In [ ]:

```
 
```

Necessary imports

In [ ]:

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import MeanSquaredError
import matplotlib.pyplot as plt
import numpy as np
import matplotlib
from tqdm import tqdm
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Reshape, Flatten
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Reshape, Dense
import matplotlib.pyplot as plt
import time
starttime = time.time()
# matplotlib.use('TkAgg')
```

In [ ]:

```
 
```

Dummy data generation

We consider `n_sensors` sensors, each having data as a time series of sine wave with some uniform zero-mean noise added. Each sensors has a phase difference. The parameters are:

- `n_sensors` : number of sensors in the dataset; 3 for this data
- `cycles` : How many time periods of sine wave in full data set; 300 for this data
- `resolution` : Total number of time stamps in `cycles` time periods; 10K for this data
- `phase` : phase difference between sensors; 9 for this data
- `PERIODICITY` : computed as

  $$\frac{\text{resolution}}{\text{cycles}}$$

  . This is also referred to as the daily offset in this demo script, since the periodicity is later used to compute the nearest neighbours in temporal bands at multiples of periodicity; (cf. `Figure 2` . in the paper)
- `NOISE` : magnitude of noise in data; default 0.2

In [ ]:

```
n_sensors = 3

import os
if not os.path.exists("plots_from_demo_data"):
    os.mkdir("plots_from_demo_data")

def generate_continuous_dataset():
    """
    Generates a continuous dataset based on sine wave cycles with added noise.

    The function creates a dataset of sine waves for a specified number of cycles and resolution.
    Noise is added to simulate real-world data. The dataset is plotted and saved as an image.

    Returns:
        tuple: A tuple containing the generated dataset and the periodicity of the sine waves.
    """
    cycles = 300 # how many sine cycles
    resolution = 10000 # how many datapoints to generate
    phase = 9

    length = np.pi * 2 * cycles

    sensor_list = []
    NOISE_LEVEL = 0.2

    for i in range(n_sensors):
        values = np.sin( (np.arange(0, length, length / resolution)) - phase * i )
        shape = values.shape
        noise_with_mean_zero = (np.random.rand(*shape) - 0.5)

        sensor_list.append ( values + noise_with_mean_zero * NOISE_LEVEL )

    dataset = np.random.rand(len(sensor_list), resolution) * 0

    for counter, sensor in enumerate(sensor_list):
        dataset[counter, :] = sensor
        plt.plot(sensor, label= "sensor " + str(counter+1))

    PERIODICITY = int(resolution/cycles)
    plt.plot(range(phase, PERIODICITY + phase), [1.5] * PERIODICITY, color="black", linestyle="--", label="daily offset " + str (PERIODICITY))

    plt.legend()
    plt.ylim(-2, 2)
    plt.xlim(0, 120)
    plt.title("Raw data")
    plt.savefig("plots_from_demo_data/two_sensors_time_series.jpg", dpi=300)
    plt.show()
    plt.clf()

    return dataset, PERIODICITY
large_dataset, PERIODICITY = generate_continuous_dataset()
```

Convert to supervised labels: The time series needs to converted to supervised labels in order to be modelled by the DL models. This is accomplished using the function `sample_blocks_for_XY`. The `i_o` value decides the number of time frames in input and ouput data. (cf. Figure 1 from the paper)

In [ ]:

```python
def dataloader(large_dataset, timestamp, i_o):
    """
    Loads a subset of data from a larger dataset.
    The dataloader should be able to access the neighbours using a timestamp index.

    Args:
        large_dataset (np.ndarray): The larger dataset from which to load data.
        i (int): The starting index for loading data.
        i_o (int): The size of the input-output blocks.

    Returns:
        tuple: A tuple containing the X and Y blocks for the specified index.

