nesy-retina-report

June 2, 2025

1 Neuro-symbolic integration report

Instructor: Prof. Jędrzej Potoniec

Authors:

- Kacper Dobek 148247
- Daniel Jankowski 148257

This report is a summary of the work done in the course Neuro-symbolic integration at the Poznan University of Technology as an addition our master's thesis *Modeling Retinal Cells with Neural Differential Equations*. We aimed to develop a better understanding of our dataset and use neurosymbolic techniques to make an attempt at transfering the dynamical system dynamics to a symbolic representation.

Bibliography:

- Maheswaranathan, N., McIntosh, L. T., Tanaka, H., Grant, S., Kastner, D. B., Melander, J. B., Nayebi, A., Brezovec, L. E., Wang, J. H., Ganguli, S., and Baccus, S. A. Interpreting the retinal neural code for natural scenes: From computations to neurons. Neuron, 111(17):2742–2755.e4, 2023. ISSN 0896-6273. doi:10.1016/j.neuron.2023.06.007. URL https://www.cell.com/neuron/abstract/S0896-6273(23)00467-1.
- Hasani, R., Lechner, M., Amini, A., Liebenwein, L., Ray, A., Tschaikowski, M., Teschl, G., and Rus, D. Closed-form continuous-time neural net- works. Nature Machine Intelligence, 4(11):992–1003, 2022. ISSN 2522-5839. doi:10.1038/s42256-022-00556-7. URL https://www.nature.com/articles/s42256-022-00556-7.

1.0.1 Import libraries

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
import numpy as np

from h5py import File

sns.set(style="whitegrid", font_scale=1.3)

%matplotlib inline
```

[2]:
```
plot_dir = Path("plots")
plot_dir.mkdir(exist_ok=True)
```

1.0.2 Part 1: Firing rate rise detection EDA

The first part of this report focuses on the exploratory data analysis of firing rate increases. Our objective is to analyze and visualize the data to better understand the mechanisms underlying rises in firing rate.

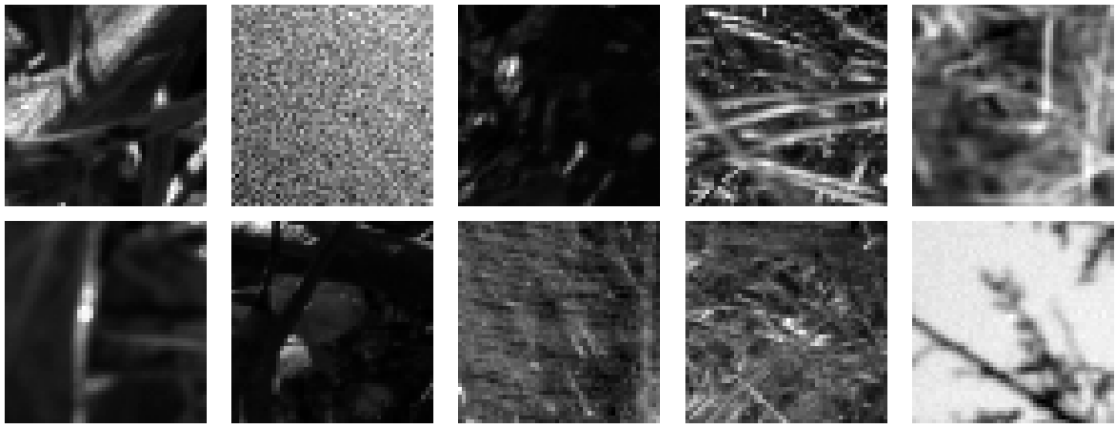
We hypothesize that increases in firing rate are triggered by changes in the input signal, potentially with a temporal delay. The following analysis aims to investigate and validate this assumption.

[21]:
```
def read_h5_to_numpy(file_path: str, subset_type: str, response_type: str):
    """
    Reads data from an HDF5 file and converts it to numpy arrays. Normalizes␣
 ↪the output data if the scaler is provided.
    Returns:
        Tuple[ndarray[Any, dtype[Any]], ndarray[Any, dtype[Any]]]: A tuple␣
 ↪containing the input data (X) and the output data (y).
    """  # noqa: E501
    with File(file_path, "r") as h5file:
        # Read as numpy arrays
        X = np.asarray(h5file[subset_type]["stimulus"])
        y = np.asarray(h5file[subset_type]["response"][response_type])
    X = X.astype("float32") / 255.0
    y = y.astype("float32")

