deepda

Jingyang Min

Aug 04, 2023

CONTENTS

CHAPTER

ONE

DEEPDA

1.1 DeepDA

Use Deep Learning in Data Assimilation

DeepDA 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

- **builder:**
  Provides a CaseBuilder class for configuring and 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 deepda.`**`CaseBuilder`**(*case_name: str = ''*, *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.
- **`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_all_parameters`(*parameters: dict[str, Any]* | Parameters)
: -> CaseBuilder: Set all parameters for the data assimilation case.

`set_algorithm`(*algorithm: Algorithms*) → CaseBuilder:
: Set the data assimilation algorithm to use (EnKF, 3D-Var, 4D-Var).

`set_device`(*device: Device*) → CaseBuilder:
: Set the device (CPU or GPU) for computations.

`set_forward_model`(*forward_model: Callable*) → CaseBuilder:
: Set the state transition function 'M' for EnKF.

`set_observation_model`(*observation_model: torch.Tensor* | *Callable*)
: -> 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 |`
: tuple[torch.Tensor] | list[torch.Tensor]) -> CaseBuilder: Set the observed measurements.

`set_observation_time_steps`(*observation_time_steps: _GenericTensor*)
: -> CaseBuilder: Set the observation time steps.

`set_gap`(*gap: int*) → CaseBuilder:
: Set the number of time steps between observations.

`set_num_steps`(*num_steps: int*) → CaseBuilder:
: Set the number of time steps to propagate the state forward.

`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_max_iterations`(*max_iterations: int*) → CaseBuilder:
: Set the maximum number of iterations for optimization-based algorithms.

`set_learning_rate`(*learning_rate: int* | *float*) → CaseBuilder:
: Set the learning rate for optimization-based algorithms.

`set_logging`(*logging: bool*) → CaseBuilder:
: Set whether to print log messages during execution.

`execute`() → dict[str, torch.Tensor]:
: Execute the data assimilation case and return the results.

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

Get the dictionary containing the results of the executed case.

`get_result`(*name: str*) → torch.Tensor:

Get a specific result from the executed case by name.

`get_parameters_dict()` → dict[str, Any]:

Get the dictionary of configured parameters for the case.

`class deepda.Parameters`(*algorithm: Algorithms = Algorithms.EnKF*, *observation_model: Tensor | Callable | None = None*, *background_covariance_matrix: Tensor | None = None*, *observation_covariance_matrix: Tensor | None = None*, *background_state: Tensor | None = None*, *observations: Tensor | tuple[torch.Tensor] | list[torch.Tensor] | None = None*, *device: Device | None = Device.CPU*, *forward_model: Callable | None = None*, *observation_time_steps: _GenericTensor | None = None*, *gap: int | None = 0*, *num_steps: int | None = 0*, *num_ensembles: int | None = 0*, *start_time: int | float | None = 0.0*, *max_iterations: int | None = 1000*, *learning_rate: int | float | None = 0.001*, *logging: bool | None = True*, *args: tuple | None = (None,)*)

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

`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, optional

`background_covariance_matrix`

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

Type
torch.Tensor, optional

`observation_covariance_matrix`

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

Type
torch.Tensor, optional

`background_state`

The initial background state estimate 'xb'.

Type
torch.Tensor, optional

`observations`

torch.Tensor | tuple[torch.Tensor] | list[torch.Tensor], optional The observed measurements corresponding to the given observation times. It can be a single tensor or a collection of tensors.

Type
torch.Tensor | tuple[torch.Tensor] | list[torch.Tensor]

`device`

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

Type
Device, optional

`forward_model`

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, optional

`observation_time_steps`

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

Type
_GenericTensor, optional

`gap`

The number of time steps between consecutive observations.

Type
int, optional

`num_steps`

The number of time steps to propagate the state forward. Required for EnKF and 4D-Var.

Type
int, 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

`max_iterations`

The maximum number of iterations for optimization-based algorithms (3D-Var, 4D-Var).

Type
int, optional

`learning_rate`

The learning rate for optimization-based algorithms (3D-Var, 4D-Var).