    Refer to: https://github.com/mie-lab/Complexity-Aware-Traffic-Prediction/blob/1c7c302e68276a5c8a61be7bbefca5d36b871ec6/CATP/preprocessing/ProcessRaw.py#L261C2-L261C2
    for implementation for traffic-4-cast dataset
    """
    x_i = large_dataset[:,timestamp: timestamp+i_o]
    y_i = large_dataset[:,timestamp + i_o + 1: timestamp + 2 * i_o + 1]
    return x_i, y_i

def sample_blocks_for_XY(dataset, i_o):
    """
    Converts sequential data into X,Y for time series supervised regression task
    This function ensures that there is enough room in the dataset for the input-output blocks.

    Args:
        dataset (np.ndarray): The dataset from which to sample.
        i_o (int): The size of the input-output blocks.

    Returns:
        tuple: A tuple containing arrays for X, Y, and a list of indices.
    """
    X, Y = [], []
    max_start_index = dataset.shape[1] - 2 * i_o - 1 # Ensure room for i_o at the ends of dataset
    indices_list = []
    for i in range(max_start_index):
        x_block, y_block = dataloader(dataset, timestamp=i, i_o=i_o) # dataset[:, start_index:start_index + i_o]
        X.append(x_block)
        Y.append(y_block)
        indices_list.append(i)
    return np.array(X), np.array(Y), indices_list
```

Build model We use keras with tensorflow backend, the input and output shapes are shown for references

In [ ]:

```
def build_model_fc(i_o):
    # Calculate the total number of elements in the input (e.g., 2*100 for a 2x100 input)
    model = Sequential([
        Flatten(input_shape=((n_sensors, i_o))),
        Dense(64, activation='relu'),
        Dense(n_sensors * i_o),  # Output layer with as many neurons as the total elements in the input
        Reshape((n_sensors, i_o))  # Reshape the output to match the input shape
    ])
    model.compile(optimizer='sgd', loss='mse')
    return model

def linf_distance(a,b):
    """
    Computes the L-infinity distance between two vectors.
    Args:
        a (np.ndarray): The first array.
        b (np.ndarray): The second array.

    Returns:
        float: The L-infinity distance between the two arrays.
    """
    return np.max(np.abs(a.flatten()-b.flatten()))

dummy_model = build_model_fc(i_o = 10)
dummy_model.summary()
```

Model complexity function

To make the model complexity computation efficient (and straightforward to implement), the key implementation steps are shown below:

- **Custom Dataloader**: For each input data point, find its temporal neighbours; this is a constant time operations for time series tasks, since the dataloader can be tweaked to return the input data point at a given `timestamp`.

In [ ]:

```
def dataloader(large_dataset, timestamp, i_o):
    """
    Loads a subset of data from a larger dataset.
    The dataloader should be able to access the neighbours using a timestamp index.

    Args:
        large_dataset (np.ndarray): The larger dataset from which to load data.
        i (int): The starting index for loading data.
        i_o (int): The size of the input-output blocks.

    Returns:
        tuple: A tuple containing the X and Y blocks for the specified index.

    Refer to: https://github.com/mie-lab/Complexity-Aware-Traffic-Prediction/blob/1c7c302e68276a5c8a61be7bbefca5d36b871ec6/CATP/preprocessing/ProcessRaw.py#L261C2-L261C2
    for implementation for traffic-4-cast dataset
    """
    x_i = large_dataset[:,timestamp: timestamp+i_o]
    y_i = large_dataset[:,timestamp + i_o + 1: timestamp + 2 * i_o + 1]
    return x_i, y_i
```

- **Clubbing all `model.predict` together**: To make the Model complexity (MC) efficient, it is recommended to compute all predictions using a dataloader since it is inefficient to call `model.predict` intermittently while doing other processing for neighbourhood search etc.. Once all predictions are ready, we can compute the model complexity by measuring the degree to which the model transforms the input space. As excerpted from the `model_complexity_MC` function below, since here our data set is small, all predictions can be stored in a list. For real datasets, (as was the case in our experiments in the paper), all predictions can be saved to disk and a similar dataloader can be used to extract the relevant predictions later on.