    return X, y
```

[22]:
```
x, y = read_h5_to_numpy(
    "../data/neural_code_data/naturalscene.h5", "train", "firing_rate_10ms"
)
```

Let's show a few examples of input frames

[23]:
```
np.random.seed(42)
random_indices = np.random.choice(x.shape[0], 10, replace=False)
x_sample = x[random_indices]
fig, axes = plt.subplots(2, 5, figsize=(15, 6))
for i, ax in enumerate(axes.flat):
    ax.imshow(x_sample[i], cmap="gray")
    ax.axis("off")
plt.tight_layout()
plt.show()
```

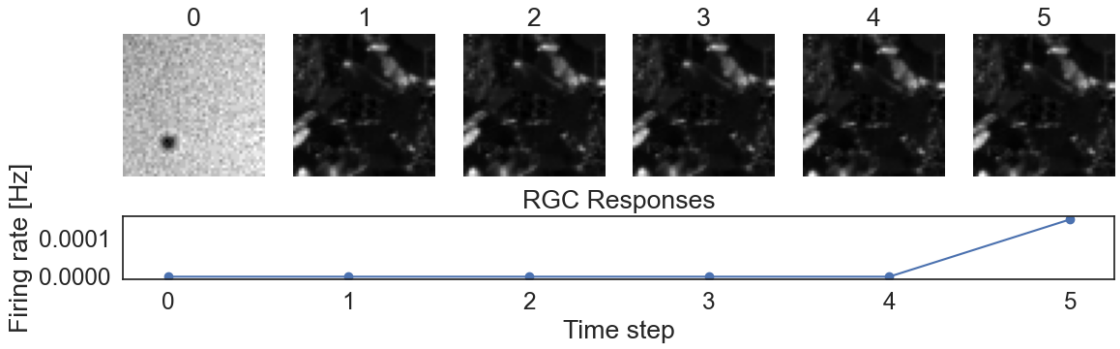


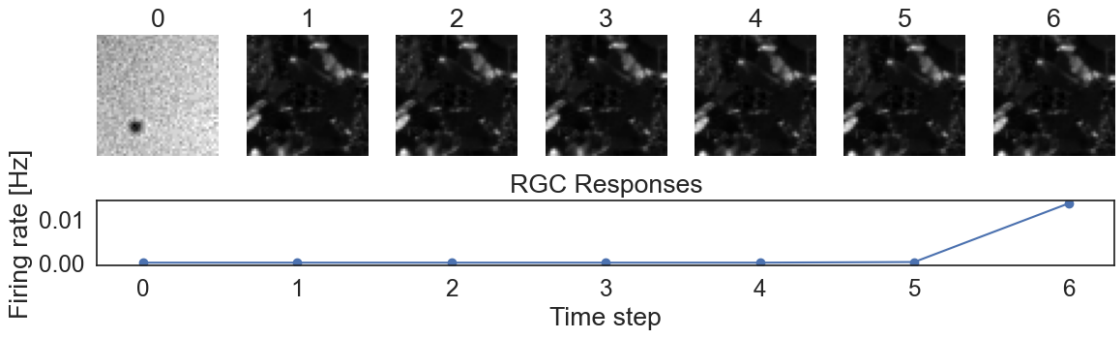
The function below plots the last N images along with response values below them when a rising firing rate is detected.

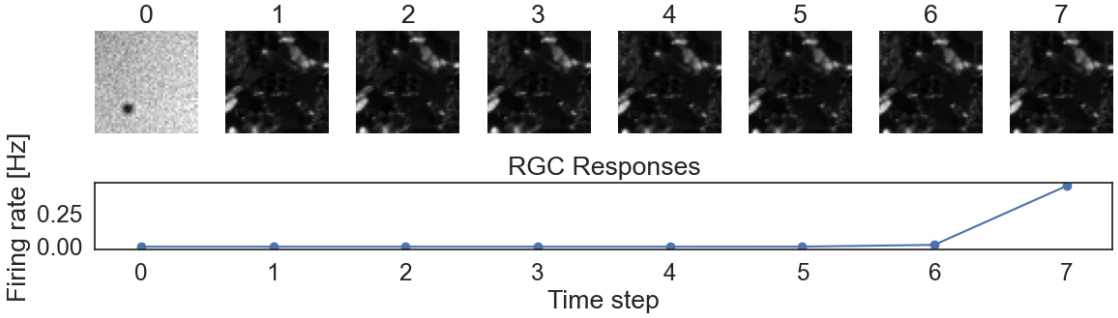
[24]:
```
def plot_spike_images(
    responses: np.ndarray, images: np.ndarray, N: int = 5, save=False
):
    """
    Plots the last N images when a rising firing rate (response > 0) is␣
↪detected and also plots the responses below them.

