.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model.
- **initial_stepsize** – float The initial stepsize for the weight update.
- **min_stepsize** – float Do not not decrease the stepsize below this point

- **max_stepsize** – float Do not increase the stepsize above this point
- **stepsize_inc_factor** – float Factor (>1) for increasing the stepsize
- **stepsize_dec_factor** – float Factor in the range [0, 1] for decreasing the stepsize

WeightUpdater



class gmmvi.optimization.gmmvi_modules.weight_updater.**WeightUpdater**(*model:* GmmWrapper, *temperature: float*, *use_self_normalized_importance_weights: bool*)

This class provides a common interface for updating the weights of the mixture model.

It currently supports two options:

1. The *DirectWeightUpdater* straightforwardly applies a natural gradient update using the given stepsize.
2. The *TrustRegionBasedWeightUpdater* treats the stepsize as a trust-region constraint between the current distribution over weights and the updated distribution, and performs the largest step in the direction of the natural gradient that confines to this constraint.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to update the weights.
- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.
- **use_self_normalized_importance_weights** – bool if true, use self-normalized importance weighting (normalizing the importance weights such they sum to one), rather than standard importance weighting for estimating the natural gradient.

static build_from_config(*config*, *gmm_wrapper*)

This static method provides a convenient way to create a *DirectWeightUpdater* or *TrustRegionBasedWeightUpdater* depending on the provided config.

Parameters

- **config** – dict The dictionary is typically read from YAML a file, and holds all hyperparameters.
- **gmm_wrapper** – *GmmWrapper* The wrapped model for which we want to update the weights.

update_weights(*samples: Tensor*, *background_mixture_densities: Tensor*, *target_lnpdfs: Tensor*, *stepsize: float*)

Computes the importance weights and uses them to estimate the natural gradient. Performs a natural gradient step using the given stepsize.

Parameters

- **samples** – tf.Tensor The samples for which the *background_mixture_densities* and *target_lnpdfs* were evaluated. Needed for computing the importance weights.
- **background_mixture_densities** – tf.Tensor The log_densities of the *samples* for the distribution that was effectively used for obtain the provided *samples*. Needed for computing the importance weights.
- **target_lnpdfs** – tf.Tensor The log densities of the target distribution evaluated for the provided *samples*, $\log p(\mathbf{x})$.
- **stepsize** – float The stepsize that should be used for performing the weight update.

DirectWeightUpdater

class gmmvi.optimization.gmmvi_modules.weight_updater.**DirectWeightUpdater**(*model: GmmWrapper*, *temperature: float*, *use_self_normalized_importance_weig bool*)

This class can be used for directly updating the weights along the natural gradient, using the given stepsize.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to update the weights.
- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.
- **use_self_normalized_importance_weights** – bool if true, use self-normalized importance weighting (normalizing the importance weights such they sum to one), rather than standard importance weighting for estimating the natural gradient.

_update_weights_from_expected_log_ratios(*expected_log_ratios: Tensor*, *stepsize: tf.float32*)

Directly uses the stepsize to update the weights towards the expected_log_ratios

Parameters

- **expected_log_ratios** – tf.Variable(tf.float32) A vector containing an (MC-)estimate of $\mathbb{E}_{q(\mathbf{x}|o)}\left[\log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$, for every component o.
- **stepsize** – tf.float32 The stepsize β, the new weights are proportional to $\exp(\text{old\_log\_weights} + \beta * \text{expected\_log\_ratios})$.

TrustRegionBasedWeightUpdater

class gmmvi.optimization.gmmvi_modules.weight_updater.**TrustRegionBasedWeightUpdater**(*model: GmmWrapper*, *temperature: float*, *use_self_normalized_imp bool*)

This class can be used for performing the weight update by treating the stepsize as a KL constraint.

Constrains the KL between the updated weights and the current weights $\mathrm{KL}(q_{\text{new}}(o)||q(o))$.

Parameters

- **gmm_wrapper** – *GmmWrapper* The wrapped model where we want to update the weights.
- **temperature** – float Usually temperature=1., can be used to scale the importance of maximizing the model entropy.
- **use_self_normalized_importance_weights** – bool if true, use self-normalized importance weighting (normalizing the importance weights such they sum to one), rather than standard importance weighting for estimating the natural gradient.

_bracketing_search(*expected_log_ratios: Tensor*, *kl_bound: tf.float32*, *lower_bound: tf.float32*, *upper_bound: tf.float32*) → [tf.float32, tf.float32, <class 'tensorflow.python.framework.ops.Tensor'>]

This method finds the largest stepsize eta, such that the updated weight distribution stays within a KL-constrained trust-region around the current distribution. Here we use a simple bracketing search, which can be efficiently performed within a Tensorflow graph. The procedure simple keeps track of a lower bound and an upper bound on the optimal stepsize and recursively evaluates the arithmetic mean of both bounds. If this mean-stepsize results in a too large KL divergence, it becomes the new lower bound; otherwise the new upper bound.

