ads-wildfire-peshtigo

Release 2023-05-25

Team Peshtigo

May 26, 2023

CONTENTS:

CHAPTER

ONE

BACKGROUND BRIEFING

Every year, economic losses from wildfire, propelled by the global climate change and human activities, reaches nearly 50 billion dollars.

Due to the massive population growth and the rapid progress of urbanization, tackling wildfire, specifically predicting its occuring and forecasting its development both spatially and temporally, has been a critical challenge.

With the assistance of Neural Networks, we are able to accomplish the tasks handed over to us.

In this project, focusing on two kinds of Neural Networks:

- Generative Neural Networks
- Recurrent Neural Networks

we take one small step for predicting wildfire but one giant leap for mankind.

This repository contains three functionalities of our work.

All codes are written in Python from scratch.

CHAPTER

TWO

INTRODUCTION TO CORE POINTS

2.1 Generative Neural Network - Variational AutoEncoder

- Generating new data of wildfire based on fed inputs.
- On the Notebook, mean squared error between observation data and generated data is printed; and 4 images would be generated with 4 inputs for comparison. On input of an image at time point t=n, the prediction yields an image at t=n+1.

2.2 Recurrent Neural Network - Convolutional Long Short-Term Memory

- Predicting new data of wildfire based on fed inputs.
- On the Notebook, mean squared error between observation data and generated data is printed; and 3 images would be generated with 3 inputs in which each one contains 2 timesteps.

2.3 Data Assimilation

- With a different Variational AutoEncoder, assimilating data in a latent space and decompressing the data.
- A mean square error between observation data and generated data on full space before data assimilation and a mean square error between observation data and generated data on full space after data assimilation are printed.

2.4 Installation Guide

Before installing flood_tool it is worth ensuring your environment meets the dependencies below:

- Python > 3.9
- numpy >= 1.13.0
- pandas
- scikit-learn
- matplotlib
- pytest
- sphinx

CHAPTER

THREE

FILES

3.1 .ipynb notebooks

1. Objective_1.ipynb: Implementing all requirements from objective 1.
2. Objective_2.ipynb: Implementing all requirements from objective 2.
3. Objective_3.ipynb: Implementing all requirements from objective 3.

3.2 .py files

The tool is delivered via a series of .py files following:

1. tools.py: Functionalities including reading data, reshaping data, generating data for models, printing MSE scores, printing generated results and their comparism.
2. VAE.py: Functionalities including defining VAE model, calculating and visualising loss, training VAE model, testing VAE model.
3. DA.py: Functionalities including reading linear VAE model, calling both results from VAE model and ConvLSTM model, compressing data, assimilating data and decompressing data.
4. LSTM.py: Functionalities including defining LSTM model, calculating and visualising loss, training LSTM model, testing LSTM model.
6. VAE_linear.py: Functionalities including defining linear VAE model, training linear VAE model, testing linear VAE model.
7. test: Multiple test.py files for testing all the other functions in the project.

CHAPTER

FOUR

USER INSTRUCTIONS

4.1 Variational AutoEncoder

- Read data and reshape data into TensorFlow
- Import saved VAE model
- Generate result

4.2 Convolutional Long Short-Term Memory

- Read data and reshape data into TensorFlow
- Import saved ConvLSTM model
- Generate result

4.3 Data Assimilation

- Read data and reshape data into TensorFlow
- Import saved Linear VAE model
- Data compression
- Data assimilation
- Data decompression and generate result

CHAPTER

FIVE

TESTING & DOCUMENTATION

5.1 Testing

The tool includes several tests, which you can use to check its operation on your system. With pytest installed, these can be run with

```bash pytest test_LSTM.py ```

Additional more specific tests have been saved in the test directory. These can be run where required to check different parts of the tool functionality individually.

- *test_VAE.py*
- *test_LSTM.py*
- *test_DA*
- *test_tools*

5.2 Documentation

The tool *html* directory contains detailed documentation for the package, including examples and details of all functions and inputs.
```

CHAPTER

SIX

AUTHOR

Team Peshtigo(Alphabetical):

- Fan Feng
- Ligen Cai
- Luisina Canto
- Leyang Zhang
- Manya Sinha
- Wenxin Ran
- Yulin Zhuo
- Zongyang Gao

CHAPTER

SEVEN

FULL TEXT

class DA.**Autoencoder**(**args*, ***kwargs*)

Autoencoder model for image reconstruction.

Attributes:

inp_shape: The shape of the input images (height, width, channels). encoded_dim: The dimension of the encoded representation. encoder: The encoder model. decoder: The decoder model. ae_model: The full autoencoder model.

Methods:

build_encoder(): Build the encoder part of the autoencoder. build_decoder(): Build the decoder part of the autoencoder. build_ae(learning_rate=0.001, momentum=0.92): Build the full autoencoder model. train(train_x, train_y, test_x, test_y, epochs=20, batch_size=256): Train the autoencoder model. plot_loss(): Plot the training loss.

build_ae(*learning_rate=0.001*, *momentum=0.92*)

Build the full autoencoder model.

Parameters

- **learning_rate** (*float, optional*) – The learning rate for the optimizer.
- **momentum** (*float, optional*) – The momentum value for the optimizer.

Returns

The full autoencoder model.

Return type

tf.keras.Model

build_decoder()

Build the decoder part of the autoencoder.

Returns

The decoder model.

Return type

tf.keras.Model

build_encoder()

Build the encoder part of the autoencoder.

Returns

The encoder model.

Return type

tf.keras.Model

plot_loss()

Plot the training loss.

train(*train_x*, *train_y*, *test_x*, *test_y*, *epochs=20*, *batch_size=256*)

Train the autoencoder model.

Parameters

- **train_x** (*np.ndarray or tf.Tensor*) – The input training data.
- **train_y** (*np.ndarray or tf.Tensor*) – The target training data.
- **test_x** (*np.ndarray or tf.Tensor*) – The input test data.
- **test_y** (*np.ndarray or tf.Tensor*) – The target test data.
- **epochs** (*int, optional*) – The number of epochs to train for.
- **batch_size** (*int, optional*) – The batch size for training.

class DA.DataAssimilation(*enc_model*, *dec_model*, *latent_dim=64*, *R_val=0.01*)

A class for data assimilation using the Kalman filter.

Attributes:

enc_model: The encoder model used for data compression. dec_model: The decoder model used for data reconstruction. latent_dim (int): The dimension of the latent space. R_val (float): The coefficient of the observation error covariance matrix. I: The identity matrix of shape (latent_dim, latent_dim). R: The observation error covariance matrix. H: The observation operator.

Methods:

covariance_matrix: Compute the covariance matrix of a given dataset. update_prediction: Update the predicted state using the Kalman filter equations. KalmanGain: Compute the Kalman gain matrix. assimilate: Perform data assimilation using the Kalman filter.

KalmanGain(*B*, *H*, *R*)

Compute the Kalman gain matrix.

Parameters

- **B** (*np.ndarray or tf.Tensor*) – The background error covariance matrix of shape (latent_dim, latent_dim).
- **H** (*np.ndarray or tf.Tensor*) – The observation operator of shape (latent_dim, latent_dim).
- **R** (*np.ndarray or tf.Tensor*) – The observation error covariance matrix of shape (latent_dim, latent_dim).

Returns

The Kalman gain matrix of shape (latent_dim, latent_dim).

Return type

np.ndarray or tf.Tensor

assimilate(*bg_data*, *obs_data*)

Perform data assimilation using the Kalman filter

Parameters

- **bg_data** (*np.ndarray or tf.Tensor*) – The background data of shape (num_samples, height, width, channels)
- **obs_data** (*np.ndarray or tf.Tensor*) – The observed data of shape (num_samples, height, width, channels)

Returns
The updated and reconstructed data of shape (num_samples, height, width, channels)

Return type
np.ndarray or tf.Tensor

`covariance_matrix`(*X*)

Compute the covariance matrix of a given dataset.

Parameters
X (*np.ndarray or tf.Tensor*) – The input dataset of shape (num_samples, num_features).

Returns
The covariance matrix of shape (num_features, num_features).

Return type
np.ndarray or tf.Tensor

`update_prediction`(*x*, *K*, *H*, *y*)

Update the predicted state using the Kalman filter equations.