    Parameters:
    - responses: 1D numpy array of neural responses.
    - images: 3D numpy array of shape (num_samples, 50, 50) containing␣
↪grayscale images.
    - N: Number of images to plot before the detected spike.
    """
    spike_indices = np.where(responses > 0)[0]  # Indices where response > 0

    if len(spike_indices) == 0:
        print("No spikes detected.")
        return

    for spike_idx in spike_indices:
        start_idx = max(0, spike_idx - N)  # Ensure valid index range
        selected_indices = np.arange(start_idx, spike_idx + 1)
        selected_responses = responses[selected_indices]

        fig = plt.figure(figsize=(12, 4))
        gs = fig.add_gridspec(2, spike_idx - start_idx + 1, height_ratios=[3,␣
↪1])
        # fig.suptitle(f"Spike detected: {responses[spike_idx]:.2f} at index␣
↪{spike_idx}")
```

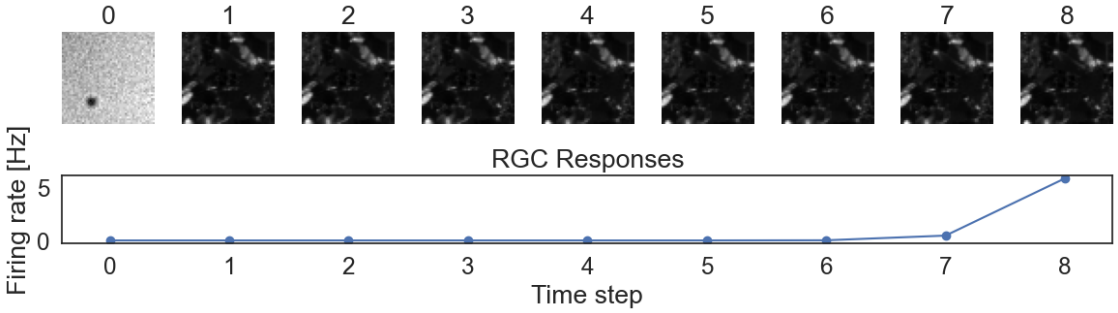
```
        for i, idx in enumerate(selected_indices):
            ax = fig.add_subplot(gs[0, i])
            ax.imshow(images[idx], cmap="gray")
            ax.axis("off")
            ax.set_title(str(idx))

        ax_response = fig.add_subplot(gs[1, :])
        ax_response.plot(
            selected_indices, selected_responses, marker="o", linestyle="-",␣
↪color="b"
        )
        ax_response.set_title("RGC Responses")
        ax_response.set_xlabel("Time step")
        ax_response.set_ylabel("Firing rate [Hz]")

        plt.tight_layout()
        plt.show()
        if save:
            fig.savefig(f"plots/spike_images_{spike_idx}.pdf")
```

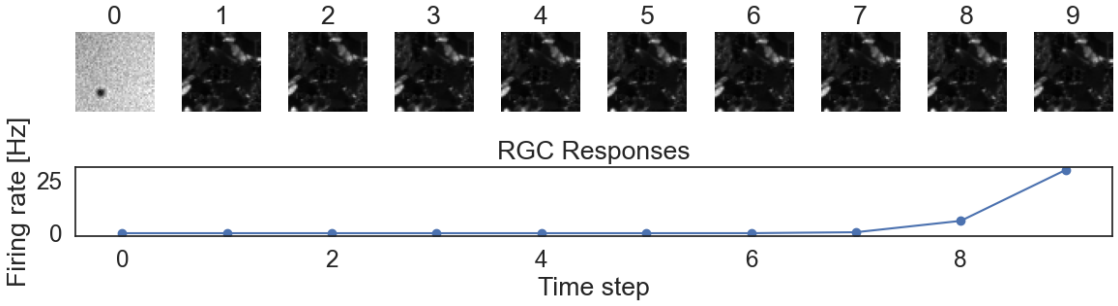
[25]:
```
y_0 = y[0]  # Select the first channel