Parameters

- **expected_log_ratios** – tf.Tensor A vector containing an (MC-)estimate of $\mathbb{E}_{q(\mathbf{x}|o)}\left[\log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$, for every component o.
- **kl_bound** – tf.float32 The trust region constraint
- **lower_bound** – tf.float32 The initial lower bound on the stepsize
- **upper_bound** – The initial upper bound on the stepsize

Returns

new_lower_bound - The lower bound after a stopping criterion was reached.

new_upper_bound - The upper bound after a stopping criterion was reached.

new_log_weights - log of the updated weights, $\log(q_{\text{new}}(o))$.

Return type
tuple(tf.float32, tf.float32, tf.Tensor)

_update_weights_from_expected_log_ratios(*expected_log_ratios*, *kl_bound*)

Perform the weight update, treating the stepsize as constraint on the KL-divergence.

Parameters

- **expected_log_ratios** – tf.Variable(tf.float32) A vector containing an (MC-)estimate of $\mathbb{E}_{q(\mathbf{x}|o)}\left[\log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$, for every component o.
- **stepsize** – tf.float32 The stepsize ϵ, the new weights will satisfy $\mathrm{KL}(q_{\text{new}}(o)||q(o)) < \epsilon$.

kl(*eta: tf.float32*, *component_rewards: Tensor*) → [tf.float32, <class 'tensorflow.python.framework.ops.Tensor'>]

Computes the Kullback-Leibler divergence between the updated component and current component, when updating with stepsize eta along the natural gradient.

Parameters

- **eta** – tf.float32 The stepsize for which the KL divergence should be computed.
- **component_rewards** – tf.float32 A tensor containing an MC-estimate of the expected reward (expected logratios) of each component, $R(o) = \mathbb{E}_{q(\mathbf{x}|o)}\left[\log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right]$

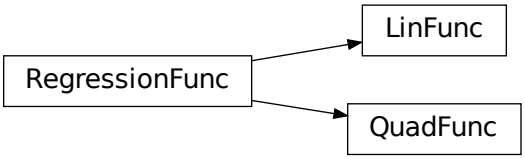
Returns

kl - a float corresponding to the KL between the updated and previous weight distribution.

new_log_weights - log of the updated weights, $\log(q_{\text{new}}(o))$.

Return type
tuple(float, tf.Tensor)

3.3.2 Least Squares



class gmmvi.optimization.least_squares.**LinFunc**(*reg_fact: float*)

This class can be used to learn a function that is linear in the inputs.

Parameters
reg_fact – float coefficient for ridge regularization

class gmmvi.optimization.least_squares.**QuadFunc**(*dim: int*)

This class can be used to learn a function that is quadratic in the inputs (or linear in the quadratic features). The approximation takes the form: $x^T Rx + x^T r + r_0$

Parameters
dim – int the dimension of x

fit_quadratic(*regularizer: float*, *num_samples: int*, *inputs: Tensor*, *outputs: Tensor*, *weights: Optional[Tensor] = None*, *sample_mean: Optional[Tensor] = None*, *sample_chol_cov: Optional[Tensor] = None*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Fits the quadratic model.

Parameters

- **regularizer** – float Coefficient for ridge regression
- **num_samples** – int Number of input samples (we could get this from *inputs*)
- **inputs** – tf.Tensor A two-dimensional tensor containing the inputs x.
- **outputs** – tf.Tensor A one-dimensional tensor containing the targets / dependant variables.
- **weights** – tf.Tensor (importance) weights used for weighted least-squares
- **sample_mean** – tf.Tensor Mean of the Gaussian distribution that sampled the input (used for whitening)
- **sample_chol_cov** – tf.Tensor Cholesky matrix of the Gaussian distribution that sampled the input (used for whitening)

Returns

quad_term - the matrix R

lin_term - the vector r

const_term - the scalar bias

Return type
tuple(tf.Tensor, tf.Tensor, tf.Tensor)

class gmmvi.optimization.least_squares.**RegressionFunc**(*bias_entry: Optional[int] = None*)

Base class for least-square regression

Parameters
bias_entry – int index of the weight that corresponds to the constant offset (will not get regularized)

fit(*regularizer: float*, *num_samples: int*, *inputs: Tensor*, *outputs: Tensor*, *weights: Optional[Tensor] = None*) → Tensor

Compute the coefficients of the linear model.

Parameters

- **regularizer** – float ridge coefficient
- **num_samples** – int number of samples (could be obtained from *inputs*)
- **inputs** – tf.Tensor the data / samples
- **outputs** – tf.Tensor the targets / dependent variables
- **weights** – tf.Tensor or None (importance) weights for weighted least-squares

Returns
the learned parameters of the linear model

Return type
tf.Tensor

class gmmvi.optimization.sample_db.**SampleDB**(*dim*, *diagonal_covariances*, *keep_samples*, *max_samples=None*)

A database for storing samples and meta-information.

Along the samples, we also store

1. The parameters of the Gaussian distribution that were used for obtaining each sample
2. log-density evaluations of the target distribution, $\log p(\mathbf{x})$
3. (if available), gradients of the log-densites of the target distribution, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$

Parameters

- **dim** – int dimensionality of the samples to be stored
- **diagonal_covariances** – bool True, if the samples are always drawn from Gaussians with diagonal covariances (saves memory)
- **keep_samples** – bool If this is False, the samples are not actually stored
- **max_samples** – int Maximal number of samples that are stored. If adding new samples would exceed this limit, every N-th sample in the database gets deleted.

add_samples(*samples*, *means*, *chols*, *target_lnpdfs*, *target_grads*, *mapping*)

Add the given samples to the database.