```
# predict for all data points so that we can process later
 predicted = [0] * N
 for i in tqdm(range(0, N, batch_size), "Predicting for all data points"):
   X = []
   for j in range(batch_size):
     x,y = dataloader(large_dataset, j, i_o=i_o)
     X.append(x)
   X = np.array(X)
   predicted[i:i+X.shape[0]] = [predicted for predicted in model_predict(X.reshape((-1, n_sensors, i_o)))]
```

- **Determine the neighbours**: The neighbours in temporal band of look forward and backward policy of `n_h` at multiples of periodicity are searched by:

```
for day in range(-n_d, n_d+1):
   for hour in range(-n_h, n_h+1):
     j = i + day * periodicity + hour
```

From Equation 4 in the paper, we had the set of neighbours for data tensor at time stamp `t` as \mathbb{U}_t, given by:

$$\mathbb{U}_t = \left\{ t' \mid t' \neq t,\, t' = t \pm \underbrace{n_d \cdot 24 \cdot \frac{60}{p}}_{\text{daily periodicity offset}} \pm \underbrace{n_h}_{h \text{ hours look forward \& backward}} \right\}$$

Given our custom dataloader, the input data point at timestamp `j` values can be extracted in constant time as:

```
x_j, y_j = dataloader(large_dataset, j, i_o=i_o)
```

- **Compute the maximum distance in input space** From equations 6 and 7 in the paper, we have:

$$\mathbb{T}_\mathbf{x} = \{\mathbf{x_t} \mid t \in \mathbb{U}_t\} \tag{1}$$

$$r_\mathbf{x} = \max(\{||\mathbf{x_i} - \mathbf{x}||_\infty\} \,|\mathbf{x_i} \in \mathbb{T}_\mathbf{x}) \tag{2}$$

In the function `model_complexity_MC`, the r_x is computed using:

```
.
.  # inside the loop
   neighbour_index_list.append(j)
   x_distance_list.append(linf_distance(x_i, x_j))
max_dist_x = np.max(x_distance_list)
```

- **Criticality of each data point**: For each data point, we track the predictions (output of `model.predict`, which has been pre-computed and saved in the variable `predictions`), to compute the criticality defined in Equation 8 as:

$$CRIT(\mathbf{x}|f,\mathcal{D}) = \sum_{\mathbf{x_j}\in\mathbb{T}_\mathbf{x}} \left(d_{f(\mathbf{x_j})} - r_\mathbf{x}\right) \cdot 1_{d_{f(\mathbf{x_j})}>r_\mathbf{x}} \tag{3}$$

```
# inside loop
   compute_criticality = [0]
   for y_distance in y_distance_list:
     if y_distance > max_dist_x:
```

```
                compute_criticality.append(y_distance)
        criticality = sum(compute_criticality)
        list_of_criticality_values.append(criticality)
```

- MC **as the mean over all n criticality values**:

```
# outside loop
return np.mean(list_of_criticality_values) # computed complexity value of the model
```

Reproducing the Equation 9 from the paper, we have:

$$MC(f|\mathcal{D}) = \frac{1}{N}\sum_{k=1}^{N} CRIT(\mathbf{x}_k|f, \mathcal{D}) \tag{4}$$

In [ ]:

```
def model_complexity_MC(large_dataset,
                i_o,
                n,
                model_predict,
                periodicity,
                n_d=3,
                n_h=2,
                batch_size=32):
    """
    Computes the model complexity metric using a given DL model

    This function is similar to `model_complexity_MC` but uses the ground truth data as the prediction
    from the perfect model. It calculates the intrinsic complexity based on input-output distances.