plot_spike_images(y_0[809:820], x[809:820], N=10, save=False)
```

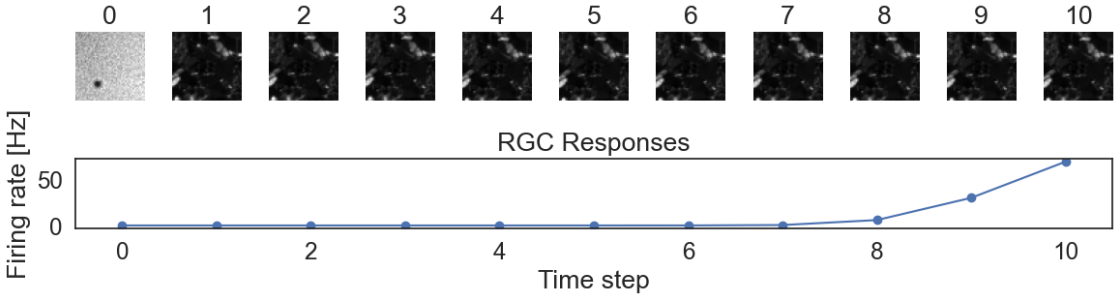





0 1 2 3 4 5 6 7
RGC Responses
Firing rate [Hz]
0.25
0.00
0 1 2 3 4 5 6 7
Time step


0 1 2 3 4 5 6 7 8
RGC Responses
Firing rate [Hz]
5
0
0 1 2 3 4 5 6 7 8
Time step


0 1 2 3 4 5 6 7 8 9
RGC Responses
Firing rate [Hz]
25
0
0 2 4 6 8
Time step


0 1 2 3 4 5 6 7 8 9 10
RGC Responses
Firing rate [Hz]
50
0
0 2 4 6 8 10
Time step

Observe the frame at time step 0 (left), which has distinct visual characteristics as compared to the following frames. For time step 10, we can see a very high firing rate, suggesting a lag between the change of the stimuli and the retinal ganglion cell (RGC) signal.

The dataset consists of recordings from a real biological nervous system. As such, it inherently contains biological variability and noise, both from the neural activity itself and the recording process. The firings are sparse, with approximately 80% of the response values being equal to zero. When a high-frequency firing occurs, it is typically brief, with a short onset followed by a rapid return to a low value. Such events are most likely present shortly after changes in the visual scene, although they appear with a characteristic temporal lag - generally about $8-12$ frames later from the moment when the scene changes.

1.0.3 Part 2: Approximation of CfC dynamics using PySINDy

In the second part of the project, we aim to express the dynamics of the CfC component using symbolic equations. We attach the `extract_equation.py` script to this report. The script was used to extract the equation of the system using the PySINDy Python package.

This code implements a pipeline for extracting and analyzing interpretable dynamical equations from the hidden states of the neural network we designed, composed of a convolutional encoder and a Closed-form Continuous-time network (CfC). The main goal is to use the PySINDy (Sparse Identification of Nonlinear Dynamics) framework to approximate the evolution of the model's hidden states with explicit, human-readable differential equations.

The workflow begins by loading a trained neural network and dataset, then pruning the CfC model to reduce its complexity (global pruning of the 20 lowest weights according to the L1 norm). The code collects hidden state trajectories from the CfC component as the model processes the training data. We fit select only $20,000$ data points from the training set to reduce the computational burden. These trajectories are used to fit a SINDy model, which identifies a sparse set of nonlinear differential equations that best describe the hidden state dynamics. The identified equations are saved in a txt file.