Parameters

- **samples** – tf.Tensor a two-dimensional tensor of shape num_samples x num_dimensions containing the samples to be added.
- **means** – tf.Tensor a two-dimensional tensor containing for each Gaussian distribution that was used for obtaining the samples the corresponding mean. The first dimension of the tensor can be smaller than the number of samples, if several samples where drawn from the same Gaussian (see the parameter *mapping*).
- **chols** – tf.Tensor a three-dimensional tensor containing for each Gaussian distribution that was used for obtaining the samples the corresponding Cholesky matrix. The first dimension of the tensor can be smaller than the number of samples, if several samples where drawn from the same Gaussian (see the parameter *mapping*).
- **target_lnpdfs** – tf.Tensor a one-dimensional tensor containing the log-densities of the (unnormalized) target distribution, $\log p(\mathbf{x})$.
- **target_grads** – tf.Tensor a two-dimensional tensor containing the gradients of the log-densities of the (unnormalized) target distribution, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.
- **mapping** – tf.Tensor a tensor of size number_of_samples, which corresponds for every sample the index to *means* and *chols* that corresponds to the Gaussian distribution that was used for drawing that sample.

static build_from_config(*config*, *num_dimensions*)

A static method to conveniently create a *SampleDB* from a given config dictionary.

Parametes:

config: dict
The dictionary is typically read from YAML a file, and holds all hyperparameters.

num_dimensions: int
dimensionality of the samples to be stored

`evaluate_background`(*weights*, *means*, *chols*, *inv_chols*, *samples*)

Evaluates the log-densities of the given samples on a GMM with the given parametes. This function is implemented in a memory-efficient way to scale to mixture models with many components.

Parameters

`weights` –

tf.Tensor
The weights of the GMM that should be evaluated

means: tf.Tensor
The means of the GMM that should be evaluated

chols: tf.Tensor
The Cholesky matrices of the GMM that should be evaluated

inv_chols: tf.Tensor
The inverse of abovementioned *chols*

samples: tf.Tensor
The samples to be evaluated.

`get_newest_samples`(*N*)

Returns (up to) the N newest samples, and their meta-information.

Returns

log_pdfs - the log-density of the GMM that was effectively used for drawing the samples (used for importance sampling)

active_sample - the selected samples

active_mapping - contains for every sample the index of the component that was used for drawing it

active_target_lnpdfs - log-density evaluations of the target distribution for the selected samples

active_target_grads - gradients evaluations of the log-density of the target distribution for the selected samples

Return type
tuple(tf.Tensor, tf.Tensor, tf.Tensor, tf.Tensor, tf.Tensor)

`get_random_sample`(*N: int*)

Get N random samples from the database.

Parameters

`N` – int abovementioned N

Returns

tuple(tf.Tensor, tf.Tensor)

samples - the chosen samples

target_lnpdfs - the corresponding log densities of the target distribution

`remove_every_nth_sample`(*N: int*)

Deletes Every N-th sample from the database and the associated meta information.

Parameters

`N` – int abovementioned N

class gmmvi.optimization.gmmvi.**GMMVI**(*model:* GmmWrapper, *sample_db:* SampleDB, *temperature:* *tf.float32*, *sample_selector:* SampleSelector, *num_component_adapter:* ComponentAdaptation, *component_stepsize_adapter:* ComponentStepsizeAdaptation, *ng_estimator:* NgEstimator, *ng_based_updater:* NgBasedComponentUpdater, *weight_stepsize_adapter:* WeightStepsizeAdaptation, *weight_updater:* WeightUpdater)

The main class of this framework, which provides the functionality to perform a complete update step for the GMM.

Responsibilities for performing the necessary sub-steps (sample selection, natural gradient estimation, etc.) and for keeping track of data are delegated to the `GMMVI Modules`, the *SampleDB* and *GmmWrapper*. Hence, this class acts mainly as a manager between these components.

Parameters

- **model** – *GmmWrapper* The (wrapped) model that we are optimizing.
- **sample_db** – *SampleDB* The database for storing samples.
- **temperature** – tf.float32 The temperature parameter β for weighting the model entropy $H(q)$ in the optimization problem $\arg\max_q \mathbb{E}\left[\log(\tilde{p}(x))\right] + \beta H(q)$.
- **sample_selector** – *SampleSelector* The SampleSelector for selecting the samples that are used during each iteration.
- **num_component_adapter** – *NumComponentAdaptation* The NumComponentAdapter used for adding and deleting components.
- **component_stepsize_adapter** – *ComponentStepsizeAdaptation* The ComponentStepsizeAdapter for choosing the learning rates for the component update.
- **ng_estimator** – *NgEstimator* The NgEstimator for estimating the natural gradient for the component update.
- **ng_based_updater** – *NgBasedComponentUpdater* The NgBasedComponentUpdater for updating the components based on the estimated natural gradients.
- **weight_stepsize_adapter** – *WeightStepsizeAdaptation* The WeightStepsizeAdapter for choosing the learning rate for updating the mixture weights.
- **weight_updater** – *WeightUpdater* The NgBasedComponentUpdater for updating the components based on the estimated natural gradients.

static build_from_config(*config:* *dict*, *target_distribution:* LNPDF, *model:* GmmWrapper)

Create a *GMMVI* instance from a configuration dictionary.