    Args:
        large_dataset (np.ndarray): The dataset to compute the complexity on.
        i_o (int): The number of frames in input and output.
        n (int): The number of data points to consider in the complexity calculation.
        periodicity refers to the offset required for 1 day;
        n_d=3 corresponds to 1 week of neighbours (3 days look ahead and back; and the current day)
        n_h=2 corresponds to 2 hours of neighbours (1 hour look ahead and back)
        model_predict (function): The prediction function of the model
    """


    list_of_criticality_values = []

    N =  large_dataset.shape[1]

    # predict for all data points so that we can process later
    predicted = [0] * N
    for i in tqdm(range(0, N, batch_size), "Predicting for all data points"):
        X = []
        for j in range(batch_size):
            x,y = dataloader(large_dataset, j, i_o=i_o)
            X.append(x)
        X = np.array(X)

        predicted[i:i+X.shape[0]] = [predicted for predicted in model_predict(X.reshape((-1, n_sensors, i_o)))]


    for i in tqdm(range(i_o, n), "Iterating over all " + str(n) + " data points"):

        # create list of all neighbours in temporal bands
        # at multiples of periodicity
        neighbour_list = []
        x_distance_list = []
        neighbour_index_list = []
        x_i, y_i = dataloader(large_dataset, i, i_o=i_o)

        f_x_i = predicted[i]             # f(x_i)

        for day in range(-n_d, n_d+1):
            for hour in range(-n_h, n_h+1):

                j = i + day * periodicity + hour

                # ignore the tensors which are at the boundaries of the dataset
                if j + i_o < 0 or j < 0 or \
                        j >= large_dataset.shape[1] or j+i_o >= large_dataset.shape[1]:
```

```
                    continue

                if j != i:
                    x_j, y_j = dataloader(large_dataset, j, i_o=i_o)
                    assert (x_j.shape == x_i.shape)
                    neighbour_list.append(x_j)
                    neighbour_index_list.append(j)
                    x_distance_list.append(linf_distance(x_i, x_j))

        max_dist_x = np.max(x_distance_list)

        y_distance_list = []
        for neighbour_index in neighbour_index_list:
            # get f(x)
            f_x_j = predicted[neighbour_index] # model_predict(neighbour.reshape((-1,2,i_o)))
            y_distance_list.append(linf_distance(f_x_j, f_x_i))

        compute_criticality = [0]
        for y_distance in y_distance_list:
            if y_distance > max_dist_x:
                compute_criticality.append(y_distance)

        criticality = sum(compute_criticality)

        list_of_criticality_values.append(criticality)
    return np.mean(list_of_criticality_values)
```

- **The Intrinsic complexity (IC)** is implemented similarly as MC with the only difference that $f_{PM}(x_j) = y_j$ instead of $f(x_j)$ = `model.predict` (x_j). In the following function, this is marked as: `# THE ONLY DIFFERENCE FROM MODEL COMPLEXITY COMPUTATION`

In [ ]:

```
def intrinsic_complexity_IC(large_dataset,
                i_o,
                n,
                model_predict,
                periodicity,
                n_d=3,
                n_h=2,
                batch_size=32):
    """
    Computes the intrinsic complexity metric using ground truth data.

    This function is similar to `model_complexity_MC` but uses the ground truth data as the prediction
    from the perfect model. It calculates the intrinsic complexity based on input-output distances.

    Args:
        large_dataset (np.ndarray): The dataset to compute the complexity on.
        i_o (int): The number of frames in input and output.
        n (int): The number of data points to consider in the complexity calculation.
        periodicity refers to the offset required for 1 day;
        n_d=3 corresponds to 1 week of neighbours (3 days look ahead and back; and the current day)
        n_h=2 corresponds to 2 hours of neighbours (1 hour look ahead and back)
        model_predict (function): The prediction function of the model
    """


    list_of_criticality_values = []

    N =  large_dataset.shape[1]

    # predict for all data points so that we can process later

    predicted = [0] * N
    for i in tqdm(range(0, N, batch_size), "Predicting for all data points"):
        X = []
        Y = []  # only differences from MC function. Here we use the ground truth as the prediction
                # from the perfect model
        for j in range(batch_size):
            x,y = dataloader(large_dataset, j, i_o=i_o)
            X.append(x)
            Y.append(y)
        X = np.array(X)