Parameters

- **x** (*np.ndarray or tf.Tensor*) – The predicted state of shape (latent_dim,).
- **K** (*np.ndarray or tf.Tensor*) – The Kalman gain matrix of shape (latent_dim, latent_dim).
- **H** (*np.ndarray or tf.Tensor*) – The observation operator of shape (latent_dim, latent_dim).
- **y** (*np.ndarray or tf.Tensor*) – The observed state of shape (latent_dim,).

Returns
The updated state of shape (latent_dim,).

Return type
np.ndarray or tf.Tensor

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from numpy.linalg import inv
import tensorflow as tf
from tensorflow.keras.optimizers import SGD

class Autoencoder(tf.keras.Model):
    """
    Autoencoder model for image reconstruction.

    Attributes:
        inp_shape: The shape of the input images (height, width, channels).
        encoded_dim: The dimension of the encoded representation.
        encoder: The encoder model.
        decoder: The decoder model.
        ae_model: The full autoencoder model.

    Methods:
```

(continues on next page)

(continued from previous page)

```
        build_encoder(): Build the encoder part of the autoencoder.
        build_decoder(): Build the decoder part of the autoencoder.
        build_ae(learning_rate=0.001, momentum=0.92): Build the full autoencoder model.
        train(train_x, train_y, test_x, test_y, epochs=20, batch_size=256): Train the␣
↪autoencoder model.
        plot_loss(): Plot the training loss.
    """

    def __init__(self, inp_shape=(256, 256, 1), encoded_dim=64):
        """
        Initialize the Autoencoder object.

        :param inp_shape: The shape of the input images (height, width, channels).
        :type inp_shape: tuple, optional
        :param encoded_dim: The dimension of the encoded representation.
        :type encoded_dim: int, optional
        """
        super(Autoencoder, self).__init__()
        self.inp_shape = inp_shape
        self.encoded_dim = encoded_dim

        self.encoder = self.build_encoder()
        self.decoder = self.build_decoder()
        self.ae_model = self.build_ae()

    def build_encoder(self):
        """
        Build the encoder part of the autoencoder.

        :return: The encoder model.
        :rtype: tf.keras.Model
        """
        inputs = tf.keras.Input(shape=self.inp_shape)
        x = tf.keras.layers.Conv2D(32, (3, 3), padding='same')(inputs)
        x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.activations.relu(x)
        x = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)
        x = tf.keras.layers.Dropout(0.25)(x)

        x = tf.keras.layers.Conv2D(16, (3, 3), padding='same')(x)
        x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.activations.relu(x)

        flatten = tf.keras.layers.Flatten()(x)
        outputs = tf.keras.layers.Dense(self.encoded_dim)(flatten)

        return tf.keras.Model(inputs, outputs)

    def build_decoder(self):
        """
        Build the decoder part of the autoencoder.
```

(continues on next page)

(continued from previous page)

```
        :return: The decoder model.
        :rtype: tf.keras.Model
        """
        inputs = tf.keras.Input(shape=(self.encoded_dim,))
        x = tf.keras.layers.Dense(128*128*16)(inputs)
        x = tf.keras.layers.Reshape(target_shape=(128,128,16))(x)
        x = tf.keras.layers.UpSampling2D((2, 2))(x)
        x = tf.keras.layers.Conv2D(16, (3, 3), padding='same')(x)
        x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.activations.relu(x)
        x = tf.keras.layers.Dropout(0.25)(x)

        outputs = tf.keras.layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same
↪')(x)

        return tf.keras.Model(inputs, outputs)

    def build_ae(self, learning_rate=0.001, momentum=0.92):
        """
        Build the full autoencoder model.

        :param learning_rate: The learning rate for the optimizer.
        :type learning_rate: float, optional
        :param momentum: The momentum value for the optimizer.
        :type momentum: float, optional
        :return: The full autoencoder model.
        :rtype: tf.keras.Model
        """
        encoder_inputs = tf.keras.Input(shape=self.inp_shape)
        encoded = self.encoder(encoder_inputs)
        decoded = self.decoder(encoded)
        ae = tf.keras.Model(encoder_inputs, decoded)
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate,␣
↪momentum=momentum)
        ae.compile(optimizer=optimizer, loss='binary_crossentropy')
        return ae

    def train(self, train_x, train_y, test_x, test_y, epochs=20, batch_size=256):
        """
        Train the autoencoder model.

        :param train_x: The input training data.
        :type train_x: np.ndarray or tf.Tensor
        :param train_y: The target training data.
        :type train_y: np.ndarray or tf.Tensor
        :param test_x: The input test data.
        :type test_x: np.ndarray or tf.Tensor
        :param test_y: The target test data.
        :type test_y: np.ndarray or tf.Tensor
        :param epochs: The number of epochs to train for.
        :type epochs: int, optional
        :param batch_size: The batch size for training.
```

(continues on next page)

(continued from previous page)

```
        :type batch_size: int, optional
        """
        early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0,␣
↪patience=3, verbose=1, mode='auto')
        his = self.ae_model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size,␣
↪verbose=2, validation_data=(test_x, test_y), callbacks=[early_stop], shuffle=True)
        self.his = his

    def plot_loss(self):
        """
        Plot the training loss.
        """
        plt.plot(self.his.history['loss'])
        plt.plot(self.his.history['val_loss'])
        plt.legend(['Loss', 'Val_loss'])
        plt.show()

class DataAssimilation:
    """
    A class for data assimilation using the Kalman filter.

    Attributes:
        enc_model: The encoder model used for data compression.
        dec_model: The decoder model used for data reconstruction.
        latent_dim (int): The dimension of the latent space.
        R_val (float): The coefficient of the observation error covariance matrix.
        I: The identity matrix of shape (latent_dim, latent_dim).
        R: The observation error covariance matrix.
        H: The observation operator.

    Methods:
        covariance_matrix: Compute the covariance matrix of a given dataset.
        update_prediction: Update the predicted state using the Kalman filter equations.
        KalmanGain: Compute the Kalman gain matrix.
        assimilate: Perform data assimilation using the Kalman filter.
    """

    def __init__(self, enc_model, dec_model, latent_dim=64, R_val=0.01):
        """
        Initialize the DataAssimilation object.

        :param enc_model: The encoder model used for data compression.
        :type enc_model: tf.keras.Model
        :param dec_model: The decoder model used for data reconstruction.
        :type dec_model: tf.keras.Model
        :param latent_dim: The dimension of the latent space.
        :type latent_dim: int, optional
        :param R_val: The coefficient of the observation error covariance matrix.
        :type R_val: float, optional
        """
        self.enc_model = enc_model
        self.dec_model = dec_model
```

(continues on next page)

(continued from previous page)

```
        self.latent_dim = latent_dim
        self.R_val = R_val
        self.I = np.identity(latent_dim)
        self.R = R_val * self.I
        self.H = self.I

    def covariance_matrix(self, X):
        """
        Compute the covariance matrix of a given dataset.

        :param X: The input dataset of shape (num_samples, num_features).
        :type X: np.ndarray or tf.Tensor

        :return: The covariance matrix of shape (num_features, num_features).
        :rtype: np.ndarray or tf.Tensor
        """
        means = np.array([np.mean(X, axis = 1)]).transpose()
        dev_matrix = X - means
        res = np.dot(dev_matrix, dev_matrix.transpose())/(X.shape[1]-1)
        return res

    def update_prediction(self, x, K, H, y):
        """
        Update the predicted state using the Kalman filter equations.

        :param x: The predicted state of shape (latent_dim,).
        :type x: np.ndarray or tf.Tensor
        :param K: The Kalman gain matrix of shape (latent_dim, latent_dim).
        :type K: np.ndarray or tf.Tensor
        :param H: The observation operator of shape (latent_dim, latent_dim).
        :type H: np.ndarray or tf.Tensor
        :param y: The observed state of shape (latent_dim,).
        :type y: np.ndarray or tf.Tensor

        :return: The updated state of shape (latent_dim,).
        :rtype: np.ndarray or tf.Tensor
        """
        res = x + np.dot(K, (y - np.dot(H, x)))
        return res

    def KalmanGain(self, B, H, R):
        """
        Compute the Kalman gain matrix.