This static method provides a convenient way to create a *GMMVI* instance, based on an initial GMM (a *wrapped model*), a *target_distribution* and a dictionary containing the types and parameters of the `GMMVI modules`.

Parameters

- **config** – dict The dictionary should contain for each `GMMVI module` an entry of the form XXX_type (a string) and XXX_config (a dict) for specifying the type of each module, and the module-specific hyperparameters. For example, the dictionary could contain sample_selector_type={"component-based"} and sample_selector_config={"desired_samples_per_component": 100, "ratio_reused_samples_to_desired": 2.}. Refer to the example yml-configs, or to the individual GMMVI module for the expected parameters, and type-strings.

- **target_distribution** – *LNPDF* The (unnormalized) target distribution that we want to approximate.
- **model** – *GmmWrapper* The (wrapped) model that we are optimizing.

train_iter()

Perform a single training iteration.

This method does not take any parameters, nor does it return anything. However, it may have several effects, such as

- drawing new samples from the `model` and evaluating them on the target distribution,
- updating the `gmmvi.optimization.gmmvi.GMMVI.model` parameters,
- adapting learning rates, etc.

class gmmvi.gmmvi_runner.**GmmviRunner**(*config*, *log_metrics_interval*)

This class runs *GMMVI*, but also evaluates learning metrics and provides logging functionality.

Parameters

- **config** – dict A dictionary containing the hyperparameters and environment specifications.
- **log_metrics_interval** – int metrics that take non-negligible overhead are evaluated ever *log_metrics_interval* iterations.

static build_from_config(*config: dict*)

Create a *GMMVI* instance from a configuration dictionary.

This static method provides a convenient way to create a *GMMVI* instance, based on a dictionary containing the types and parameters of the `GMMVI modules`.

Parameters

config – dict The dictionary should contain for each `GMMVI module` an entry of the form XXX_type (a string) and XXX_config (a dict) for specifying the type of each module, and the module-specific hyperparameters. For example, the dictionary could contain sample_selector_type={"component-based"} and sample_selector_config={"desired_samples_per_component": 100, "ratio_reused_samples_to_desired": 2.}. Refer to the example yml-configs, or to the individual GMMVI module for the expected parameters, and type-strings.

finalize()

Can be called after learning. Saves the final model parameters to the hard drive.

get_cheap_metrics()

Returns a dictionary of 'cheap' metrics, e.g. the current number of components, that we can obtain after every iteration without adding computational overhead.

Returns

A dictionary containing cheap metrics.

Return type

dict

get_expensive_metrics()

Computes 'expensive' metrics, such as plots, test-set evaluations, etc. Some of these metrics can be task-specific (see *LNPDF.expensive_metrics()*).

Returns

A dictionary containing expensive metrics.

Return type
dict

`get_samples_and_entropy`(*num_samples*)

Draws *num_samples* from the model and uses them to estimate the model's entropy.

Parameters
num_samples – int Number of samples to be drawn

Returns

test_samples - The drawn samples

entropy - MC estimate of the entropy

Return type
tuple(tf.Tensor, float)

`iterate_and_log`(*n: int*) → dict

Perform one learning iteration and computes and logs metrics.

Parameters
n – int The current iteration

Returns
A dictionary containing metrics and plots that we want to log.

Return type
dict

`log_to_disk`(*n: int*)

Saves the model parameters to the hard drive

Parameters
n – int The current iteration

CHAPTER

FOUR

REFERENCES

BIBLIOGRAPHY

[ALL+15] A. Abdolmaleki, R. Lioutikov, N. Lua, L. Paulo Reis, J. Peters, and G. Neumann. Model-based relative entropy stochastic search. In *Advances in Neural Information Processing Systems (NeurIPS)*, 153–154. 2015.

[AZN18] O. Arenz, M. Zhong, and G. Neumann. Efficient gradient-free variational inference using policy search. In *International Conference on Machine Learning (ICML)*. 2018.

[AZN20] Oleg Arenz, Mingjun Zhong, and Gerhard Neumann. Trust-region variational inference with gaussian mixture models. *Journal of Machine Learning Research*, 21(163):1–60, 2020. URL: http://jmlr.org/papers/v21/19-524.html.

[GBR+12] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola. A kernel two-sample test. *Journal of Machine Learning Research (JMLR)*, 13:723–773, March 2012.

[KNT+18] Mohammad Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. Fast and scalable Bayesian deep learning by weight-perturbation in Adam. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of Proceedings of Machine Learning Research, 2611–2620. PMLR, 10–15 Jul 2018.

[Lic13] M. Lichman. UCI machine learning repository. 2013. URL: http://archive.ics.uci.edu/ml.

[LKS19a] Wu Lin, Mohammad Emtiyaz Khan, and Mark Schmidt. Fast and simple natural-gradient variational inference with mixture of exponential-family approximations. In *International Conference on Machine Learning*, 3992–4002. PMLR, 2019.