        # THE ONLY DIFFERENCE FROM MODEL COMPLEXITY COMPUTATION
        predicted[i:i+X.shape[0]] = [predicted for predicted in Y]



    for i in tqdm(range(i_o, n), "Iterating over all " + str(n) + " data points"):

        # create list of all neighbours in temporal bands
        # at multiples of periodicity
        neighbour_list = []
        x_distance_list = []
        neighbour_index_list = []
        x_i, y_i = dataloader(large_dataset, i, i_o=i_o)

        f_x_i = predicted[i]   # f(x_i)

        for day in range(-n_d, n_d+1):
            for hour in range(-n_h, n_h+1):

                j = i + day * periodicity + hour

                # ignore the tensors which are at the boundaries of the dataset
                if j + i_o < 0 or j < 0 or \
                        j >= large_dataset.shape[1] or j+i_o >= large_dataset.shape[1]:
                    continue

                if j != i:
                    x_j, y_j = dataloader(large_dataset, j, i_o=i_o)
                    assert (x_j.shape == x_i.shape)
                    neighbour_list.append(x_j)
                    neighbour_index_list.append(j)
                    x_distance_list.append(linf_distance(x_i, x_j))

        max_dist_x = np.max(x_distance_list)

        y_distance_list = []
        for neighbour_index in neighbour_index_list:
            f_x_j = predicted[neighbour_index] # model_predict(neighbour.reshape((-1,2,i_o)))
            y_distance_list.append(linf_distance(f_x_j, f_x_i))

        compute_criticality = [0]
        for y_distance in y_distance_list:
            if y_distance > max_dist_x:
                compute_criticality.append(y_distance)

        criticality = sum(compute_criticality)

        list_of_criticality_values.append(criticality)
    return np.mean(list_of_criticality_values) # computed complexity value of the model
```

`Vanilla codes for plotting training and visualising predictions of time series.`

In [ ]:

```
def plot_training_curves(model_identifer, history):
    """
    plotting the training and validation loss with time
    """
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]

    epochs = range(1, len(loss) + 1)

    # Plotting
    plt.figure(figsize=(8, 4))
    plt.plot(epochs, loss, 'tab:blue', label='Training loss')
    plt.plot(epochs, val_loss, 'tab:orange', label='Validation loss')
    plt.title('Training Loss' + model_identifer)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.savefig("plots_from_demo_data/training_curve_"+ model_identifer +".jpg", dpi=300)
    plt.show()
    plt.clf()

def plot_selected_predictions_val_data(val_data, i_o, model_predict, model_identifier):
    """
    plotting selected data points from the validation data to show the prediction performance
    """
    indices = [0, 50, 1400]
    ax = [0] * 3
    plt.clf()
    fig, (ax[0], ax[1], ax[2]) = plt.subplots(3)

    for counter, i in enumerate (indices):
        x,y = dataloader(val_data, timestamp=i, i_o=i_o)
        y_predict = model_predict(x.reshape((-1, n_sensors, i_o)))
        ax[counter].plot(x[0, :].flatten().tolist() + y[0, :].flatten().tolist(), label="sensor 1 GT", color="tab:blue",
                         linewidth=2)
        ax[counter].plot(x[1, :].flatten().tolist() + y[1, :].flatten().tolist(), label="sensor 2 GT", color="tab:orange",
                         linewidth=2)

        ax[counter].plot(x[0, :].flatten().tolist() + y_predict[0, 0, :].flatten().tolist(), label="sensor 1 pred", color="tab:blue",
                         linewidth=0.6)
        ax[counter].plot(x[1, :].flatten().tolist() + y_predict[0, 1, :].flatten().tolist(), label="sensor 2 pred", color="tab:orange",
                         linewidth=0.6)
        ax[counter].set_title(model_identifier)

    plt.legend(fontsize=6, loc="upper left")
    # plt.title(model_identifier)
    plt.tight_layout()
    plt.savefig("plots_from_demo_data/Predictions_" + model_identifier + ".jpg", dpi=300)
    plt.show()
    plt.clf()
```

In [ ]:

Driver function ("`__main__`")

In [ ]:

```
large_dataset, PERIODICITY = generate_continuous_dataset()
print (large_dataset.shape)

i_o = 7  # Length of Input and output sequences
EPOCH = 20
n_for_complexity_calculation = 5000

TrainX, TrainY, indices_list = sample_blocks_for_XY(large_dataset[:, :-2000], i_o)
ValX, ValY,_ = sample_blocks_for_XY(large_dataset[:,-2000:], i_o)



model_fc = build_model_fc(i_o=i_o)
model_fc.summary()

history = model_fc.fit(TrainX, TrainY, epochs=EPOCH, verbose=2, validation_data=
                                        [ValX, ValY], batch_size=32)
plot_training_curves(model_identifer="_fc_", history=history)





metric_value_IC = intrinsic_complexity_IC(large_dataset[:, :8000],
                                   i_o,
                                   n_for_complexity_calculation,
                                   model_fc.predict,
                                   periodicity=PERIODICITY,
                                   n_d=2,
                                   n_h=20,
                                   batch_size=32
                                   )


# Compute the custom metric for one example
metric_value_fc = model_complexity_MC(large_dataset[:, :8000],
                                   i_o,
                                   n_for_complexity_calculation,
                                   model_fc.predict,
                                   periodicity=PERIODICITY,
                                   n_d=2,
                                   n_h=20,
                                   batch_size=32
                                   )



predicted_val_Y = model_fc.predict(ValX)
assert (predicted_val_Y.shape == ValY.shape)
print ("MSE: (FC) ",  np.mean( (ValY - predicted_val_Y) ** 2 ))

plot_selected_predictions_val_data(large_dataset[:, -2000:], i_o=i_o, model_predict=model_fc.predict, model_identifier="_FC_")
```

In [ ]:

```
print("Intrinsic Complexity: ", metric_value_IC)
print("Model complexity Fully Connected: ", metric_value_fc)
print ("End-to-end Run time of script: ", round(time.time() - starttime, 2))
```

In [ ]:

In [ ]:

In [ ]:

Same thing for LSTM model

In [ ]:

```
def build_model_lstm(i_o):
    # Calculate the total number of elements in the input (e.g., 2*100 for a 2x100 input)
    output_shape = (n_sensors, i_o)  # Adjust this to your desired output shape
    total_output_elements = np.prod(output_shape)

    model = Sequential([
        LSTM(64, input_shape=(n_sensors, i_o), return_sequences=False),
        Dense(64, activation='relu'),
        Dense(total_output_elements),  # Ensure this matches the total number of elements in the output shape
        Reshape(output_shape)  # Reshape the output to the desired shape
    ])
    model.compile(optimizer='sgd', loss='mse')
    return model

model_lstm = build_model_lstm(i_o=i_o)
model_lstm.summary()
history = model_lstm.fit(TrainX, TrainY, epochs=EPOCH, verbose=2, validation_data=
                                        [ValX, ValY], batch_size=32)
plot_training_curves(model_identifer="_lstm_", history=history)

metric_value_lstm = model_complexity_MC(large_dataset[:, :8000],
                                   i_o,
                                   5000,
                                   model_lstm.predict,
                                   periodicity=PERIODICITY,
                                   n_d=2,
                                   n_h=20,
                                   batch_size = 32
                                    )
predicted_val_Y = model_lstm.predict(ValX)
assert (predicted_val_Y.shape == ValY.shape)
print ("MSE: (LSTM) ",  np.mean( (ValY - predicted_val_Y) ** 2 ))
plot_selected_predictions_val_data(large_dataset[:, -2000:], i_o=i_o, model_predict=model_lstm.predict, model_identifier="_LSTM_")
print("Model complexity LSTM: ", metric_value_lstm)
```

In [ ]: