Documentation

June 27, 2025

Contents

1 Introduction

This document provides comprehensive documentation for the simulation functions implemented in the *github* repository. These simulations are part of a study aimed at evaluating the performance of density estimation methods using neural networks in various high-dimensional scenarios. Specifically, the simulations explore the application of neural autoregressive models and other advanced techniques to estimate simple and complex probability density functions, comparing their efficacy to traditional methods.

The detailed methodology and all the scenarios for these simulations is described in Chapter 4 of the accompanying thesis document, written in Italian.

2 Function Overview

The following subsections describe in detail each Python script, indicating the simulated dimensionality and its corresponding thesis scenario:

2.0.1 `functions.py`

This module contains essential utility functions used across all simulation scripts. It includes tools for:

- Data preprocessing (e.g., normalization, column manipulation).
- Sampling from distributions (e.g., multivariate normal).
- Statistical calculations and helper methods for density estimation.

2.0.2 `RNADE.py`

Custom implementation of the RNADE (Restricted Neural Autoregressive Density Estimator) model. This file provides:

- The RNADE model definition, supporting density estimation with mixtures of Gaussians.
- Training routines with gradient scaling and early stopping mechanisms.
- Grid search utilities for hyperparameter tuning.

2.0.3 `simulation_5d_mvtnorm.py` (Scenario A)

Implements the 5-dimensional multivariate normal scenario (Scenario A in the thesis). This script:

- Generates Gaussian-distributed data in 5 dimensions.
- Benchmarks baseline density estimation methods under a well-defined, low-complexity setting.

2.0.4 `simulation_5d_complex.py` (Scenario B)

Implements the 5-dimensional complex scenario (Scenario B). It:

- Synthesizes 5D multimodal data with challenging non-Gaussian structures.
- Evaluates neural density estimators in non-trivial environments.

2.0.5 `simulation_10d_mvtnorm.py` (Scenario A)

Handles the 10-dimensional multivariate normal scenario (Scenario A). Specifically:

- Creates Gaussian-distributed data in 10 dimensions.
- Investigates the effect of increased dimensionality on baseline estimators.

2.0.6 `simulation_10d_complex.py` (Scenario B)

Runs the 10-dimensional complex scenario (Scenario B). This script:

- Generates 10D multimodal data with complex dependencies.
- Assesses the scalability and accuracy of neural density estimation methods in higher dimensions.

2.0.7 `simulation_20d_mvtnorm.py` (Scenario A)

Executes the 20-dimensional multivariate normal scenario (Scenario A). It:

- Produces high-dimensional Gaussian data.
- Evaluates the robustness of baseline estimators at extreme scale.

2.0.8 `simulation_20d_complex.py` (Scenario B)

Carries out the 20-dimensional complex scenario (Scenario B). This script:

- Synthesizes 20D multimodal data with intricate structures.
- Challenges neural density estimators under the most complex settings.

3 Usage Instructions

To use the simulation scripts:

1. Ensure that all dependencies listed in the scripts are installed and the required environemnt is loaded.
2. Choose the global seed for the experiment. See the results for the ones used in the experiment.
3. Execute each script independently by running:

   ```
   python <simulation\\_xd\\_y>.py
   ```

4. Results and visualizations will be saved to the output directories specified within each script.

4 `functions.py`

4.1 `plot_scatter_matrix()`

Purpose: Generate and save a scatterplot matrix that visualizes pairwise relationships and marginal distributions of the input dataset.

Inputs:
- `data` (`np.ndarray`, shape (N, D)): Two-dimensional array with N samples and D features.

- `save_path` (`str`): File path (including filename) to save the resulting PNG image. Defaults to 'results/training_scatter_matrix2.png'.
- `alpha` (`float`): Opacity level for scatter plot points (between 0.0 and 1.0). Default: 0.3.
- `bins` (`int`): Number of bins for the histograms on the diagonal. Default: 30.
- `figsize` (`tuple` of two `float`): Width and height of the figure in inches. Default: (14, 14).

Outputs:
- A PNG file saved at `save_path` illustrating the scatter matrix.
- A console message confirming "Scatter matrix saved to: save_path".

Behavior:
1. Ensures the directory for `save_path` exists, creating it if necessary with `os.makedirs`.
2. Constructs a pandas DataFrame from `data`, labeling columns `x1` through `xD`.
3. Calls `pd.plotting.scatter_matrix` to generate pairwise scatter plots (off-diagonal) and histograms (diagonal).
4. Iterates over all subplot axes to hide axis tick labels for visual clarity.
5. Saves the figure using `plt.savefig` with tight bounding box and closes it with `plt.close` to free memory.

4.2 `fit_multivariate_normal()`

Purpose: Compute the empirical mean vector and a regularized covariance matrix for a multivariate normal distribution estimated from sample data.

Inputs:
- `train_data` (`np.ndarray`, shape (N, D)): An array of N samples in D dimensions, where each row represents a data point.
- `reg_epsilon` (`float`, optional): A small non-negative value added to the diagonal of the covariance matrix to ensure positive-definiteness. Default: 1e-5.

Outputs:
- `mean` (`np.ndarray`, shape (D,)): The estimated mean vector, computed as the column-wise average of `train_data`.

- `cov` (`np.ndarray`, shape (D, D)): The regularized covariance matrix, computed as the sample covariance plus `reg_epsilon` times the identity matrix.

Behavior:
1. Calculates the sample mean:
2. Computes the sample covariance matrix using `np.cov` with `rowvar=False`, yielding a (D, D) matrix.
3. Adds `reg_epsilon * I` (identity matrix of size D) to the covariance to avoid singularity and improve numerical stability.
4. Returns the pair (`mean`, `cov`).

4.3 `fit_gmm_train_val(train_data, val_data, n_components_list=None, covariance_type='full', random_state=0)`

Purpose: Select optimal GMM component count via a single train/validation split.

Inputs:
- `train_data`, `val_data` (`np.ndarray`): Training and validation samples (reshaped internally if 1D).
- `n_components_list` (list of int, optional): Candidate component counts. Defaults to 1–29.
- `covariance_type` (str): Covariance structure for all GMMs.
- `random_state` (int): Seed for reproducibility.

Outputs:
- `best_gmm` (GaussianMixture): Model fitted on full training set with optimal components.
- `best_n` (int): Selected number of components.

Behavior:
1. Iterate over `n_components_list`, fit GMM on `train_data`, compute log-likelihood on `val_data`.
2. Choose `best_n` maximizing validation score.
3. Refit best GMM on entire `train_data` and return it.

4.4 `fit_kde_bandwidth_train_val()`

Purpose: Tune a Gaussian KDE bandwidth factor using train/validation split and SciPy's `gaussian_kde`.

Inputs:
- `train_data`, `val_data` (`np.ndarray`): Samples for train/validation.

- `multipliers` (`np.ndarray`, optional): Factors for pilot bandwidth. Defaults to logspace(-1,1,100).
- `kernel` (str): Only 'gaussian' supported.
- `random_state` (int): Not used directly but for reproducibility.

Outputs:
- `best_kde` (`gaussian_kde`): KDE fitted on train_data with optimal bandwidth.
- `best_factor` (float): Multiplicative factor on pilot bandwidth.

Behavior:
1. Compute pilot bandwidth via Scott's rule.
2. Generate candidate factors, evaluate average log-likelihood on val_data.
3. Select best factor, refit KDE on train_data, and return.

4.5 `train_realnvp()`

Purpose: Train a RealNVP normalizing flow model by minimizing the negative log-likelihood (NLL) using the Adam optimizer. Optionally applies early stopping based on validation performance.

Inputs:
- `model` (`RealNVP`): Uninitialized RealNVP model instance, as built in here
- `train_loader` (`DataLoader`): DataLoader providing training batches.
- `val_loader` (`DataLoader` or `None`): Validation DataLoader for early stopping. If `None`, early stopping is disabled. Default: `None`.
- `lr` (float): Learning rate for the Adam optimizer. Default: 0.0125.
- `weight_decay` (float): L2 regularization coefficient. Default: 0.
- `epochs` (int): Maximum number of epochs to train. Default: 20.
- `device` (`torch.device`): Device for computation (e.g., 'cpu' or 'cuda'). Default: `cpu`.
- `patience` (int): Number of consecutive epochs with no improvement on validation NLL before stopping. Default: 5.
- `min_delta` (float): Minimum decrease in validation NLL to qualify as improvement. Default: 1e-4.

Outputs:
- Returns the trained `model` with parameters set to the best state observed on the validation set if provided; otherwise, the final state after all epochs.

Behavior:
1. Moves `model` to the specified `device`.
2. Initializes the Adam optimizer with given `lr` and `weight_decay`.
3. Copies initial model weights to track the best-performing state.
4. For each epoch:
   (a) **Training Phase:** Sets model to training mode, iterates over `train_loader`, computes batch NLL = , backpropagates, and updates parameters. Accumulates total NLL.
   (b) Computes average training NLL over the epoch.
   (c) **Validation Phase (if `val_loader` supplied):** Sets model to evaluation mode, disables gradient computation, iterates over `val_loader`, computes validation NLL, and averages.
   (d) Checks if validation NLL improved by at least `min_delta`. If improved, updates best weights and resets no-improvement counter; otherwise, increments it.
   (e) Prints epoch summary: training NLL and, if available, validation NLL.
   (f) If no-improvement counter equals `patience`, breaks training early.
5. After training, if validation was used, restores model weights to the best recorded state.

4.6 `plot_density_comparisons()`

Purpose: Plot sorted estimated vs true densities for multiple methods.

Inputs:
- `estimates` (list of (str, np.ndarray)): Named density arrays.
- `true_density` (np.ndarray): Ground-truth density.
- `dim` (int): Dimensionality tag for filenames.
- `out_dir` (str): Directory to save plots.
- `dataset` (str or None): Optional dataset identifier.

Outputs: Saves one plot per estimate named `density_comparison_name_dimdim_datasetdatase`

Behavior:
1. Sort sample indices by true_density ascending.
2. For each estimate, plot estimated and true densities on same axes.
3. Save and close each figure.

4.7 `plot_density_comparisons_filter()`

Purpose: Same as `plot_density_comparisons` but only for points with `true_density < 0.1`.

Behavior: Filter indices then proceed as in _

density_comparisonsfn:plot_density_comparisons

.

4.8 Estimator Wrappers

`rnade_est(test_data, model)`: Returns density = exp(model(log_prob)) on `test_data`.

`realnvp_est(test_data, model)`: Returns exp(model.log_prob(test_data)).

`gmm_est(test_data, gmm_model)`: Uses `gmm_model.score_samples` then exp.

`kde_est(test_data, kde_model)`: Uses `kde_model.logpdf` then exp.

4.9 Error Metrics

`evaluate_rmse(est_density, true_density)`: RMSE $= \sqrt{\text{mean}((\hat{p} - p)^2)}$.

`evaluate_rmse_median(est_density, true_density)`: Median-based RMSE.

`evaluate_ise(est_density, true_density)`: ISE $= \text{mean}(((\hat{p} - p)^2)/p)$.

`evaluate_ise_median(est_density, true_density)`: Median-based ISE.

`evaluate_rel_L1(est_density, true_density)`: Mean relative L1 $= \text{mean}(|\hat{p} - p|/p)$.

`evaluate_rel_L1_median(est_density, true_density)`: Median relative L1.

5 RNADE Functions Documentation

5.1 `RNADE`

Purpose: Implements the RNADE model for density estimation using a mixture of Gaussians.

Inputs:
- `input_dim` (int): Number of input dimensions .

- `hidden_units` (int): Number of hidden layer units .
- `num_components` (int): Number of mixture components per conditional distribution.

Outputs: An initialized RNADE model instance.

Behavior:

1. Initializes shared parameters (e.g., weights and biases) and dimension-specific parameters (e.g., means, variances).
2. Supports autoregressive density estimation by processing input features sequentially.

5.2 `forward(x)`

Purpose: Computes the log-likelihood of the input batch `x` under the RNADE model.

Inputs:

- `x` (`torch.Tensor`, shape): Batch of input samples.

Outputs:

- `log_prob` (`torch.Tensor`, shape): Log-likelihoods for each sample in the batch.

Behavior:

1. Iteratively computes conditional probabilities for each dimension using an autoregressive order.
2. Computes mixing coefficients, means, and standard deviations for the Gaussian mixture of each conditional.
3. Aggregates log-probabilities across dimensions to compute the total log-likelihood.

5.3 `train_rnade`

Purpose: Trains the RNADE model using stochastic gradient descent on the negative log-likelihood.

Inputs:

- `model` (`RNADE`): Initialized RNADE model.
- `train_data`, `valid_data` (`np.ndarray`, shape): Training and validation datasets.
- `num_epochs` (int, optional): Maximum number of training epochs. Default: 500.
- `batch_size` (int, optional): Minibatch size. Default: 100.
- `init_lr` (float, optional): Initial learning rate. Default: 0.1.

- `weight_decay` (float, optional): Regularization coefficient for shared weights. Default: 0.0.
- `patience` (int, optional): Early stopping patience in epochs. Default: 30.

Outputs:
- `model` (`RNADE`): Trained model with the best validation performance.

Behavior:
1. Converts input datasets to PyTorch tensors if not already.
2. Trains using minibatch stochastic gradient descent.
3. Scales gradients of mean parameters based on variance, as suggested in Uria et al. (2013).
4. Monitors validation log-likelihood for early stopping.
5. Updates learning rate linearly over epochs.

5.4 `train_rnade_finale`

Purpose: Trains RNADE with early stopping when training loss exceeds the best cross-validation loss.

Inputs: Same as `train_rnade`, with an additional parameter:

- `best_cv_val_ll` (float): Best cross-validation log-likelihood.

Outputs: Trained RNADE model.

Behavior: Stops training when training loss surpasses the best validation loss.

5.5 `grid_search`

Purpose: Performs grid search to find optimal hyperparameters for RNADE.

Inputs:
- `train_data`, `valid_data` (`np.ndarray`): Training and validation datasets.
- `hidden_units` (int): Number of hidden layer units.
- `hyperparams_grid` (dict): Dictionary of hyperparameter lists:
  - `"init_lr"`: Learning rates.
  - `"weight_decay"`: Regularization coefficients.
  - `"num_components"`: Mixture component counts.

Outputs:

- `best_params` (dict): Optimal hyperparameters.
- `best_val_ll` (float): Best validation log-likelihood.

Behavior:

1. Iterates over all hyperparameter combinations.
2. Trains RNADE for each combination.
3. Evaluates validation performance and tracks the best configuration.

5.6 `load_dataset`

Purpose: Loads a CSV dataset into a pandas DataFrame.

Inputs:

- `file_name` (str): Path to the CSV file.
- `sep` (str, optional): Column separator. Default: ','.

Outputs:

- `dataset` (`pandas.DataFrame`): Loaded dataset.

5.7 `preprocess_data`

Purpose: Removes specified columns and one variable from each highly correlated pair.

Inputs:

- `df` (`pandas.DataFrame`): Input dataset.
- `columns_to_remove` (list): List of columns to drop.

Outputs:

- `df` (`pandas.DataFrame`): Processed dataset.

Behavior: Drops columns with correlation ¿ 0.98 and user-specified columns.

5.8 `split_data`

Purpose: Splits data into training and testing sets.

Inputs:

- `data` (`np.ndarray`, shape): Input dataset.
- `test_size` (float, optional): Proportion of data for testing. Default: 0.1.
- `random_state` (int, optional): Seed for reproducibility. Default: 123.

Outputs:

- Training and testing datasets.

5.9 `normalize_data`

Purpose: Normalizes datasets using mean and standard deviation of the training set.

Inputs: Training and testing datasets.

Outputs: Normalized datasets.