[LKS19b] Wu Lin, Mohammad Emtiyaz Khan, and Mark Schmidt. Stein's lemma for the reparameterization trick with exponential family mixtures. *arXiv preprint arXiv:1910.13398*, 2019.

[LSK20] Wu Lin, Mark Schmidt, and Mohammad Emtiyaz Khan. Handling the positive-definite constraint in the Bayesian learning rule. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of Proceedings of Machine Learning Research, 6116–6126. PMLR, 13–18 Jul 2020.

[PTA+19] J. Pajarinen, H.L. Thai, R. Akrour, J. Peters, and G. Neumann. Compatible natural gradient policy search. *Machine Learning (MLJ)*, pages 1443–1466, 2019.

[PS08] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008.

[SMSM99] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

PYTHON MODULE INDEX

g

gmmvi.models.diagonal_gmm, 20
gmmvi.models.full_cov_gmm, 18
gmmvi.models.gmm, 13
gmmvi.models.gmm_wrapper, 21
gmmvi.optimization.least_squares, 59

INDEX

Symbols

_bracketing_search() (*gmmvi.optimization.gmmvi_modules.weight_updater.TrustRegionBasedWeightUpdater method*), 58

_update_weights_from_expected_log_ratios() (*gmmvi.optimization.gmmvi_modules.weight_updater.DirectWeightUpdater method*), 57

_update_weights_from_expected_log_ratios() (*gmmvi.optimization.gmmvi_modules.weight_updater.TrustRegionBasedWeightUpdater method*), 58

A

adapt_number_of_components() (*gmmvi.optimization.gmmvi_modules.component_adaptation.FixedComponentAdaptation method*), 39

adapt_number_of_components() (*gmmvi.optimization.gmmvi_modules.component_adaptation.VipsComponentAdaptation method*), 41

add_at_best_location() (*gmmvi.optimization.gmmvi_modules.component_adaptation.VipsComponentAdaptation method*), 41

add_component() (*gmmvi.models.diagonal_gmm.DiagonalGMM method*), 20

add_component() (*gmmvi.models.full_cov_gmm.FullCovGMM method*), 18

add_component() (*gmmvi.models.gmm.GMM method*), 13

add_component() (*gmmvi.models.gmm_wrapper.GmmWrapper method*), 21

add_new_component() (*gmmvi.optimization.gmmvi_modules.component_adaptation.VipsComponentAdaptation method*), 41

add_samples() (*gmmvi.optimization.sample_db.SampleDB method*), 61

apply_NG_update() (*gmmvi.optimization.gmmvi_modules.ng_based_component_updater.DirectNgBasedComponentUpdater method*), 46

apply_NG_update() (*gmmvi.optimization.gmmvi_modules.ng_based_component_updater.KLConstrainedNgBasedComponentUpdater method*), 47

apply_NG_update() (*gmmvi.optimization.gmmvi_modules.ng_based_component_updater.TrustRegionBasedWeightUpdater method*), 45

apply_NG_update() (*gmmvi.optimization.gmmvi_modules.ng_based_component_updater.VipsNgBasedComponentUpdater method*), 49

average_loss() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 31

avg_bayesian_inference_loss() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 31

avg_bayesian_inference_test_loss() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 31

B

bayesian_inference_test_loss() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 31

BNN_LNPDF (*class in gmmvi.experiments.target_distributions.bnn*), 31

bracketing_search() (*gmmvi.optimization.gmmvi_modules.ng_based_component_updater.KLConstrainedNgBasedComponentUpdater method*), 47

build_from_config() (*gmmvi.gmmvi_runner.GmmviRunner static method*), 64

build_from_config() (*gmmvi.models.gmm_wrapper.GmmWrapper static method*), 22

build_from_config() (*gmmvi.optimization.gmmvi.GMMVI static method*), 63

build_from_config() (*gmmvi.optimization.gmmvi_modules.component_adaptation.ComponentAdaptation static method*), 39

build_from_config() (*gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.StepsizeAdaptation static method*), 42

build_from_config() (*gmmvi.optimization.gmmvi_modules.ng_based_component_updater.NgBasedComponentUpdater static method*), 46
build_from_config() (*gmmvi.optimization.gmmvi_modules.ng_estimator.NgEstimator static method*), 50
build_from_config() (*gmmvi.optimization.gmmvi_modules.sample_selector.SampleSelector static method*), 34
build_from_config() (*gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation.WeightStepsizeAdaptation static method*), 54
build_from_config() (*gmmvi.optimization.gmmvi_modules.weight_updater.WeightUpdater static method*), 56
build_from_config() (*gmmvi.optimization.sample_db.SampleDB static method*), 61

C

can_sample() (*gmmvi.experiments.target_distributions.gmm.GMM_LNPDF method*), 25
can_sample() (*gmmvi.experiments.target_distributions.lnpdf.LNPDF method*), 23
can_sample() (*gmmvi.experiments.target_distributions.student_t_mixture.StudentTMixture method*), 27
component_entropies() (*gmmvi.models.gmm.GMM method*), 14
component_log_densities() (*gmmvi.models.diagonal_gmm.DiagonalGMM method*), 20
component_log_densities() (*gmmvi.models.full_cov_gmm.FullCovGMM method*), 19
component_log_densities() (*gmmvi.models.gmm.GMM method*), 14
component_log_density() (*gmmvi.models.full_cov_gmm.FullCovGMM method*), 19
component_log_density() (*gmmvi.models.gmm.GMM method*), 14
component_log_density_and_grad() (*gmmvi.models.gmm.GMM method*), 14
component_marginal_log_densities() (*gmmvi.models.full_cov_gmm.FullCovGMM method*), 19
component_marginal_log_densities() (*gmmvi.models.gmm.GMM method*), 14
ComponentAdaptation (*class in gmmvi.optimization.gmmvi_modules.component_adaptation*), 38
ComponentStepsizeAdaptation (*class in gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation*), 42
compute_MMD() (*gmmvi.experiments.evaluation.mmd.MMD method*), 33
covs (*gmmvi.models.diagonal_gmm.DiagonalGMM property*), 20
covs (*gmmvi.models.full_cov_gmm.FullCovGMM property*), 19
covs (*gmmvi.models.gmm.GMM property*), 15
create_bnn() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 31

D

DecayingComponentStepsizeAdaptation (*class in gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation*), 43
DecayingWeightStepsizeAdaptation (*class in gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation*), 55
delete_bad_components() (*gmmvi.optimization.gmmvi_modules.component_adaptation.VipsCo method*), 41
det_cov() (*gmmvi.models.gmm.GMM method*), 15
diagonal_gaussian_log_pdf() (*gmmvi.models.diagonal_gmm.DiagonalGMM static method*), 21
DiagonalGMM (*class in gmmvi.models.diagonal_gmm*), 20
DirectNgBasedComponentUpdater (*class in gmmvi.optimization.gmmvi_modules.ng_based_component_updater*), 46
DirectWeightUpdater (*class in gmmvi.optimization.gmmvi_modules.weight_updater*), 57

E

evaluate_background() (*gmmvi.optimization.sample_db.SampleDB method*), 61
expensive_metrics() (*gmmvi.experiments.target_distributions.gmm.GMM_LNPDF method*), 25
expensive_metrics() (*gmmvi.experiments.target_distributions.lnpdf.LNPDF method*), 24
expensive_metrics() (*gmmvi.experiments.target_distributions.logistic_regression.LogisticR method*), 29
expensive_metrics() (*gmmvi.experiments.target_distributions.planar_robot.PlanarRobot method*), 30
expensive_metrics() (*gmmvi.experiments.target_distributions.student_t_mixture.StudentTM method*), 27

F

finalize() (*gmmvi.gmmvi_runner.GmmviRunner method*), 64
fit() (*gmmvi.optimization.least_squares.RegressionFunc method*), 60
fit_quadratic() (*gmmvi.optimization.least_squares.QuadFunc method*), 59
FixedComponentAdaptation (*class in gmmvi.optimization.gmmvi_modules.component_adaptation*), 39
FixedComponentStepsizeAdaptation (*class in gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation*), 43
FixedWeightStepsizeAdaptation (*class in gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation*), 55
forward_from_weight_vector() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 31
forward_kinematics() (*gmmvi.experiments.target_distributions.planar_robot.PlanarRobot method*), 30
FullCovGMM (*class in gmmvi.models.full_cov_gmm*), 18

G

gaussian_entropy() (*gmmvi.models.diagonal_gmm.DiagonalGMM method*), 21
gaussian_entropy() (*gmmvi.models.full_cov_gmm.FullCovGMM method*), 19
gaussian_entropy() (*gmmvi.models.gmm.GMM method*), 15
get_average_entropy() (*gmmvi.models.gmm.GMM method*), 15
get_cheap_metrics() (*gmmvi.gmmvi_runner.GmmviRunner method*), 64
get_effective_samples() (*gmmvi.optimization.gmmvi_modules.sample_selector.EmSampleSelector method*), 35
get_effective_samples() (*gmmvi.optimization.gmmvi_modules.sample_selector.VipsSampleSelector method*), 37
get_expected_hessian_and_grad() (*gmmvi.optimization.gmmvi_modules.ng_estimator.MoreNgEstimator method*), 52
get_expected_hessian_and_grad() (*gmmvi.optimization.gmmvi_modules.ng_estimator.NgEstimator method*), 50
get_expected_hessian_and_grad() (*gmmvi.optimization.gmmvi_modules.ng_estimator.SteinNgEstimator method*), 53
get_expensive_metrics() (*gmmvi.gmmvi_runner.GmmviRunner method*), 64
get_newest_samples() (*gmmvi.optimization.sample_db.SampleDB method*), 62
get_num_dimensions() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 31
get_num_dimensions() (*gmmvi.experiments.target_distributions.gmm.GMM_LNPDF method*), 26
get_num_dimensions() (*gmmvi.experiments.target_distributions.lnpdf.LNPDF method*), 24
get_num_dimensions() (*gmmvi.experiments.target_distributions.logistic_regression.LogisticR method*), 28
get_num_dimensions() (*gmmvi.experiments.target_distributions.planar_robot.PlanarRobot method*), 30
get_num_dimensions() (*gmmvi.experiments.target_distributions.student_t_mixture.StudentTM method*), 27
get_random_sample() (*gmmvi.optimization.sample_db.SampleDB method*), 62
get_rewards_for_comp() (*gmmvi.optimization.gmmvi_modules.ng_estimator.NgEstimator method*), 51
get_samples_and_entropy() (*gmmvi.gmmvi_runner.GmmviRunner method*), 65
GMM (*class in gmmvi.models.gmm*), 13
GMM_LNPDF (*class in gmmvi.experiments.target_distributions.gmm*), 25
GMMVI (*class in gmmvi.optimization.gmmvi*), 62
gmmvi.models.diagonal_gmm
    module, 20
gmmvi.models.full_cov_gmm
    module, 18
gmmvi.models.gmm
    module, 13
gmmvi.models.gmm_wrapper
    module, 21
gmmvi.optimization.least_squares
    module, 59
GmmviRunner (*class in gmmvi.gmmvi_runner*), 64
GmmWrapper (*class in gmmvi.models.gmm_wrapper*), 21

I

ImprovementBasedComponentStepsizeAdaptation (*class in gm-*

mvi.optimization.gmmvi_modules.component_stepsize_adaptation.ComponentStepsizeAdaptation method), 44
ImprovementBasedWeightStepsizeAdaptation (*class in gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation*), 55
iterate_and_log() (*gmmvi.gmmvi_runner.GmmviRunner method*), 65

K

kl() (*gmmvi.optimization.gmmvi_modules.ng_based_component_updater.KLConstrainedNgBasedComponentUpdater method*), 48
kl() (*gmmvi.optimization.gmmvi_modules.weight_updater.TrustRegionBasedWeightUpdater method*), 59
KLConstrainedNgBasedComponentUpdater (*class in gmmvi.optimization.gmmvi_modules.ng_based_component_updater*), 47

L

likelihood() (*gmmvi.experiments.target_distributions.planar_robot.PlanarRobot method*), 31
likelihood_batch() (*gmmvi.experiments.target_distributions.logistic_regression.LogisticRegression_minibatch method*), 29
LinFunc (*class in gmmvi.optimization.least_squares*), 59
LinSampleSelector (*class in gmmvi.optimization.gmmvi_modules.sample_selector*), 34
LNPDF (*class in gmmvi.experiments.target_distributions.lnpdf*), 23
log_densities_also_individual() (*gmmvi.models.gmm.GMM method*), 15
log_density() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 31
log_density() (*gmmvi.experiments.target_distributions.gmm.GMM_LNPDF method*), 26
log_density() (*gmmvi.experiments.target_distributions.lnpdf.LNPDF method*), 24
log_density() (*gmmvi.experiments.target_distributions.logistic_regression.LogisticRegression method*), 28
log_density() (*gmmvi.experiments.target_distributions.logistic_regression.LogisticRegression_minibatch method*), 29
log_density() (*gmmvi.experiments.target_distributions.planar_robot.PlanarRobot method*), 31
log_density() (*gmmvi.experiments.target_distributions.student_t_mixture.StudentTMixture method*), 28
log_density() (*gmmvi.models.gmm.GMM method*), 16
log_density_and_grad() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 32
log_density_and_grad() (*gmmvi.experiments.target_distributions.lnpdf.LNPDF method*), 24
log_density_and_grad() (*gmmvi.models.gmm.GMM method*), 16
log_density_fb() (*gmmvi.experiments.target_distributions.logistic_regression.LogisticRegression method*), 30
log_likelihood() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 32
log_likelihood() (*gmmvi.experiments.target_distributions.logistic_regression.LogisticRegression method*), 29
log_likelihood_and_grad() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 32
log_likelihood_old() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 32
log_prior() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 32
log_prior_and_grad() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 32
log_to_disk() (*gmmvi.gmmvi_runner.GmmviRunner method*), 65
LogisticRegression (*class in gmmvi.experiments.target_distributions.logistic_regression*), 28
LogisticRegression_minibatch (*class in gmmvi.experiments.target_distributions.logistic_regression*), 29

M

marginal_log_density() (*gmmvi.experiments.target_distributions.gmm.GMM_LNPDF method*), 26
marginal_log_density() (*gmmvi.experiments.target_distributions.student_t_mixture.StudentTMixture method*), 28
marginal_log_density() (*gmmvi.models.gmm.GMM method*), 16
MMD (*class in gmmvi.experiments.evaluation.mmd*), 32
module
gmmvi.models.diagonal_gmm, 20
gmmvi.models.full_cov_gmm, 18
gmmvi.models.gmm, 13
gmmvi.models.gmm_wrapper, 21
gmmvi.optimization.least_squares, 59
MoreNgEstimator (*class in gmmvi.optimization.gmmvi_modules.ng_estimator*), 51

N

NgBasedComponentUpdater (*class in gm-*

mvi.optimization.gmmvi_modules.ng_based_component_updater), 45
NgBasedComponentUpdaterIblr (*class in gmmvi.optimization.gmmvi_modules.ng_based_component_updater*), 49
NgEstimator (*class in gmmvi.optimization.gmmvi_modules.ng_estimator*), 50
num_components (*gmmvi.models.gmm.GMM property*), 17

P

PlanarRobot (*class in gmmvi.experiments.target_distributions.planar_robot*), 30
prepare_data() (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF method*), 32
prior_std (*gmmvi.experiments.target_distributions.bnn.BNN_LNPDF property*), 32
prior_std (*gmmvi.experiments.target_distributions.logistic_regression.LogisticRegression property*), 29

Q

QuadFunc (*class in gmmvi.optimization.least_squares*), 59

R

RegressionFunc (*class in gmmvi.optimization.least_squares*), 60
remove_component() (*gmmvi.models.gmm.GMM method*), 17
remove_component() (*gmmvi.models.gmm_wrapper.GmmWrapper method*), 22
remove_every_nth_sample() (*gmmvi.optimization.sample_db.SampleDB method*), 62
replace_components() (*gmmvi.models.gmm.GMM method*), 17
replace_weights() (*gmmvi.models.gmm.GMM method*), 17
replace_weights() (*gmmvi.models.gmm_wrapper.GmmWrapper method*), 22
requires_gradients (*gmmvi.optimization.gmmvi_modules.ng_estimator.NgEstimator property*), 51

S

safe_for_tf_graph (*gmmvi.experiments.target_distributions.lnpdf.LNPDF property*), 24
sample() (*gmmvi.experiments.target_distributions.gmm.GMM_LNPDF method*), 26
sample() (*gmmvi.experiments.target_distributions.lnpdf.LNPDF method*), 25
sample() (*gmmvi.experiments.target_distributions.student_t_mixture.StudentTMixture_LNPDF method*), 28
sample() (*gmmvi.models.gmm.GMM method*), 17
sample_categorical() (*gmmvi.models.gmm.GMM method*), 17
sample_from_component() (*gmmvi.models.diagonal_gmm.DiagonalGMM method*), 21
sample_from_component() (*gmmvi.models.full_cov_gmm.FullCovGMM method*), 20
sample_from_component() (*gmmvi.models.gmm.GMM method*), 18
sample_from_components() (*gmmvi.models.gmm.GMM method*), 18
sample_from_components_no_shuffle() (*gmmvi.models.gmm.GMM method*), 18
sample_where_needed() (*gmmvi.optimization.gmmvi_modules.sample_selector.LinSampleSelector method*), 35
sample_where_needed() (*gmmvi.optimization.gmmvi_modules.sample_selector.VipsSampleSelector method*), 37
SampleDB (*class in gmmvi.optimization.sample_db*), 60
SampleSelector (*class in gmmvi.optimization.gmmvi_modules.sample_selector*), 33
select_samples() (*gmmvi.optimization.gmmvi_modules.sample_selector.LinSampleSelector method*), 36
select_samples() (*gmmvi.optimization.gmmvi_modules.sample_selector.SampleSelector method*), 34
select_samples() (*gmmvi.optimization.gmmvi_modules.sample_selector.VipsSampleSelector method*), 37
select_samples_for_adding_heuristic() (*gmmvi.optimization.gmmvi_modules.component_adaptation.VipsComponentAdaptation method*), 41
shuffle_data() (*gmmvi.experiments.target_distributions.logistic_regression.LogisticRegression method*), 30
SteinNgEstimator (*class in gmmvi.optimization.gmmvi_modules.ng_estimator*), 53
store_rewards() (*gmmvi.models.gmm_wrapper.GmmWrapper method*), 22
StudentTMixture_LNPDF (*class in gmmvi.experiments.target_distributions.student_t_mixture*),

26

T

train_iter() (*gmmvi.optimization.gmmvi.GMMVI method*), 64
TrustRegionBasedWeightUpdater (*class in gmmvi.optimization.gmmvi_modules.weight_updater*), 58

U

update_stepsize() (*gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.ComponentStepsizeAdaptation method*), 42
update_stepsize() (*gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.DecayingComponentStepsizeAdaptation method*), 43
update_stepsize() (*gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.FixedComponentStepsizeAdaptation method*), 43
update_stepsize() (*gmmvi.optimization.gmmvi_modules.component_stepsize_adaptation.ImprovementBasedComponentStepsizeAdaptation method*), 44
update_stepsize() (*gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation.WeightStepsizeAdaptation method*), 54
update_stepsizes() (*gmmvi.models.gmm_wrapper.GmmWrapper method*), 22
update_weights() (*gmmvi.optimization.gmmvi_modules.weight_updater.WeightUpdater method*), 56
use_log_density_and_grad (*gmmvi.experiments.target_distributions.lnpdf.LNPDF property*), 25

V

VipsComponentAdaptation (*class in gmmvi.optimization.gmmvi_modules.component_adaptation*), 39
VipsSampleSelector (*class in gmmvi.optimization.gmmvi_modules.sample_selector*), 36

W

weights (*gmmvi.models.gmm.GMM property*), 18
WeightStepsizeAdaptation (*class in gmmvi.optimization.gmmvi_modules.weight_stepsize_adaptation*), 54
WeightUpdater (*class in gmmvi.optimization.gmmvi_modules.weight_updater*), 56