        :param B: The background error covariance matrix of shape (latent_dim, latent_
↪dim).
        :type B: np.ndarray or tf.Tensor
        :param H: The observation operator of shape (latent_dim, latent_dim).
        :type H: np.ndarray or tf.Tensor
        :param R: The observation error covariance matrix of shape (latent_dim, latent_
↪dim).
        :type R: np.ndarray or tf.Tensor
```

(continues on next page)

(continued from previous page)

```
        :return: The Kalman gain matrix of shape (latent_dim, latent_dim).
        :rtype: np.ndarray or tf.Tensor
        """
        tempInv = inv(R + np.dot(H, np.dot(B, H.transpose())))
        res = np.dot(B, np.dot(H.transpose(), tempInv))
        return res

    def assimilate(self, bg_data, obs_data):
        """
        Perform data assimilation using the Kalman filter

        :param bg_data: The background data of shape (num_samples, height, width,␣
↪channels)
        :type bg_data: np.ndarray or tf.Tensor
        :param obs_data: The observed data of shape (num_samples, height, width,␣
↪channels)
        :type obs_data: np.ndarray or tf.Tensor
        :return: The updated and reconstructed data of shape (num_samples, height, width,
↪ channels)
        :rtype:  np.ndarray or tf.Tensor
        """
        pix_1 = np.array(bg_data).shape[1]
        pix_2 = np.array(bg_data).shape[2]
        bg_compr = np.array(self.enc_model(bg_data.reshape(-1, pix_1, pix_2, 1)))
        obs_compr = np.array(self.enc_model(obs_data.reshape(-1, pix_1, pix_2, 1)))

        B = self.covariance_matrix(bg_compr.T)

        K = self.KalmanGain(B, self.H, self.R)

        updated_data_list = []
        for i in range(len(bg_compr)):
            updated_data = self.update_prediction(bg_compr[i], K, self.H, obs_compr[i])
            updated_data_list.append(updated_data)
        updated_data_array = np.array(updated_data_list)

        updated_dec = self.dec_model(updated_data_array)

        return updated_dec
```

class VAE.**VAE**(**args*, ***kwargs*)

Variational Autoencoder (VAE) model for image generation and reconstruction.

Attributes:

latent_dim: The dimensionality of the latent space. input_dim: The dimensionality of the input data. encoder: The encoder model. decoder: The decoder model. vae: The full VAE model.

Methods:

build_encoder(): Build the encoder part of the VAE. build_decoder(): Build the decoder part of the VAE. build_vae(learning_rate=0.0001, reconstruction_weight=0.5, kl_weight=0.5): Build the full VAE model. sampling(args): Reparameterization trick to sample from the latent space. loss_function(inputs, outputs,

z_mean, z_log_var, reconstruction_weight=0.5, kl_weight=0.5): Compute the loss function for the VAE. train(x_train, y_train, batch_size, epochs, x_test, y_test): Train the VAE model on the provided data. plot_loss(): Plot the training loss.

`build_decoder()`

Build the decoder part of the autoencoder.

Returns
The decoder model.

Return type
tf.keras.Model

`build_encoder()`

Build the encoder part of the autoencoder.

Returns
The encoder model.

Return type
tf.keras.Model

`build_vae(learning_rate=0.0001, reconstruction_weight=0.5, kl_weight=0.5)`

Build the full autoencoder model.

Parameters

- **learning_rate** (*float, optional*) – The learning rate for the optimizer.
- **reconstruction_weight** (*float, optional*) – The weight for the reconstruction loss term in the VAE loss function.
- **kl_weight** (*float, optional*) – The weight for the Kullback-Leibler (KL) divergence loss term in the VAE loss function.

Returns
The full autoencoder model.

Return type
tf.keras.Model

`loss_function(inputs, outputs, z_mean, z_log_var, reconstruction_weight=0.5, kl_weight=0.5)`

Compute the loss function for the Variational Autoencoder (VAE).

Parameters

- **inputs** (*tf.Tensor*) – The input tensor representing the original data.
- **outputs** (*tf.Tensor*) – The output tensor representing the reconstructed data.
- **z_mean** (*tf.Tensor*) – The tensor representing the mean of the latent space.
- **z_log_var** (*tf.Tensor*) – The tensor representing the log variance of the latent space.
- **reconstruction_weight** (*float, optional*) – The weight for the reconstruction loss term in the VAE loss function.
- **kl_weight** (*float, optional*) – The weight for the Kullback-Leibler (KL) divergence loss term in the VAE loss function.

Returns
The computed loss value.

Return type
tf.Tensor

plot_loss()

Plot the training loss.

sampling(*args*)

Reparameterization trick to sample from the latent space.

Parameters

args (*tuple*) – Tuple containing the mean and log variance tensors of the latent space.

Returns

The sampled data from the latent space.

Return type

tf.Tensor

train(*x_train*, *y_train*, *batch_size*, *epochs*, *x_test*, *y_test*)

Train the VAE model on the provided training data.

Parameters

- **x_train** (*np.ndarray or tf.Tensor*) – The input training data.
- **y_train** (*np.ndarray or tf.Tensor*) – The target training data.
- **batch_size** (*int*) – The batch size for training.
- **epochs** (*int*) – The number of epochs for training.
- **x_test** (*np.ndarray or tf.Tensor*) – The input test data.
- **y_test** (*np.ndarray or tf.Tensor*) – The target test data.

```
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Definition of the Variational Autoencoder (VAE) class.
class VAE(tf.keras.Model):
    """
    Variational Autoencoder (VAE) model for image generation and reconstruction.

    Attributes:
        latent_dim: The dimensionality of the latent space.
        input_dim: The dimensionality of the input data.
        encoder: The encoder model.
        decoder: The decoder model.
        vae: The full VAE model.

    Methods:
        build_encoder(): Build the encoder part of the VAE.
        build_decoder(): Build the decoder part of the VAE.
        build_vae(learning_rate=0.0001, reconstruction_weight=0.5, kl_weight=0.5): Build␣
↪the full VAE model.
        sampling(args): Reparameterization trick to sample from the latent space.
        loss_function(inputs, outputs, z_mean, z_log_var, reconstruction_weight=0.5, kl_
↪weight=0.5): Compute the loss function for the VAE.
        train(x_train, y_train, batch_size, epochs, x_test, y_test): Train the VAE model␣
```

(continues on next page)

(continued from previous page)

```
↪on the provided data.
        plot_loss(): Plot the training loss.
    """

    def __init__(self, latent_dim=128, input_dim=256*256*1, learning_rate=0.0001,␣
↪reconstruction_weight=0.5, kl_weight=0.5):
        super(VAE, self).__init__()

        """
        :param latent_dim: The dimensionality of the latent space.
        :type latent_dim: int, optional
        :param input_dim: The dimensionality of the input data.
        :type input_dim: int, optional
        :param learning_rate: The learning rate for model training.
        :type learning_rate: float, optional
        :param reconstruction_weight: The weight for the reconstruction loss term in the␣
↪VAE loss function.
        :type reconstruction_weight: float, optional
        :param kl_weight: The weight for the Kullback-Leibler (KL) divergence loss term␣
↪in the VAE loss function.
        :type kl_weight: float, optional
        """
        self.latent_dim = latent_dim
        self.input_dim = input_dim
        self.encoder = self.build_encoder()
        self.decoder = self.build_decoder()
        self.vae = self.build_vae(learning_rate=learning_rate, reconstruction_
↪weight=reconstruction_weight, kl_weight=kl_weight)

    def build_encoder(self):
        """
        Build the encoder part of the autoencoder.

        :return: The encoder model.
        :rtype: tf.keras.Model
        """
        inputs = tf.keras.Input(shape=(self.input_dim,))
        x = layers.Dense(1024, activation='relu')(inputs)
        x = layers.Dense(512, activation='relu')(x)
        z_mean = layers.Dense(self.latent_dim)(x)
        z_log_var = layers.Dense(self.latent_dim)(x)
        z = layers.Lambda(self.sampling, output_shape=(self.latent_dim,))([z_mean, z_log_
↪var])
        return tf.keras.Model(inputs, [z_mean, z_log_var, z], name='encoder')

    def build_decoder(self):
        """
        Build the decoder part of the autoencoder.

