torchda

acse-jm122

Oct 07, 2024

CONTENTS

TORCHDA

Important Notes:

- Background state *x0* or *xb* must be a 1D or 2D tensor with shape ([*batch size*], state dimension), and *batch size* is optional. *batch size* is only available for 3D Variational (3D-Var) algorithm.

- Observations or measurements *y* must be a 2D tensor.

  - The shape of *y* must be (number of observations, state dimension) for Kalman Filter (KF), Ensemble Kalman Filter (EnKF), and 4D Variational (4D-Var) algorithm. The number of observations must be at least 1 in KF or EnKF, and this number must be at least 2 in 4D-Var.

  - The shape of *y* must be ([*batch size*], state dimension) for 3D-Var algorithm, and *batch size* is optional for `batch size == 1`.

- A callable code object *M* must be able to handle the 1D tensor input *x* with shape (state dimension,). The output of the *M* must be a 2D tensor with shape (time window sequence length, state dimension).

- *H* could be either a tensor or a callable code object, and tensor *H* is only available in KF algorithm.

  - If *H* is a callable code object, it must be able to handle the input *x* in 2D tensor with shape ([*number of ensemble*], state dimension), and default `number of ensemble == 1` for all algorithms except EnKF. The output of this callable code object *H* must be a corresponding shape ([*number of ensemble*], measurement dimension), and the output must be a 2D tensor with shape (1, state dimension) in all other algorithms.

  - If *H* is a tensor, it must be a 2D tensor of shape (measurement dimension, state dimension). This matrix maps the state space to the measurement space.

  - Notice: A callable *H* code object would be approximated by the Jacobian method in KF algorithm. That might cause unexpected error due to incorrect shaping of tensor. Recommended input tensor *x* for the callable *H* is a 1D tensor (state dimension,) in KF algorithm.

1.1 TorchDA

Use Deep Learning in Data Assimilation

TorchDA is a Python package that provides a flexible and user-friendly framework for performing data assimilation with neural networks on various algorithms, including Ensemble Kalman Filter (EnKF), 3D Variational (3D-Var) assimilation, and 4D Variational (4D-Var) assimilation.

This package is designed to simplify the process of configuring and executing data assimilation cases, making it easier for researchers and practitioners to apply data assimilation techniques with neural networks to their scientific and engineering problems.

1.1.1 Modules

- **parameters:**
    Contains the Parameters class for specifying data assimilation parameters.

- **builder:**
    Provides a CaseBuilder class for configuring and executing data assimilation cases.

- **executor:**
    Implements the _Executor class for executing data assimilation cases.

- **kalman_filter:**
    Implements EnKF and Kalman Filter algorithms for data assimilation.

- **variational:**
    Implements 3D-Var and 4D-Var algorithms for variational data assimilation.

For more information, please refer to the package documentation.

class torchda.**CaseBuilder**(*case_name: str | None = None*, *parameters: dict[str, Any] |* Parameters *| None = None*)

A builder class for configuring and executing data assimilation cases.

This class provides a convenient way for users to set up and execute data assimilation cases using various algorithms, including Ensemble Kalman Filter (EnKF), 3D-Var, and 4D-Var.

> **Parameters**
>
> - **case_name** (`str`, `optional`) – A name for the data assimilation case. Default is 'case_{current_timestamp}'.
> - **parameters** (`dict[str, Any]` | `Parameters`, `optional`) – A dictionary or an instance of Parameters class containing configuration parameters for the data assimilation case.

case_name

A name for the data assimilation case.

> **Type**
>
> str

set_parameters(*parameters: dict[str, Any] |* Parameters)

-> CaseBuilder: Set a batch of parameters for the data assimilation case.

set_parameter(*self*, *name: str*, *value: Any*) $\rightarrow$ CaseBuilder:

Set a parameter for the data assimilation case.

set_algorithm(*algorithm: Algorithms*) $\rightarrow$ CaseBuilder:

Set the data assimilation algorithm to use (EnKF, 3D-Var, 4D-Var).

set_device(*device: Device*) $\rightarrow$ CaseBuilder:

Set the device (CPU or GPU) for computations.

set_forward_model(forward_model: Callable[[torch.Tensor, _GenericTensor],

torch.Tensor] | Callable[…, torch.Tensor]) -> CaseBuilder: Set the state transition function 'M' for EnKF.

set_output_sequence_length(*output_sequence_length: int*) $\rightarrow$ "CaseBuilder":

Set the output sequence length for the forward model.

set_observation_model(observation_model: torch.Tensor |

Callable[[torch.Tensor], torch.Tensor]) -> CaseBuilder: Set the observation model or matrix 'H' for EnKF.

set_background_covariance_matrix()

(background_covariance_matrix: torch.Tensor) -> CaseBuilder: Set the background covariance matrix 'B'.

set_observation_covariance_matrix()

(observation_covariance_matrix: torch.Tensor) -> CaseBuilder: Set the observation covariance matrix 'R'.

set_background_state(*background_state: torch.Tensor*) → CaseBuilder:

Set the initial background state estimate 'xb'.

set_observations(*observations: torch.Tensor*) → CaseBuilder:

Set the observed measurements.

set_observation_time_steps(*observation_time_steps: _GenericTensor*)

-> CaseBuilder: Set the observation time steps.

set_gaps(*gaps: _GenericTensor*) → CaseBuilder:

Set the number sequence of time steps between observations.

set_num_ensembles(*num_ensembles: int*) → CaseBuilder:

Set the number of ensembles for EnKF.

set_start_time(*start_time: int | float*) → CaseBuilder:

Set the starting time of the data assimilation process.

set_args(*args: tuple*) → CaseBuilder:

Set additional arguments for state transition function.

set_optimizer_cls(*optimizer_cls: Type[torch.optim.Optimizer]*)

-> CaseBuilder: Set the optimizer class for optimization-based algorithms.

set_optimizer_args(*optimizer_args: dict[str, Any]*)

-> CaseBuilder: Set the optimizer class arguments for optimization-based algorithms.

set_max_iterations(*max_iterations: int*) → CaseBuilder:

Set the maximum number of iterations for optimization-based algorithms.

set_early_stop(*early_stop: tuple[int, int | float] | None*) → CaseBuilder:

Set the early stopping criterion (early_stop_iterations, absolute_tolerance) for optimization-based algorithms.

set_record_log(*record_log: bool*) → CaseBuilder:

Set whether to record and print log messages during execution.

execute() → dict[str, torch.Tensor | dict[str, list]]:

Execute the data assimilation case and return the results.

get_results_dict() → dict[str, torch.Tensor | dict[str, list]]:

Get the dictionary containing the results of the executed case.

get_parameters_dict() → dict[str, Any]:

Get the dictionary of configured parameters for the case.

static check_covariance_matrix(*cov_matrix: Tensor*) → None

Check if a given covariance matrix is valid.

> **Parameters**
> **cov_matrix** (`torch.Tensor`) – The covariance matrix to be checked.

Raises

    LinAlgError – If the covariance matrix is not a valid square matrix, not symmetric, or is singular.

get_result(*name: str*) → Tensor | dict[str, list]

    Get a specific result from the executed case by name.

    Parameters

        name (`str`) – The name of the result to retrieve.

- **'average_ensemble_all_states'**

  Only available when algorithm is `Algorithms.EnKF`.

- **'each_ensemble_all_states'**

  Only available when algorithm is `Algorithms.EnKF`.

- **'assimilated_state'**

  Only available when algorithm is `Algorithms.Var3D` or `Algorithms.Var4D`.

- **'intermediate_results'**

  Only available when algorithm is `Algorithms.Var3D` or `Algorithms.Var4D`.

    Returns

        A requested result.

    Return type

        torch.Tensor | dict[str, list]

class torchda.**Parameters**(*algorithm: ~torchda.Algorithms | None = None, device: ~torchda.Device = Device.CPU, observation_model: ~torch.Tensor | ~typing.Callable[[~torch.Tensor], ~torch.Tensor] | None = None, background_covariance_matrix: ~torch.Tensor | None = None, observation_covariance_matrix: ~torch.Tensor | None = None, background_state: ~torch.Tensor | None = None, observations: ~torch.Tensor | None = None, forward_model: ~typing.Callable[[~torch.Tensor, ~torchda._GenericTensor], ~torch.Tensor] | ~typing.Callable[[...], ~torch.Tensor] = <function Parameters.<lambda>>, output_sequence_length: int = 1, observation_time_steps: ~torchda._GenericTensor = (), gaps: ~torchda._GenericTensor = (), num_ensembles: int = 0, start_time: int | float = 0.0, optimizer_cls: ~typing.Type[~torch.optim.optimizer.Optimizer] = <class 'torch.optim.adamw.AdamW'>, optimizer_args: dict[str, typing.Any] = <factory>, max_iterations: int = 1000, early_stop: tuple[int, int | float] | None = None, record_log: bool = True, args: tuple = ()*)

Data class to hold parameters for data assimilation.

This class encapsulates the parameters required for data assimilation algorithms, such as Ensemble Kalman Filter (EnKF), 3D-Var, and 4D-Var.

algorithm

    The data assimilation algorithm to use (EnKF, 3D-Var, 4D-Var).

    Type

        Algorithms

device

    The device (CPU or GPU) to perform computations on. Default is CPU.

    Type

        Device, optional

observation_model

The observation model or matrix 'H' that relates the state space to the observation space. It can be a pre-defined tensor or a Callable function that computes observations from the state.

> **Type**
>
> torch.Tensor | Callable[[torch.Tensor], torch.Tensor]

background_covariance_matrix

The background covariance matrix 'B' representing the uncertainty of the background state estimate.

> **Type**
>
> torch.Tensor

observation_covariance_matrix

The observation covariance matrix 'R' representing the uncertainty in the measurements.

> **Type**
>
> torch.Tensor

background_state

The initial background state estimate 'xb'.

> **Type**
>
> torch.Tensor

observations

The observed measurements corresponding to the given observation times.

> **Type**
>
> torch.Tensor

forward_model

Callable[. . . , torch.Tensor], optional The state transition function 'M' that predicts the state of the system given the previous state and the time range. Required for EnKF and 4D-Var.

> **Type**
>
> Callable[[torch.Tensor, _GenericTensor], torch.Tensor] |

output_sequence_length

The number of output states along the time for the forward model. Used to determine the length of the output sequence for the forward model.

> **Type**
>
> int, optional

observation_time_steps

A 1D array containing the observation times in increasing order.

> **Type**
>
> _GenericTensor, optional

gaps

A 1D array containing the number of time steps between consecutive observations.

> **Type**
>
> _GenericTensor, optional

num_ensembles

The number of ensembles used in the Ensemble Kalman Filter (EnKF) algorithm.

> **Type**
>> int, optional

start_time

> The starting time of the data assimilation process. Default is 0.0.
>
>> **Type**
>>> int | float, optional

optimizer_cls

> The selected optimizer class for optimization-based algorithms (3D-Var, 4D-Var).
>
>> **Type**
>>> Type[torch.optim.Optimizer], optional

optimizer_args

> The hyperparameters in the selected optimizer class for optimization-based algorithms (3D-Var, 4D-Var). Default is {'lr': 1e-3}.
>
>> **Type**
>>> dict[str, Any], optional

max_iterations

> The maximum number of iterations for optimization-based algorithms (3D-Var, 4D-Var).
>
>> **Type**
>>> int, optional

early_stop

> The early stopping criterion (early_stop_iterations, absolute_tolerance) applied on optimization-based algorithms (3D-Var, 4D-Var).
>
>> **Type**
>>> tuple[int, int | float] | None, optional

record_log

> Whether to record and print logs for iteration progress. Default is True.
>
>> **Type**
>>> bool, optional

args

> Additional arguments to pass to state transition function.
>
>> **Type**
>>> tuple, optional

Notes

- Ensure that the provided tensors are properly shaped and compatible with the algorithm's requirements.

- For EnKF, 'forward_model' should be provided, and 'observation_time_steps' should have at least 1 time point.

- For 3D-Var and 4D-Var, 'optimizer_cls', 'optimizer_args', and 'max_iterations' control the optimization process.

- For 4D-Var, 'observation_time_steps' should have at least 2 time points.

class torchda.**_Executor**(*parameters:* Parameters | *None = None*)

> Data assimilation executor class.
>
> This class provides a high-level integration for configuring and running data assimilation algorithms. It supports Ensemble Kalman Filter (EnKF), 3D-Var, and 4D-Var.

> **Warning:** There is no validation for input parameters in this class.

> **Parameters**
> > **parameters** (Parameters, *optional*) – An instance of Parameters class containing configuration parameters for the data assimilation case.

get_result(*name: str*) → Tensor | dict[str, list]

> Get a deep copy of a specific result by name from the results dictionary.
>
> **Parameters**
> > **name** (*str*) – The name of the result to retrieve.
>
> > - **'average_ensemble_all_states'**
> >   Only available when algorithm is `Algorithms.EnKF`.
> >
> > - **'each_ensemble_all_states'**
> >   Only available when algorithm is `Algorithms.EnKF`.
> >
> > - **'assimilated_state'**
> >   Only available when algorithm is `Algorithms.Var3D` or `Algorithms.Var4D`.
> >
> > - **'intermediate_results'**
> >   Only available when algorithm is `Algorithms.Var3D` or `Algorithms.Var4D`.
>
> **Returns**
> > A deep copy of the requested result.
>
> **Return type**
> > torch.Tensor | dict[str, list]

get_results_dict() → dict[str, torch.Tensor | dict[str, list]]

> Get a deep copy of the results dictionary.
>
> **Returns**
> > A deep copy of the results dictionary containing data assimilation results.
>
> > - **'average_ensemble_all_states'**
> >   Only available when algorithm is `Algorithms.EnKF`.
> >
> > - **'each_ensemble_all_states'**
> >   Only available when algorithm is `Algorithms.EnKF`.
> >
> > - **'assimilated_state'**
> >   Only available when algorithm is `Algorithms.Var3D` or `Algorithms.Var4D`.
> >
> > - **'intermediate_results'**
> >   Only available when algorithm is `Algorithms.Var3D` or `Algorithms.Var4D`.
>
> **Return type**
> > dict[str, torch.Tensor | dict[str, list]]

run() → dict[str, torch.Tensor | dict[str, list]]

> Run the selected data assimilation algorithm and return results.
>
> > **Returns**
> >
> > > A dictionary containing the results of the data assimilation algorithm.
> > >
> > > - **'average_ensemble_all_states'**
> > >   Only available when algorithm is `Algorithms.EnKF`.
> > >
> > > - **'each_ensemble_all_states'**
> > >   Only available when algorithm is `Algorithms.EnKF`.
> > >
> > > - **'assimilated_state'**
> > >   Only available when algorithm is `Algorithms.Var3D` or `Algorithms.Var4D`.
> > >
> > > - **'intermediate_results'**
> > >   Only available when algorithm is `Algorithms.Var3D` or `Algorithms.Var4D`.
> >
> > **Return type**
> >
> > > dict[str, torch.Tensor | dict[str, list]]

set_input_parameters(*parameters:* Parameters) → *_Executor*

> Set the input parameters for the data assimilation algorithm.
>
> > **Parameters**
> >
> > > **parameters** (Parameters) – An instance of the Parameters data class containing the configuration for the data assimilation algorithm.
> >
> > **Returns**
> >
> > > The _Executor instance with updated input parameters.
> >
> > **Return type**
> >
> > > *_Executor*

torchda.**apply_3DVar**(*H: Callable[[Tensor], Tensor], B: Tensor, R: Tensor, xb: Tensor, y: Tensor, optimizer_cls: Type[Optimizer] | None = None, optimizer_args: dict[str, Any] | None = None, max_iterations: int = 1000, early_stop: tuple[int, int | float] | None = None, record_log: bool = True*) → tuple[torch.Tensor, dict[str, list]]

Implementation of the 3D-Var (Three-Dimensional Variational) assimilation.

This function applies the 3D-Var assimilation algorithm to estimate the state of a dynamic system given noisy measurements. It aims to find the optimal state that minimizes the cost function combining background error and observation error.

> **Parameters**
>
> > - **H** (*Callable[[torch.Tensor], torch.Tensor]*) – The observation operator that maps the state space to the observation space. It should have the signature H(x: torch.Tensor) -> torch.Tensor.
> >
> > - **B** (*torch.Tensor*) – The background error covariance matrix. A 2D tensor of shape (state_dim, state_dim). It represents the uncertainty in the background.
> >
> > - **R** (*torch.Tensor*) – The observation error covariance matrix. A 2D tensor of shape (observation_dim, observation_dim). It models the uncertainty in the measurements.
> >
> > - **xb** (*torch.Tensor*) – The background state estimate. A 1D or 2D tensor of shape (state_dim,) or (batch_size, state_dim).
> >
> > - **y** (*torch.Tensor*) – The observed measurements. A 1D or 2D tensor of shape (observation_dim,) or (batch_size, observation_dim).

- **optimizer_cls** (*Type[torch.optim.Optimizer], optional*) – The optimizer selected in the optimization algorithm. Default is 'AdamW'.

- **optimizer_args** (*dict[str, Any], optional*) – The hyperparameters in the selected optimizer class. Default is {'lr': 1e-3}.

- **max_iterations** (*int, optional*) – The maximum number of optimization iterations. Default is 1000.

- **early_stop** (*tuple[int, int | float] | None, optional*) – The early stopping criterion (early_stop_iterations, absolute_tolerance). Default is None.

- **record_log** (*bool, optional*) – Whether to record and print logs for iteration progress. Default is True.

Returns

- **x_optimal** (*torch.Tensor*) – The optimal state estimate obtained using the 3D-Var assimilation.

- **intermediate_results** (*dict[str, list]*) – A dictionary containing intermediate results during optimization.

  – **'J'**
      List of cost function values at each iteration.

  – **'J_grad_norm'**
      List of norms of the cost function gradients at each iteration.

  – **'background_states'**
      List of background state estimates at each iteration.

Raises
    TypeError – If 'H' is not a Callable.

Notes

- The function assumes that the input tensors are properly shaped and valid for the 3D-Var assimilation. Ensure that 'xb', 'B', 'R', and 'y' are appropriate for the dimensions of 'H'.

- The 3D-Var algorithm seeks an optimal state estimate by minimizing a cost function that incorporates both background and observation errors.

torchda.**apply_4DVar**(*time_obs: _GenericTensor*, *gaps: _GenericTensor*, *M: Callable[[Tensor, _GenericTensor], Tensor] | Callable[[...], Tensor]*, *H: Callable[[Tensor], Tensor]*, *B: Tensor*, *R: Tensor*, *xb: Tensor*, *y: Tensor*, **args*, *optimizer_cls: Type[Optimizer] | None = None*, *optimizer_args: dict[str, Any] | None = None*, *max_iterations: int = 1000*, *early_stop: tuple[int, int | float] | None = None*, *record_log: bool = True*) → tuple[torch.Tensor, dict[str, list]]

Implementation of the 4D-Var (Four-Dimensional Variational) assimilation.

This function applies the 4D-Var assimilation algorithm to estimate the state of a dynamic system given noisy measurements. It aims to find the optimal state that minimizes the cost function combining background error and observation error over a specified time window.

Parameters

- **time_obs** (*_GenericTensor*) – A 1D array containing the observation times in increasing order.

- **gaps** (*_GenericTensor*) – A 1D array containing the number of time steps between consecutive observations.

- **M** (*Callable[[torch.Tensor, _GenericTensor], torch.Tensor] |*) – Callable[..., torch.Tensor] The state transition function (process model) that predicts the state of the system given the previous state and the time range. It should have the signature M(x: torch.Tensor, time_range: torch.Tensor, *args) -> torch.Tensor. 'x' is the state vector, 'time_range' is a 1D tensor of time steps to predict the state forward, and '*args' represents any additional arguments required by the state transition function.

- **H** (*Callable[[torch.Tensor], torch.Tensor]*) – The observation operator that maps the state space to the observation space. It should have the signature H(x: torch.Tensor) -> torch.Tensor.

- **B** (*torch.Tensor*) – The background error covariance matrix. A 2D tensor of shape (state_dim, state_dim). It represents the uncertainty in the background.

- **R** (*torch.Tensor*) – The observation error covariance matrix. A 2D tensor of shape (observation_dim, observation_dim). It models the uncertainty in the measurements.

- **xb** (*torch.Tensor*) – The background state estimate. A 1D tensor of shape (state_dim).

- **y** (*torch.Tensor*) – The observed measurements. A 2D tensor of shape (number of observations, measurement_dim). Each row represents a measurement at a specific time step.

- **args** (*tuple, optional*) – Additional arguments to pass to the state transition function 'M'.

- **optimizer_cls** (*Type[torch.optim.Optimizer], optional*) – The optimizer class selected in the optimization algorithm. Default is 'AdamW'.

- **optimizer_args** (*dict[str, Any], optional*) – The hyperparameters in the selected optimizer class. Default is {'lr': 1e-3}.

- **max_iterations** (*int, optional*) – The maximum number of optimization iterations. Default is 1000.

- **early_stop** (*tuple[int, int | float] | None, optional*) – The early stopping criterion (early_stop_iterations, absolute_tolerance). Default is None.

- **record_log** (*bool, optional*) – Whether to record and print logs for iteration progress. Default is True.

Returns

- **x_optimal** (*torch.Tensor*) – The optimal state estimate obtained using the 4D-Var assimilation.

- **intermediate_results** (*dict[str, list]*) – A dictionary containing intermediate results during optimization.

  – **'Jb'**
    List of background cost function values at each iteration.

  – **'Jo'**
    List of observation cost function values at each iteration.

  – **'J'**
    List of cost function values at each iteration.

  – **'J_grad_norm'**
    List of norms of the cost function gradients at each iteration.

  – **'background_states'**
    List of background state estimates at each iteration.

Raises
    TypeError – If 'M' or 'H' are not Callable, or if 'y' is not a tuple or list.

Notes

- The function assumes that the input tensors are properly shaped and valid for the 4D-Var assimilation. Ensure that 'xb', 'B', 'R', and 'y' are appropriate for the dimensions of 'M', 'H', and the observation times.

- The 4D-Var algorithm seeks an optimal state estimate over a time window by minimizing a cost function that incorporates both background and observation errors.

torchda.**apply_EnKF**(*time_obs: _GenericTensor*, *gaps: _GenericTensor*, *Ne: int*, *M: Callable[[Tensor, _GenericTensor], Tensor] | Callable[[...], Tensor]*, *H: Tensor | Callable[[Tensor], Tensor]*, *P0: Tensor*, *R: Tensor*, *x0: Tensor*, *y: Tensor*, **args*, *start_time: float = 0.0*) → tuple[torch.Tensor, torch.Tensor]

Implementation of the Ensemble Kalman Filter See e.g. Evensen, Ocean Dynamics (2003), Eqs. 44–54

This function applies the Ensemble Kalman Filter algorithm to estimate the state of a dynamic system given noisy measurements. It uses an ensemble of state estimates to represent the uncertainty in the estimated state. It is executed within a no-grad context, meaning that gradient computation is disabled.

Parameters

- **time_obs** (*_GenericTensor*) – A 1D array containing the observation times in increasing order.

- **gaps** (*_GenericTensor*) – A 1D array containing the number of time steps between consecutive observations.

- **Ne** (*int*) – The number of ensemble members representing the state estimates.

- **M** (*Callable[[torch.Tensor, _GenericTensor], torch.Tensor] |*) – Callable[..., torch.Tensor] The state transition function (process model) that predicts the state of the system given the previous state and the time range. It should have the signature M(x: torch.Tensor, time_range: torch.Tensor, *args) -> torch.Tensor. 'x' is the state vector, 'time_range' is a 1D tensor of time steps to predict the state forward, and '*args' represents any additional arguments required by the state transition function.

- **H** (*torch.Tensor | Callable[[torch.Tensor], torch.Tensor]*) – The measurement matrix or a function that computes the measurement matrix. If 'H' is a torch.Tensor, it is a 2D tensor of shape (measurement_dim, state_dim), where 'measurement_dim' is the dimension of measurement and 'state_dim' is the dimension of the state vector. This matrix maps the state space to the measurement space. If 'H' is a Callable, it should have the signature H(x: torch.Tensor) -> torch.Tensor to compute the measurement, and 'H' must be able to handle the input 'x' with shape (number of ensemble, state_dim). The output of Callable 'H' must be a Tensor with shape (number of ensemble, measurement_dim).

- **P0** (*torch.Tensor*) – The initial covariance matrix of the state estimate. A 2D tensor of shape (state_dim, state_dim). It represents the uncertainty of the initial state estimate.

- **R** (*torch.Tensor*) – The measurement noise covariance matrix. A 2D tensor of shape (measurement_dim, measurement_dim). It models the uncertainty in the measurements.

- **x0** (*torch.Tensor*) – The initial state estimate. A 1D tensor of shape (state_dim).

- **y** (*torch.Tensor*) – The observed measurements. A 2D tensor of shape (number of observations, measurement_dim). Each row represents a measurement at a specific time step.

- **args** (`tuple, optional`) – Additional arguments to pass to the state transition function 'M'.

- **start_time** (`float, optional`) – The starting time of the filtering process. Default is 0.0.

Returns

- **x_ave** (*torch.Tensor*) – A 2D tensor of shape (num_steps + 1, state_dim). Each row represents the ensemble mean state vector at a specific time step, including the initial state.

- **x_ens** (*torch.Tensor*) – A 3D tensor of shape (num_steps + 1, Ne, state_dim). Each slice along the second dimension represents an ensemble member's state estimates at a specific time step, including the initial state.