Type
int | float, optional

`logging`

Whether to print log messages during execution. 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 *num_steps* should be > 0.
- For 3D-Var and 4D-Var, *max_iterations* and *learning_rate* control the optimization process.
- For 4D-Var, *observations* should be a tuple or list of tensors, and *observation_time_steps* should have at least 2 time points.

`deepda.`**`apply_3DVar`**(*H: Callable*, *B: Tensor*, *R: Tensor*, *xb: Tensor*, *y: Tensor*, *max_iterations: int = 1000*, *learning_rate: float = 0.001*, *logging: 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`*) – 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 (sequence_length, state_dim).
- **`y`** (*`torch.Tensor`*) – The observed measurements. A 1D or 2D tensor of shape (observation_dim,) or (sequence_length, observation_dim).
- **`max_iterations`** (*`int, optional`*) – The maximum number of optimization iterations. Default is 1000.
- **`learning_rate`** (*`float, optional`*) – The learning rate for the optimization algorithm. Default is 1e-3.
- **`logging`** (*`bool, optional`*) – Whether to print 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.

`deepda.`**`apply_4DVar`**(*time_obs: _GenericTensor*, *gap: int*, *M: Callable*, *H: Callable*, *B: Tensor*, *R: Tensor*, *xb: Tensor*, *y: tuple[torch.Tensor] | list[torch.Tensor]*, *max_iterations: int = 1000*, *learning_rate: float = 0.001*, *logging: bool = True*, *args: tuple = (None,)*) → 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.
- **`gap`** (*`int`*) – The number of time steps between consecutive observations.
- **`M`** (*`Callable`*) – 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`*) – 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`** (*`tuple[torch.Tensor] | list[torch.Tensor]`*) – A tuple or list of observed measurements. Each element is a 1D tensor representing the measurements at a specific observation time.
- **`max_iterations`** (*`int, optional`*) – The maximum number of optimization iterations. Default is 1000.
- **`learning_rate`** (*`float, optional`*) – The learning rate for the optimization algorithm. Default is 1e-3.
- **`logging`** (*`bool, optional`*) – Whether to print iteration progress. Default is True.
- **`args`** (*`tuple, optional`*) – Additional arguments to pass to the state transition function 'M'. Default is (None,).

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_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.

`deepda.`**`apply_EnKF`**(*n_steps: int*, *time_obs: _GenericTensor*, *gap: int*, *Ne: int*, *M: Callable*, *H: Tensor | Callable*, *P0: Tensor*, *R: Tensor*, *x0: Tensor*, *y: Tensor*, *start_time: float = 0.0*, *args: tuple = (None,)*) → 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

- **`n_steps`** (*`int`*) – The number of time steps to propagate the state forward.
- **`time_obs`** (*`_GenericTensor`*) – A 1D array containing the observation times in increasing order.
- **`gap`** (*`int`*) – The number of time steps between consecutive observations.
- **`Ne`** (*`int`*) – The number of ensemble members representing the state estimates.
- **`M`** (*`Callable`*) – 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`*) – 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 and compute the measurement matrix, and 'H' must be able to handle the input 'x' with shape (number of ensemble, state_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.
- `start_time` (*float, optional*) – The starting time of the filtering process. Default is 0.0.
- `args` (*tuple, optional*) – Additional arguments to pass to the state transition function 'M'. Default is (None,).

Returns

- **x_ave** (*torch.Tensor*) – A 2D tensor of shape (n_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 (n_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 'gap' specifies the 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.

`deepda.apply_KF`(*n_steps: int*, *time_obs: _GenericTensor*, *gap: int*, *M: Callable*, *H: Tensor*, *P0: Tensor*, *R: Tensor*, *x0: Tensor*, *y: Tensor*, *start_time: float = 0.0*, *args: tuple = (None,)*) → 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

- `n_steps` (*int*) – The number of time steps to propagate the state forward.
- `time_obs` (*_GenericTensor*) – A 1D array containing the observation times in increasing order.
- `gap` (*int*) – The number of time steps between consecutive observations.
- `M` (*Callable*) – 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*) – The measurement matrix. 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.
- `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.
- **start_time** (*float, optional*) – The starting time of the filtering process. Default is 0.0.
- **args** (*tuple, optional*) – Additional arguments to pass to the state transition function 'M'. Default is (None,).

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 (n_steps + 1, state_dim).

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

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 'gap' specifies the 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