        :return: The decoder model.
        :rtype: tf.keras.Model
        """
```

(continues on next page)

(continued from previous page)

```
        latent_inputs = tf.keras.Input(shape=(self.latent_dim,))
        x = layers.Dense(512, activation='relu')(latent_inputs)
        x = layers.Dense(1024, activation='relu')(x)
        outputs = layers.Dense(self.input_dim, activation='sigmoid')(x)
        return tf.keras.Model(latent_inputs, outputs, name='decoder')

    def build_vae(self, learning_rate=0.0001, reconstruction_weight=0.5, kl_weight=0.5):
        """
        Build the full autoencoder model.

        :param learning_rate: The learning rate for the optimizer.
        :type learning_rate: float, optional
        :param reconstruction_weight: The weight for the reconstruction loss term in the␣
↪VAE loss function.
        :type reconstruction_weight: float, optional
        :param kl_weight: The weight for the Kullback-Leibler (KL) divergence loss term␣
↪in the VAE loss function.
        :type kl_weight: float, optional
        :return: The full autoencoder model.
        :rtype: tf.keras.Model
        """
        encoder_inputs = tf.keras.Input(shape=(self.input_dim,))
        z_mean, z_log_var, z = self.encoder(encoder_inputs)
        decoder_outputs = self.decoder(z)
        vae = tf.keras.Model(encoder_inputs, decoder_outputs, name='vae')
        vae.add_loss(self.loss_function(encoder_inputs, decoder_outputs, z_mean, z_log_
↪var,reconstruction_weight=reconstruction_weight, kl_weight=kl_weight))
        vae.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate))
        return vae

    # Reparameterization trick
    def sampling(self, args):
        """
        Reparameterization trick to sample from the latent space.

        :param args: Tuple containing the mean and log variance tensors of the latent␣
↪space.
        :type args: tuple
        :return: The sampled data from the latent space.
        :rtype: tf.Tensor
        """
        z_mean, z_log_var = args
        epsilon = tf.keras.backend.random_normal(shape=(tf.keras.backend.shape(z_
↪mean)[0], self.latent_dim), mean=0., stddev=1.)
        return z_mean + tf.keras.backend.exp(0.5 * z_log_var) * epsilon

    # Custom loss function
    def loss_function(self, inputs, outputs, z_mean, z_log_var, reconstruction_weight=0.
↪5, kl_weight=0.5):
        """
        Compute the loss function for the Variational Autoencoder (VAE).
```

(continues on next page)

(continued from previous page)

```
        :param inputs: The input tensor representing the original data.
        :type inputs: tf.Tensor
        :param outputs: The output tensor representing the reconstructed data.
        :type outputs: tf.Tensor
        :param z_mean: The tensor representing the mean of the latent space.
        :type z_mean: tf.Tensor
        :param z_log_var: The tensor representing the log variance of the latent space.
        :type z_log_var: tf.Tensor
        :param reconstruction_weight: The weight for the reconstruction loss term in the␣
↪VAE loss function.
        :type reconstruction_weight: float, optional
        :param kl_weight: The weight for the Kullback-Leibler (KL) divergence loss term␣
↪in the VAE loss function.
        :type kl_weight: float, optional
        :return: The computed loss value.
        :rtype: tf.Tensor
        """
        reconstruction_loss = tf.keras.losses.binary_crossentropy(tf.keras.backend.
↪flatten(inputs), tf.keras.backend.flatten(outputs))
        reconstruction_loss *= self.input_dim  # Adjust for image size
        kl_loss = 1 + z_log_var - tf.keras.backend.square(z_mean) - tf.keras.backend.
↪exp(z_log_var)
        kl_loss = tf.keras.backend.mean(kl_loss, axis=-1)
        kl_loss *= -0.5
        return tf.keras.backend.mean(reconstruction_weight * reconstruction_loss + kl_
↪weight * kl_loss)

    def train(self, x_train, y_train, batch_size, epochs, x_test, y_test):
        """
        Train the VAE model on the provided training data.

        :param x_train: The input training data.
        :type x_train: np.ndarray or tf.Tensor
        :param y_train: The target training data.
        :type y_train: np.ndarray or tf.Tensor
        :param batch_size: The batch size for training.
        :type batch_size: int
        :param epochs: The number of epochs for training.
        :type epochs: int
        :param x_test: The input test data.
        :type x_test: np.ndarray or tf.Tensor
        :param y_test: The target test data.
        :type y_test: np.ndarray or tf.Tensor
        """
        early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0,␣
↪patience=3, verbose=1, mode='auto')
        his = self.vae.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,␣
↪validation_data = (x_test, y_test), callbacks=[early_stop], shuffle=True)
        self.his = his

    def plot_loss(self):
        """
```

(continues on next page)

(continued from previous page)

```
        Plot the training loss.
        """
        plt.plot(self.his.history['loss'])
        plt.plot(self.his.history['val_loss'])
        plt.legend(['Loss', 'Val_loss'])
        plt.show()
```

class LSTM.**ConvLSTMModel**(**args*, ***kwargs*)

A class used to represent a Convolutional LSTM Model.

Attributes:

convlstm_input_shape: Shape of the input data for the ConvLSTM layer. conv_lstm_layer: Convolutional LSTM layer created using the build_conv_lstm() method.

Methods:

build_conv_lstm(): Build a Convolutional LSTM (ConvLSTM) model with specified parameters. call(inputs): Run the ConvLSTM model on the provided inputs. train(x_train, y_train, x_test, y_test, lr = 0.01, batch_size=64, epochs=50): Train the ConvLSTM model with the provided training data. plot_loss(): Plot the model's training and validation loss over epochs.

build_conv_lstm()

Build a Convolutional LSTM (ConvLSTM) model.

Returns

A ConvLSTM model with a single ConvLSTM2D layer.

Return type

tf.keras.models.Sequential

call(*inputs*)

Execute the ConvLSTM model on the provided inputs.

Parameters

inputs (*np.ndarray or tf.Tensor*) – The input data or features to be passed through the ConvLSTM model.

Returns

The output of the ConvLSTM model.

Return type

tf.Tensor

plot_loss()

Plot the model's training and validation loss over epochs.

train(*x_train*, *y_train*, *x_test*, *y_test*, *lr=0.01*, *batch_size=64*, *epochs=50*)

Train the ConvLSTM model.

Parameters

- **x_train** (*np.ndarray or tf.Tensor*) – The input training data.
- **y_train** (*np.ndarray or tf.Tensor*) – The target training data.
- **x_test** (*np.ndarray or tf.Tensor*) – The input test data.
- **y_test** (*np.ndarray or tf.Tensor*) – The target test data.
- **epochs** (*int, optional*) – The number of epochs to train for.
- **batch_size** (*int, optional*) – The batch size for training.

- **lr** (*float, optional*) – The learning rate for the optimizer.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt

class ConvLSTMModel(tf.keras.Model):
    """
    A class used to represent a Convolutional LSTM Model.

    Attributes:
        convlstm_input_shape: Shape of the input data for the ConvLSTM layer.
        conv_lstm_layer: Convolutional LSTM layer created using the build_conv_lstm()␣
↪method.

    Methods:
        build_conv_lstm(): Build a Convolutional LSTM (ConvLSTM) model with specified␣
↪parameters.
        call(inputs): Run the ConvLSTM model on the provided inputs.
        train(x_train, y_train, x_test, y_test, lr = 0.01, batch_size=64, epochs=50):␣
↪Train the ConvLSTM model with the provided training data.
        plot_loss(): Plot the model's training and validation loss over epochs.
    """

    def __init__(self, convlstm_input_shape=(None, None, None, None)):
        super(ConvLSTMModel, self).__init__()
        """
        Initialize the ConvLSTMModel class.

        :param convlstm_input_shape: The shape of the input data for the ConvLSTM layer.
        :type onvlstm_input_shape: tuple
        """
        self.convlstm_input_shape = convlstm_input_shape
        self.conv_lstm_layer = self.build_conv_lstm()

    def build_conv_lstm(self):
        """
        Build a Convolutional LSTM (ConvLSTM) model.

        :return: A ConvLSTM model with a single ConvLSTM2D layer.
        :rtype: tf.keras.models.Sequential
        """
        model = tf.keras.models.Sequential()
        model.add(layers.ConvLSTM2D(filters=1, kernel_size=(3, 3), input_shape=self.
↪convlstm_input_shape, padding='same', activation='sigmoid', return_sequences=False))
        return model

    def call(self, inputs):
        """
        Execute the ConvLSTM model on the provided inputs.
```

(continues on next page)

(continued from previous page)

```
        :param inputs: The input data or features to be passed through the ConvLSTM␣
↪model.
        :type inputs: np.ndarray or tf.Tensor
        :return: The output of the ConvLSTM model.
        :rtype: tf.Tensor
        """
        return self.conv_lstm_layer(inputs)

    def train(self, x_train, y_train, x_test, y_test, lr=0.01, batch_size=64, epochs=50):
        """
        Train the ConvLSTM model.

        :param x_train: The input training data.
        :type x_train: np.ndarray or tf.Tensor
        :param y_train: The target training data.
        :type y_train: np.ndarray or tf.Tensor
        :param x_test: The input test data.
        :type x_test: np.ndarray or tf.Tensor
        :param y_test: The target test data.
        :type y_test: np.ndarray or tf.Tensor
        :param epochs: The number of epochs to train for.
        :type epochs: int, optional
        :param batch_size: The batch size for training.
        :type batch_size: int, optional
        :param lr: The learning rate for the optimizer.
        :type lr: float, optional
        """
        # Create an Adam optimizer with the given learning rate
        opt = Adam(learning_rate = lr)
        # Compile the model
        self.compile(optimizer=opt, loss='mean_squared_error')

        # Train the model
        his = self.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=2,
↪ validation_data = (x_test, y_test), shuffle = True)
        self.his = his

    def plot_loss(self):
        """
        Plot the model's training and validation loss over epochs.
        """
        plt.plot(self.his.history['loss'])
        plt.plot(self.his.history['val_loss'])
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title('Training Loss Curve')
        plt.legend(['Loss', 'Val_loss'])
        plt.show()
```

tools.**generate_lstm_input**(*data*, *num_batches*)

Generates input data set for an LSTM model from the given data.

Parameters

- **data** (*np.ndarray or tf.Tensor*) – The original dataset from which the inputs are generated.
- **num_batches** (*int*) – The number of batches to be generated.

Returns
The input data for the model and the target data.

Return type
tuple of np.ndarray

tools.**plot_data**(*data*, *ts*, *titles*)

Creates a figure and a set of subplots.

Parameters

- **data** (*np.ndarray or tf.Tensor*) – dataset
- **ts** (*list of int*) – timesteps to plot
- **titles** (*list of str*) – titles for each subplot

tools.**pred_n_steps**(*model*, *dd*, *steps*)

Predict behaviors at (t+n) for given model

Parameters

- **model** (*tf.keras.Model*) – model used to predict
- **dd** (*np.ndarray or tf.Tensor*) – data used for prediction:
- **steps** (*int*) – n timesteps later

Returns
predicted data at timestep (t+n)

Return type
np.ndarray or tf.Tensor

tools.**reshape_2d**(*dd*)

This function reshape a 3d dataset to 2d kept first dimension unchanged

Parameters
dd (*np.ndarray or tf.Tensor*) – 3d dataset

Returns
reshaped 2d dataset

Return type
np.ndarray or tf.Tensor

tools.**reshape_background**(*background*)

Reshape the background data for LSTM model

Parameters
background (*np.ndarray or tf.Tensor*) – Input data for LSTM model

Returns
Reshaped input data and target data

Return type
np.ndarray or tf.Tensor

tools.**select_samples**(*dd*, *start=0*, *step=100*)

This function selects samples from a training set by taking every 'step'-th sample.

Parameters

- **dd** (*np.ndarray or tf.Tensor*) – 2d or 3d dataset
- **start** (*int*) – the first element to remove
- **step** (*int*) – the step size for sample selection

Returns

dataset after selection

Return type

np.ndarray or tf.Tensor

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def reshape_2d(dd):
    """
    This function reshape a 3d dataset to 2d kept first dimension unchanged

    :param dd: 3d dataset
    :type dd: np.ndarray or tf.Tensor
    :return: reshaped 2d dataset
    :rtype: np.ndarray or tf.Tensor
    """
    dd_new = dd.reshape(-1, dd.shape[1]*dd.shape[2])
    return dd_new

def select_samples(dd, start=0, step=100):
    """
    This function selects samples from a training set by taking every 'step'-th sample.

    :param dd: 2d or 3d dataset
    :type dd: np.ndarray or tf.Tensor
    :param start: the first element to remove
    :type start: int
    :param step: the step size for sample selection
    :type step: int
    :return: dataset after selection
    :rtype: np.ndarray or tf.Tensor
    """
    indices = np.array(range(start, dd.shape[0], step))
    mask = np.ones(dd.shape[0], bool)
    mask[indices] = False
    dd_new = dd[mask, :].reshape(-1, dd.shape[1])
    return dd_new

def plot_data(data, ts, titles):
    """
    Creates a figure and a set of subplots.
```

(continues on next page)

(continued from previous page)

```
    :param data: dataset
    :type data: np.ndarray or tf.Tensor
    :param ts: timesteps to plot
    :type ts: list of int
    :param titles: titles for each subplot
    :type titles: list of str
    """
    fig, axes = plt.subplots(1, len(ts), figsize=(20, 8))
    for t, title, ax in zip(ts, titles, axes.ravel()):
        im = ax.imshow(data[t])
        ax.set_title(title)
    plt.show()


def pred_n_steps(model, dd, steps):
    """
    Predict behaviors at (t+n) for given model

    :param model: model used to predict
    :type model: tf.keras.Model
    :param dd: data used for prediction:
    :type dd: np.ndarray or tf.Tensor
    :param steps: n timesteps later
    :type steps: int
    :return: predicted data at timestep (t+n)
    :rtype: np.ndarray or tf.Tensor
    """
    y = np.copy(dd)
    for i in range(steps):
      y = model.predict(y)
    return y

def generate_lstm_input(data, num_batches):
    """
    Generates input data set for an LSTM model from the given data.

    :param data: The original dataset from which the inputs are generated.
    :type data: np.ndarray or tf.Tensor
    :param num_batches: The number of batches to be generated.
    :type num_batches: int
    :return: The input data for the model and the target data.
    :rtype: tuple of np.ndarray
    """
    lstm_input = []

    for n in range(num_batches):
        # Select a segment(n_th segment) of the input data
        data_segment = data[n*100:(n+1)*100,:,:]

        new_batch = []
        for i in range(79):
            selected_train = data_segment[i:21+i:10, :, :]  # Select every 10th time␣
```

(continues on next page)

(continued from previous page)

```
↪step before time step 100
            new_batch.append(selected_train)

        # Reshape the data to be suitable for input to an LSTM model
        new_batch = np.reshape(new_batch, (-1, 3, 256, 256, 1))

        # Append the prepared sample to the lstm_input list
        lstm_input.append(new_batch)

    # Convert the list to a numpy array for further processing
    lstm_input = np.array(lstm_input)

    # Reshape the array to have a uniform shape for all samples
    lstm_input = np.reshape(lstm_input, (-1, 3, 256, 256, 1))

    return lstm_input[:,:2,:,:], lstm_input[:,-1,:,:]


def reshape_background(background):
    """
    Reshape the background data for LSTM model

    :param background: Input data for LSTM model
    :type background: np.ndarray or tf.Tensor
    :return: Reshaped input data and target data
    :rtype: np.ndarray or tf.Tensor
    """
    back_pred = []
    for i in range(3):
        selected_back = background[i:i+3, :, :]  # Select the first 3 time step to␣
↪predict
        back_pred.append(selected_back)
    background_reshape = np.reshape(back_pred, (-1, 3, 256, 256, 1))

    return background_reshape[:,:2,:,:], background_reshape[:,-1,:,:]
```

test_DA.**test_autoencoder_build_ae**(*autoencoder_instance*)

Test function to validate the build_ae method of the Autoencoder class.

Args:

autoencoder_instance (Autoencoder): An instance of the Autoencoder class.

Raises:

AssertionError: If the output of build_ae is not an instance of tf.keras.Model.

Example:

```
>>> autoencoder = Autoencoder()
>>> test_autoencoder_build_ae(autoencoder)
```

test_DA.**test_autoencoder_build_decoder**(*autoencoder_instance*)

Test function to validate the build_decoder method of the Autoencoder class.

Args:

autoencoder_instance (Autoencoder): An instance of the Autoencoder class.

Raises:
AssertionError: If the output of build_decoder is not an instance of tf.keras.Model.

Example:

```
>>> autoencoder = Autoencoder()
>>> test_autoencoder_build_decoder(autoencoder)
```

test_DA.**test_autoencoder_build_encoder**(*autoencoder_instance*)

Test function to validate the build_encoder method of the Autoencoder class.

Args:
autoencoder_instance (Autoencoder): An instance of the Autoencoder class.

Raises:
AssertionError: If the output of build_encoder is not an instance of tf.keras.Model.

Example:

```
>>> autoencoder = Autoencoder()
>>> test_autoencoder_build_encoder(autoencoder)
```

test_DA.**test_autoencoder_train**(*autoencoder_instance*)

Test function to validate the train method of the Autoencoder class.

Args:
autoencoder_instance (Autoencoder): An instance of the Autoencoder class.

Raises:
AssertionError: If the Autoencoder instance does not have the 'his' attribute after training.

Example:

```
>>> autoencoder = Autoencoder()
>>> test_autoencoder_train(autoencoder)
```

test_DA.**test_data_assimilation_KalmanGain**(*data_assimilation_instance*)

Test function to validate the KalmanGain method of the DataAssimilation class.

Args:
data_assimilation_instance (DataAssimilation): An instance of the DataAssimilation class.

Raises:
AssertionError: If the shape of the Kalman gain matrix is not as expected.

Example:

```
>>> enc_model = Autoencoder().encoder
>>> dec_model = Autoencoder().decoder
>>> data_assimilation = DataAssimilation(enc_model, dec_model)
>>> B = np.random.randn(64, 64)
>>> H = np.eye(64)
>>> R = np.eye(64) * 0.01
>>> test_data_assimilation_KalmanGain(data_assimilation)
```

test_DA.**test_data_assimilation_assimilate**(*data_assimilation_instance*)

Test function to validate the assimilate method of the DataAssimilation class.

Args:
data_assimilation_instance (DataAssimilation): An instance of the DataAssimilation class.

Raises:
AssertionError: If the shape of the updated decoded data is not as expected.

Example:

```
>>> enc_model = Autoencoder().encoder
>>> dec_model = Autoencoder().decoder
>>> data_assimilation = DataAssimilation(enc_model, dec_model)
>>> bg_data = np.random.randn(10, 256, 256)
>>> obs_data = np.random.randn(10, 256, 256)
```

test_DA.**test_data_assimilation_covariance_matrix**(*data_assimilation_instance*)

Test function to validate the covariance_matrix method of the DataAssimilation class.

Args:
data_assimilation_instance (DataAssimilation): An instance of the DataAssimilation class.

Raises:
AssertionError: If the shape of the covariance matrix is not as expected.

Example:

```
>>> enc_model = Autoencoder().encoder
>>> dec_model = Autoencoder().decoder
>>> data_assimilation = DataAssimilation(enc_model, dec_model)
>>> test_data_assimilation_covariance_matrix(data_assimilation)
```

test_DA.**test_data_assimilation_update_prediction**(*data_assimilation_instance*)

Test function to validate the update_prediction method of the DataAssimilation class.

Args:
data_assimilation_instance (DataAssimilation): An instance of the DataAssimilation class.

Raises:
AssertionError: If the shape of the updated prediction is not as expected.

Example:

```
>>> enc_model = Autoencoder().encoder
>>> dec_model = Autoencoder().decoder
>>> data_assimilation = DataAssimilation(enc_model, dec_model)
>>> x = np.random.randn(64)
>>> K = np.random.randn(64, 64)
>>> H = np.eye(64)
>>> y = np.random.randn(64)
>>> test_data_assimilation_update_prediction(data_assimilation)
```

```
import pytest
import numpy as np
import tensorflow as tf
from DA import Autoencoder, DataAssimilation

@pytest.fixture
def autoencoder_instance():
    return Autoencoder()

@pytest.fixture
```

(continues on next page)

(continued from previous page)

```
def data_assimilation_instance(autoencoder_instance):
    enc_model = autoencoder_instance.encoder
    dec_model = autoencoder_instance.decoder
    return DataAssimilation(enc_model, dec_model)

def test_autoencoder_build_encoder(autoencoder_instance):
    """
    Test function to validate the build_encoder method of the Autoencoder class.

    Args:
        autoencoder_instance (Autoencoder): An instance of the Autoencoder class.

    Raises:
        AssertionError: If the output of build_encoder is not an instance of tf.keras.
↪Model.

    Example:
        >>> autoencoder = Autoencoder()
        >>> test_autoencoder_build_encoder(autoencoder)
    """
    encoder = autoencoder_instance.build_encoder()
    assert isinstance(encoder, tf.keras.Model)

def test_autoencoder_build_decoder(autoencoder_instance):
    """
    Test function to validate the build_decoder method of the Autoencoder class.

    Args:
        autoencoder_instance (Autoencoder): An instance of the Autoencoder class.

    Raises:
        AssertionError: If the output of build_decoder is not an instance of tf.keras.
↪Model.

    Example:
        >>> autoencoder = Autoencoder()
        >>> test_autoencoder_build_decoder(autoencoder)
    """
    decoder = autoencoder_instance.build_decoder()
    assert isinstance(decoder, tf.keras.Model)

def test_autoencoder_build_ae(autoencoder_instance):
    """
    Test function to validate the build_ae method of the Autoencoder class.

    Args:
        autoencoder_instance (Autoencoder): An instance of the Autoencoder class.

    Raises:
        AssertionError: If the output of build_ae is not an instance of tf.keras.Model.

    Example:
```

(continues on next page)

(continued from previous page)

```
        >>> autoencoder = Autoencoder()
        >>> test_autoencoder_build_ae(autoencoder)
    """
    ae_model = autoencoder_instance.build_ae()
    assert isinstance(ae_model, tf.keras.Model)

def test_autoencoder_train(autoencoder_instance):
    """
    Test function to validate the train method of the Autoencoder class.

    Args:
        autoencoder_instance (Autoencoder): An instance of the Autoencoder class.

    Raises:
        AssertionError: If the Autoencoder instance does not have the 'his' attribute␣
↪after training.

    Example:
        >>> autoencoder = Autoencoder()
        >>> test_autoencoder_train(autoencoder)
    """
    train_x = np.random.randn(100, 256, 256, 1)
    train_y = np.random.randn(100, 256, 256, 1)
    test_x = np.random.randn(20, 256, 256, 1)
    test_y = np.random.randn(20, 256, 256, 1)
    autoencoder_instance.train(train_x, train_y, test_x, test_y)
    assert hasattr(autoencoder_instance, 'his')

def test_data_assimilation_covariance_matrix(data_assimilation_instance):
    """
    Test function to validate the covariance_matrix method of the DataAssimilation class.

    Args:
        data_assimilation_instance (DataAssimilation): An instance of the␣
↪DataAssimilation class.

    Raises:
        AssertionError: If the shape of the covariance matrix is not as expected.

    Example:
        >>> enc_model = Autoencoder().encoder
        >>> dec_model = Autoencoder().decoder
        >>> data_assimilation = DataAssimilation(enc_model, dec_model)
        >>> test_data_assimilation_covariance_matrix(data_assimilation)
    """
    X = np.random.randn(100, 100)
    cov_matrix = data_assimilation_instance.covariance_matrix(X)
    assert cov_matrix.shape == (X.shape[0], X.shape[0])



def test_data_assimilation_update_prediction(data_assimilation_instance):
```

(continues on next page)

(continued from previous page)

```
    """
    Test function to validate the update_prediction method of the DataAssimilation class.

    Args:
        data_assimilation_instance (DataAssimilation): An instance of the␣
↪DataAssimilation class.

    Raises:
        AssertionError: If the shape of the updated prediction is not as expected.

    Example:
        >>> enc_model = Autoencoder().encoder
        >>> dec_model = Autoencoder().decoder
        >>> data_assimilation = DataAssimilation(enc_model, dec_model)
        >>> x = np.random.randn(64)
        >>> K = np.random.randn(64, 64)
        >>> H = np.eye(64)
        >>> y = np.random.randn(64)
        >>> test_data_assimilation_update_prediction(data_assimilation)
    """
    x = np.random.randn(data_assimilation_instance.latent_dim)
    K = np.random.randn(data_assimilation_instance.latent_dim, data_assimilation_
↪instance.latent_dim)
    H = np.eye(data_assimilation_instance.latent_dim)
    y = np.random.randn(data_assimilation_instance.latent_dim)

    updated_prediction = data_assimilation_instance.update_prediction(x, K, H, y)
    assert updated_prediction.shape == (data_assimilation_instance.latent_dim,)

def test_data_assimilation_KalmanGain(data_assimilation_instance):
    """
    Test function to validate the KalmanGain method of the DataAssimilation class.

    Args:
        data_assimilation_instance (DataAssimilation): An instance of the␣
↪DataAssimilation class.

    Raises:
        AssertionError: If the shape of the Kalman gain matrix is not as expected.

    Example:
        >>> enc_model = Autoencoder().encoder
        >>> dec_model = Autoencoder().decoder
        >>> data_assimilation = DataAssimilation(enc_model, dec_model)
        >>> B = np.random.randn(64, 64)
        >>> H = np.eye(64)
        >>> R = np.eye(64) * 0.01
        >>> test_data_assimilation_KalmanGain(data_assimilation)
    """
    B = np.random.randn(data_assimilation_instance.latent_dim, data_assimilation_
↪instance.latent_dim)
    H = np.eye(data_assimilation_instance.latent_dim)
```

(continues on next page)

(continued from previous page)

```
    R = np.eye(data_assimilation_instance.latent_dim) * data_assimilation_instance.R_val

    Kalman_gain = data_assimilation_instance.KalmanGain(B, H, R)
    assert Kalman_gain.shape == (data_assimilation_instance.latent_dim, data_
↪assimilation_instance.latent_dim)

def test_data_assimilation_assimilate(data_assimilation_instance):
    """
    Test function to validate the assimilate method of the DataAssimilation class.

    Args:
        data_assimilation_instance (DataAssimilation): An instance of the␣
↪DataAssimilation class.

    Raises:
        AssertionError: If the shape of the updated decoded data is not as expected.

    Example:
        >>> enc_model = Autoencoder().encoder
        >>> dec_model = Autoencoder().decoder
        >>> data_assimilation = DataAssimilation(enc_model, dec_model)
        >>> bg_data = np.random.randn(10, 256, 256)
        >>> obs_data = np.random.randn(10, 256, 256)
       """
    bg_data = np.random.randn(10, 256, 256)
    obs_data = np.random.randn(10, 256, 256)

    updated_decoded_data = data_assimilation_instance.assimilate(bg_data, obs_data)
    assert updated_decoded_data.shape == (10, 256, 256, 1)
```

test_VAE.**batch_size**()

Fixture for batch size.

test_VAE.**epochs**()

Fixture for number of epochs.

test_VAE.**plot_loss**(*vae_instance*)

Plot and save the loss plot for the VAE instance.

Args:

vae_instance (VAE): An instance of the VAE class.

test_VAE.**test_build_decoder**(*vae_instance*)

Validates the construction of the decoder.

Args:

vae_instance (VAE): An instance of the VAE class.

Raises:

AssertionError: If the decoder is not an instance of *tf.keras.Model*,
if the number of decoder inputs is not equal to 1, or if the number of decoder outputs is not equal to 1.

test_VAE.**test_build_encoder**(*vae_instance*)

Validates the construction of the encoder.

Args:
vae_instance (VAE): An instance of the VAE class.

Raises:

AssertionError: If the encoder is not an instance of *tf.keras.Model*,
if the number of encoder inputs is not equal to 1, or if the number of encoder outputs is not equal to 3.

test_VAE.**test_build_vae**(*vae_instance*)

Validates the construction of the VAE.

Args:
vae_instance (VAE): An instance of the VAE class.

Raises:

AssertionError: If the VAE is not an instance of *tf.keras.Model*,
if the number of VAE inputs is not equal to 1, or if the number of VAE outputs is not equal to 1.

test_VAE.**test_loss_function**(*vae_instance*)

Validates the loss_function method in the VAE.

Args:
vae_instance (VAE): An instance of the VAE class.

Raises:
AssertionError: If the loss function does not return the expected shape or dtype.

test_VAE.**test_plot_loss**(*vae_instance*, *x_train_y_train*, *x_test_y_test*, *batch_size*, *epochs*)

Test the plot_loss function for the VAE instance.

Args:
vae_instance (VAE): An instance of the VAE class. x_train_y_train (tuple): Training data as a tuple of input and output arrays. x_test_y_test (tuple): Validation data as a tuple of input and output arrays. batch_size (int): Batch size for training. epochs (int): Number of training epochs.

Raises:
AssertionError: If the loss plot file is not created.

test_VAE.**test_sampling**(*vae_instance*)

Validates the sampling method in the VAE.

Args:
vae_instance (VAE): An instance of the VAE class.

Raises:
AssertionError: If the sampling method does not return the expected shape or dtype.

test_VAE.**test_train**(*vae_instance*, *x_train_y_train*, *x_test_y_test*, *batch_size*, *epochs*)

Test the training function of the VAE class.

Args:
vae_instance (VAE): An instance of the VAE class. x_train_y_train (tuple): Training data as a tuple of input and output arrays. x_test_y_test (tuple): Validation data as a tuple of input and output arrays. batch_size (int): Batch size for training. epochs (int): Number of training epochs.

Raises:
AssertionError: If training fails with an error.

test_VAE.**test_vae_inputs**()

Test dimensions and properties of the VAE inputs.

Raises:
AssertionError: If any of the dimension checks fail or if the properties of inputs are invalid.

test_VAE.**vae_instance**()
Fixture for creating an instance of the VAE class.

Returns:
VAE: An instance of the VAE class.

test_VAE.**x_test_y_test**()
Create a sample validation set.

test_VAE.**x_train_y_train**()
Create a sample training set.

```
# import VAE
from VAE import VAE
import pytest
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import os

@pytest.fixture
def vae_instance():
    """Fixture for creating an instance of the VAE class.

    Returns:
        VAE: An instance of the VAE class.
    """
    return VAE(latent_dim=128, input_dim=256*256*1)


def test_vae_inputs():
    """Test dimensions and properties of the VAE inputs.

    Raises:
        AssertionError: If any of the dimension checks fail or if the properties of␣
↪inputs are invalid.
    """
    latent_dim = 128
    input_dim = 256 * 256 * 1

    # Check dimensions are not 0
    assert isinstance(latent_dim, int) and latent_dim != 0
    assert isinstance(input_dim, int) and input_dim != 0

    try:
        # only pass in the parameters that the VAE constructor can accept
        vae = VAE(latent_dim, input_dim)
    except Exception as e:
        pytest.fail(f"Failed to create VAE: {e}")
```

(continues on next page)

(continued from previous page)

```
def test_build_encoder(vae_instance):
    """Validates the construction of the encoder.

    Args:
        vae_instance (VAE): An instance of the VAE class.

    Raises:
        AssertionError: If the encoder is not an instance of `tf.keras.Model`,
            if the number of encoder inputs is not equal to 1,
            or if the number of encoder outputs is not equal to 3.
    """
    encoder = vae_instance.build_encoder()
    assert isinstance(encoder, tf.keras.Model)
    assert len(encoder.inputs) == 1
    assert len(encoder.outputs) == 3


def test_build_decoder(vae_instance):
    """Validates the construction of the decoder.

    Args:
        vae_instance (VAE): An instance of the VAE class.

    Raises:
        AssertionError: If the decoder is not an instance of `tf.keras.Model`,
            if the number of decoder inputs is not equal to 1,
            or if the number of decoder outputs is not equal to 1.
    """
    decoder = vae_instance.build_decoder()
    assert isinstance(decoder, tf.keras.Model)
    assert len(decoder.inputs) == 1
    assert len(decoder.outputs) == 1


def test_build_vae(vae_instance):
    """Validates the construction of the VAE.

    Args:
        vae_instance (VAE): An instance of the VAE class.

    Raises:
        AssertionError: If the VAE is not an instance of `tf.keras.Model`,
            if the number of VAE inputs is not equal to 1,
            or if the number of VAE outputs is not equal to 1.
    """
    vae = vae_instance.build_vae()
    assert isinstance(vae, tf.keras.Model)
    assert len(vae.inputs) == 1
    assert len(vae.outputs) == 1


def test_sampling(vae_instance):
```

(continues on next page)

(continued from previous page)

```
    """Validates the sampling method in the VAE.

    Args:
        vae_instance (VAE): An instance of the VAE class.

    Raises:
        AssertionError: If the sampling method does not return the expected shape or␣
↪dtype.
    """
    # Create dummy inputs
    dummy_args = (tf.random.normal(shape=(10, vae_instance.latent_dim)),
                  tf.random.normal(shape=(10, vae_instance.latent_dim)))

    # Call the sampling method
    sampled_output = vae_instance.sampling(dummy_args)

    # Check the shape and dtype of the sampled output
    expected_shape = (10, vae_instance.latent_dim)
    expected_dtype = tf.float32
    assert sampled_output.shape == expected_shape, f"Expected shape: {expected_shape},␣
↪Actual shape: {sampled_output.shape}"
    assert sampled_output.dtype == expected_dtype, f"Expected dtype: {expected_dtype},␣
↪Actual dtype: {sampled_output.dtype}"


def test_loss_function(vae_instance):
    """Validates the loss_function method in the VAE.

    Args:
        vae_instance (VAE): An instance of the VAE class.

    Raises:
        AssertionError: If the loss function does not return the expected shape or dtype.
    """
    # Create dummy inputs and outputs
    inputs = tf.random.normal(shape=(10, vae_instance.input_dim))
    outputs = tf.random.normal(shape=(10, vae_instance.input_dim))
    z_mean = tf.random.normal(shape=(10, vae_instance.latent_dim))
    z_log_var = tf.random.normal(shape=(10, vae_instance.latent_dim))

    # Call the loss_function method
    loss = vae_instance.loss_function(inputs, outputs, z_mean, z_log_var)

    # Check the shape and dtype of the loss
    expected_shape = ()
    expected_dtype = tf.float32
    assert loss.shape == expected_shape, f"Expected shape: {expected_shape}, Actual␣
↪shape: {loss.shape}"
    assert loss.dtype == expected_dtype, f"Expected dtype: {expected_dtype}, Actual␣
↪dtype: {loss.dtype}"
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture
def x_train_y_train():
    """Create a sample training set."""
    return np.random.rand(100, 256 * 256 * 1), np.random.rand(100, 256 * 256 * 1)


@pytest.fixture
def x_test_y_test():
    """Create a sample validation set."""
    return np.random.rand(20, 256 * 256 * 1), np.random.rand(20, 256 * 256 * 1)


@pytest.fixture
def batch_size():
    """Fixture for batch size."""
    return 64


@pytest.fixture
def epochs():
    """Fixture for number of epochs."""
    return 10


def test_train(vae_instance, x_train_y_train, x_test_y_test, batch_size, epochs):
    """Test the training function of the VAE class.

    Args:
        vae_instance (VAE): An instance of the VAE class.
        x_train_y_train (tuple): Training data as a tuple of input and output arrays.
        x_test_y_test (tuple): Validation data as a tuple of input and output arrays.
        batch_size (int): Batch size for training.
        epochs (int): Number of training epochs.

    Raises:
        AssertionError: If training fails with an error.
    """
    x_train, y_train = x_train_y_train
    x_test, y_test = x_test_y_test
    try:
        vae_instance.train(x_train, y_train, batch_size, epochs, x_test, y_test)
    except Exception as e:
        pytest.fail(f"Training failed with error: {e}")


def plot_loss(vae_instance):
    """Plot and save the loss plot for the VAE instance.

    Args:
        vae_instance (VAE): An instance of the VAE class.
    """
    vae_instance.plot_loss()
```

(continues on next page)

(continued from previous page)

```
    plt.savefig("loss_plot.png")


def test_plot_loss(vae_instance, x_train_y_train, x_test_y_test, batch_size, epochs):
    """Test the plot_loss function for the VAE instance.

    Args:
        vae_instance (VAE): An instance of the VAE class.
        x_train_y_train (tuple): Training data as a tuple of input and output arrays.
        x_test_y_test (tuple): Validation data as a tuple of input and output arrays.
        batch_size (int): Batch size for training.
        epochs (int): Number of training epochs.

    Raises:
        AssertionError: If the loss plot file is not created.
    """
    x_train, y_train = x_train_y_train
    x_test, y_test = x_test_y_test
    vae_instance.train(x_train, y_train, batch_size, epochs, x_test, y_test)
    plot_loss(vae_instance)
    assert os.path.isfile("loss_plot.png"), "Loss plot file is not created."
```

```
import pytest
import numpy as np
import LSTM
import tensorflow as tf

class TestLSTMModel():
    @pytest.fixture(autouse=True)
    def setUp(self):
        '''
        Set up the model and variables required for testing
        '''
        self.model = LSTM.ConvLSTMModel(convlstm_input_shape=(3, 256, 256, 1))
        self.x_train = np.random.rand(10, 3, 256, 256, 1)
        self.y_train = np.random.rand(10, 256, 256, 1)
        self.x_test = np.random.rand(2, 3, 256, 256, 1)
        self.y_test = np.random.rand(2, 256, 256, 1)

    def test_build_conv_lstm(self):
        '''
        Test the 'build_conv_lstm' function by checking the type of the model
        '''
        conv_lstm = self.model.build_conv_lstm()
        assert isinstance(conv_lstm, tf.keras.models.Sequential)

    def test_call(self):
        '''
        Test the 'call' function by checking the shape of the model output
        '''
        output = self.model(self.x_train)
        assert output.shape == self.y_train.shape
```

(continues on next page)

(continued from previous page)

```
    def test_train(self):
        '''
        Test the 'train' function by checking the shape of the model output
        '''
        self.model.train(self.x_train, self.y_train, self.x_test, self.y_test, epochs=1)
        assert len(self.model.his.history['loss']) == 1

    def test_plot_loss(self):
        '''
        Test the 'plot_loss' function by checking if it can run successfully
        '''
        try:
            # Train the model for plotting the loss
            self.model.train(self.x_train, self.y_train, self.x_test, self.y_test,␣
↪epochs=1)
            # Call the plot function
            self.model.plot_loss()

        except Exception as e:
            assert False, f"plot_loss() raised an exception: {e}"
```

```
import numpy as np
import pytest
from tools import reshape_2d, select_samples, pred_n_steps, plot_data, generate_lstm_
↪input, reshape_background

def test_reshape_2d():
    dd = np.random.rand(10, 5, 5)
    dd_new = reshape_2d(dd)
    assert dd_new.shape == (10, 25)

def test_select_samples():
    dd = np.random.rand(1000, 5)
    dd_new = select_samples(dd, start=0, step=100)
    assert dd_new.shape == (990, 5)

def test_generate_lstm_input():
    data = np.random.rand(12500, 256, 256)
    num_batches = 125

    input_data, output_data = generate_lstm_input(data, num_batches)
    assert input_data.shape == (9875, 2, 256, 256, 1)
    assert output_data.shape == (9875, 256, 256, 1)

def test_reshape_background():
    background = np.random.rand(5, 256, 256)
    input_data, output_data = reshape_background(background)
    assert input_data.shape == (3, 2, 256, 256, 1)
    assert output_data.shape == (3, 256, 256, 1)
```

(continues on next page)

(continued from previous page)

```
def test_plot_data():
    data = np.random.rand(10, 256, 256)
    ts = [0, 1, 2]
    titles = ["t=0", "t=1", "t=2"]

    # We are testing the function call only because it does not have a return statement
    # We assume that if it does not raise an exception, it has passed the test
    try:
        plot_data(data, ts, titles)
    except Exception as e:
        pytest.fail(f"plot_data raised exception {e}")

def test_pred_n_steps():
    from keras.models import Sequential
    from keras.layers import Dense

    # Creating a simple model for testing
    model = Sequential()
    model.add(Dense(units=64, activation='relu', input_shape=(10,)))
    model.add(Dense(units=10, activation='softmax'))
    model.compile(optimizer='sgd', loss='categorical_crossentropy')

    dd = np.random.rand(10, 10)
    steps = 2

    assert pred_n_steps(model, dd, steps).shape == (10,10)
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

U

V

X