Raises
    TypeError – If 'M' is not a Callable or 'H' is not a torch.Tensor or Callable.

Notes

- The function assumes that the input tensors are properly shaped and valid for the Ensemble Kalman Filter. Ensure that 'x0', 'P0', 'R', and 'y' are appropriate for the dimensions of 'M' and 'H'.

- The function assumes that 'time_obs' contains time points that are increasing, and 'gaps' specifies each number of time steps between consecutive observations.

- The implementation uses an ensemble of state estimates to represent the uncertainty in the estimated state. The ensemble Kalman filter provides an approximation to the true state distribution.

torchda.**apply_KF**(*time_obs: _GenericTensor*, *gaps: _GenericTensor*, *M: Callable[[Tensor, _GenericTensor], Tensor] | Callable[[...], Tensor]*, *H: Tensor | Callable[[Tensor], Tensor]*, *P0: Tensor*, *R: Tensor*, *x0: Tensor*, *y: Tensor*, **args*, *start_time: float = 0.0*) → Tensor

Implementation of the Kalman Filter (constant P assumption).

This function applies the Kalman Filter algorithm to estimate the state of a dynamic system given noisy measurements. It is executed within a no-grad context, meaning that gradient computation is disabled.

Parameters

- **time_obs** (`_GenericTensor`) – A 1D array containing the observation times in increasing order.

- **gaps** (`_GenericTensor`) – A 1D array containing the number of time steps between consecutive observations.

- **M** (`Callable[[torch.Tensor, _GenericTensor], torch.Tensor] |` Callable[…, torch.Tensor]) The state transition function (process model) that predicts the state of the system given the previous state and the time range. It should have the signature M(x: torch.Tensor, time_range: torch.Tensor, *args) -> torch.Tensor. 'x' is the state vector, 'time_range' is a 1D tensor of time steps to predict the state forward, and '*args' represents any additional arguments required by the state transition function.

- **H** (`torch.Tensor | Callable[[torch.Tensor], torch.Tensor]`) – The measurement matrix or a function that computes the measurement matrix. If 'H' is a torch.Tensor, it is a 2D tensor of shape (measurement_dim, state_dim), where 'measurement_dim' is the dimension of measurement and 'state_dim' is the dimension of the state vector. This matrix maps the state space to the measurement space. If 'H' is a Callable, it should have the signature H(x: torch.Tensor) -> torch.Tensor to compute the measurement, and 'H' must be able to handle the input 'x' with shape (state_dim,). The output of Callable 'H' must be a Tensor

with shape (measurement_dim,). Note: Jacobian calculation on Neural Network 'H' might be incorrect for the current Kalman Filter implementation.

- **P0** (*torch.Tensor*) – The initial covariance matrix of the state estimate. A 2D tensor of shape (state_dim, state_dim). It represents the uncertainty of the initial state estimate.

- **R** (*torch.Tensor*) – The measurement noise covariance matrix. A 2D tensor of shape (measurement_dim, measurement_dim). It models the uncertainty in the measurements.

- **x0** (*torch.Tensor*) – The initial state estimate. A 1D tensor of shape (state_dim).

- **y** (*torch.Tensor*) – The observed measurements. A 2D tensor of shape (number of observations, measurement_dim). Each row represents a measurement at a specific time step.

- **args** (*tuple, optional*) – Additional arguments to pass to the state transition function 'M'.

- **start_time** (*float, optional*) – The starting time of the filtering process. Default is 0.0.

Returns

    x_estimates – Each row represents the estimated state vector at a specific time step, including the initial state.

Return type

    A 2D tensor of shape (num_steps + 1, state_dim).

Raises

    TypeError – If 'M' is not a Callable or 'H' is not a torch.Tensor or Callable.

Notes

- The function assumes that the input tensors are properly shaped and valid for the Kalman Filter. Ensure that 'x0', 'P0', 'R', and 'y' are appropriate for the dimensions of 'M' and 'H'.

- The function assumes that 'time_obs' contains time points that are increasing, and 'gaps' specifies each number of time steps between consecutive observations.

- The implementation assumes a constant P assumption, meaning the state estimate covariance matrix 'P' remains the same throughout the filtering process. If a time-varying 'P' is required, you need to modify the function accordingly.

PYTHON MODULE INDEX

t

torchda, 1