The code also provides tools for simulating the hidden state evolution using the learned SINDy equations, starting from an initial state derived from the encoder. We save the predicted responses and the targets to compare the model's performance against the original neural network. Additionally, the script supports batch processing of multiple models and parallel evaluation of SINDy equation (using `joblib`), enhancing efficiency.

Despite the attempts to simplify the equations by removing terms with small coefficients, the resulting equations remain complex and contain many terms. Below, we present the extracted equation for the first output variable, `h0`. The equation is a polynomial of degree 3 (including interactions with other variables). Although other libraries such as Fourier are available in PySINDy, we decided to use the polynomial library, as it showed the best results in our experiments.

$$(h_0)' = 0.5341 + 10.377h_0 + 13.713h_1 + -24.688h_2 + -5.088h_3 + 15.654h_4 + -14.791h_5 + 23.970h_6 + -23.659h_7 + 24.527h_8 + -107.420h_0^2 + -102.237h_0h_1 + 504.749h_0h_2 + -51.477h_0h_3 + 122.794h_0h_4 + -96.619h_0h_5 + -4.978h_0h_6 + -89.888h_0h_7 + -212.798h_0h_8 + -1571.400h_1^2 + 1292.621h_1h_2 + 194.513h_1h_3 + 364.650h_1h_4 + -204.997h_1h_5 + 102.619h_1h_6 + 1472.792h_1h_7 + 3387.343h_1h_8 +$$

$$-1111.397h_2^2 + -53.275h_2h_3 + -133.208h_2h_4 + 510.764h_2h_5 + 477.261h_2h_6 + -357.330h_2h_7 + -597.828h_2h_8 + 37.741h_3^2 + -142.205h_3h_4 + -11.095h_3h_5 + -216.188h_3h_6 + -73.328h_3h_7 + -533.328h_3h_8 + 90.157h_4^2 + -93.648h_4h_5 + 245.726h_4h_6 + -287.923h_4h_7 + 109.456h_4h_8 + 69.291h_5^2 + -38.328h_5h_6 + -39.898h_5h_7 + 378.854h_5h_8 + 94.725h_6^2 + -141.611h_6h_7 + 225.512h_6h_8 + -208.231h_7^2 + 48.876h_7h_8 + 30.757h_8^2 + -121.549h_0^3 + 1833.684h_0^2h_1 + 606.277h_0^2h_2 + -204.520h_0^2h_3 + -966.325h_0^2h_4 + 69.220h_0^2h_5 + -1239.522h_0^2h_6 + -13.777h_0^2h_7 + -1306.480h_0^2h_8 + -5743.414h_0h_1^2 + -3870.399h_0h_1h_2 + 3713.875h_0h_1h_3 + -527.637h_0h_1h_4 + 145.666h_0h_1h_5 + 3429.980h_0h_1h_6 + 3740.344h_0h_1h_7 + 8803.036h_0h_1h_8 + -1294.063h_0h_2^2 + -1217.918h_0h_2h_3 + 4715.536h_0h_2h_4 + 1497.172h_0h_2h_5 + 5749.602h_0h_2h_6 + -623.769h_0h_2h_7 + 6614.306h_0h_2h_8 + 19.499h_0h_3^2 + -616.337h_0h_3h_4 + 313.868h_0h_3h_5 + -1337.583h_0h_3h_6 + -852.023h_0h_3h_7 + -3389.707h_0h_3h_8 + 545.026h_0h_4^2 + -514.091h_0h_4h_5 + 812.314h_0h_4h_6 + -962.669h_0h_4h_7 + 289.901h_0h_4h_8 + 33.230h_0h_5^2 + -14.329h_0h_5h_6 + -170.486h_0h_5h_7 + 454.253h_0h_5h_8 + 360.520h_0h_6^2 + -1485.576h_0h_6h_7 + 2593.290h_0h_6h_8 + -338.053h_0h_7^2 + -1433.292h_0h_7h_8 + 6771.100h_0h_8^2 + 15111.141h_1^3 + 18807.164h_1^2h_2 + -3148.564h_1^2h_3 + -13978.811h_1^2h_4 + -4717.872h_1^2h_5 + -22936.164h_1^2h_6 + -4289.414h_1^2h_7 + -42459.758h_1^2h_8 + -10636.998h_1h_2^2 + -1196.713h_1h_2h_3 + -752.727h_1h_2h_4 + 9830.820h_1h_2h_5 + -5501.971h_1h_2h_6 + -8853.610h_1h_2h_7 + -39207.492h_1h_2h_8 + -1312.913h_1h_3^2 + 2756.687h_1h_3h_4 + 1408.110h_1h_3h_5 + 5164.589h_1h_3h_6 + -4.442h_1h_3h_7 + 16534.850h_1h_3h_8 + 2293.476h_1h_4^2 + -1362.836h_1h_4h_5 + 2296.949h_1h_4h_6 + 10620.373h_1h_4h_7 + 12017.040h_1h_4h_8 + -2317.449h_1h_5^2 + -3472.120h_1h_5h_6 + -932.895h_1h_5h_7 + 115.146h_1h_5h_8 + -1485.884h_1h_6^2 + 13482.284h_1h_6h_7 + -1440.980h_1h_6h_8 + -1471.188h_1h_7^2 + 15148.195h_1h_7h_8 + -49971.086h_1h_8^2 + 4388.114h_2^3 + 752.592h_2^2h_3 + -3764.700h_2^2h_4 + -8714.616h_2^2h_5 + -5782.956h_2^2h_6 + 535.907h_2^2h_7 + 8960.903h_2^2h_8 + -66.942h_2h_3^2 + 853.181h_2h_3h_4 + -1186.433h_2h_3h_5 + 3094.247h_2h_3h_6 + 932.541h_2h_3h_7 + 7439.275h_2h_3h_8 + -680.857h_2h_4^2 + 1463.542h_2h_4h_5 + 491.968h_2h_4h_6 + 477.755h_2h_4h_7 + -3711.868h_2h_4h_8 + 1176.946h_2h_5^2 + 1288.895h_2h_5h_6 + 654.283h_2h_5h_7 + -7210.557h_2h_5h_8 + 1128.432h_2h_6^2 + 2439.257h_2h_6h_7 + -8914.517h_2h_6h_8 + 1423.539h_2h_7^2 + 6102.950h_2h_7h_8 + 1808.321h_2h_8^2 + 97.006h_3^3 + 116.993h_3^2h_4 + -24.957h_3^2h_5 + -227.803h_3^2h_6 + 356.810h_3^2h_7 + -1863.301h_3^2h_8 + -771.572h_3h_4^2 + -56.268h_3h_4h_5 + -1627.748h_3h_4h_6 + -979.536h_3h_4h_7 + -1532.225h_3h_4h_8 + 190.431h_3h_5^2 + 525.477h_3h_5h_6 + -158.533h_3h_5h_7 + 53.461h_3h_5h_8 + -585.906h_3h_6^2 + -1501.524h_3h_6h_7 + 1628.434h_3h_6h_8 + 201.707h_3h_7^2 + -3795.491h_3h_7h_8 + 10483.943h_3h_8^2 + 145.686h_4^3 + 205.228h_4^2h_5 + 627.173h_4^2h_6 + -1188.145h_4^2h_7 + 122.490h_4^2h_8 + 62.928h_4h_5^2 + 864.804h_4h_5h_6 + 328.773h_4h_5h_7 + 2673.697h_4h_5h_8 + 663.552h_4h_6^2 + -1798.931h_4h_6h_7 + 1118.008h_4h_6h_8 + -1432.640h_4h_7^2 + -306.694h_4h_7h_8 + -794.410h_4h_8^2 + 55.647h_5^3 + -560.710h_5^2h_6 + 844.252h_5^2h_7 + -1741.266h_5^2h_8 + 544.898h_5h_6^2 + 581.506h_5h_6h_7 + 469.582h_5h_6h_8 + 255.944h_5h_7^2 + 1335.999h_5h_7h_8 + -2620.509h_5h_8^2 + 249.998h_6^3 + -372.204h_6^2h_7 + 795.299h_6^2h_8 + -1899.068h_6h_7^2 + 1081.207h_6h_7h_8 + -2238.253h_6h_8^2 + 133.637h_7^3 + -915.008h_7^2h_8 + 3398.197h_7h_8^2 + -6991.128h_8^3$$

We will now load predicted responses and targets to compare the model's performance against the original neural network on the validation set.

[13]:
```
pred_dir = Path("../sindy/compare_models")
pred_files = pred_dir.glob("*.csv")

# Create a dictionary to store the dataframes
pred_dict = {}

for pred_file in pred_files:
    # Read the csv file
    df = pd.read_csv(pred_file)
```

```
    # Get the model name
    model_name = pred_file.stem
    # Store the dataframe in the dictionary
    pred_dict[model_name] = df

print(pred_dict.keys())
```

```
dict_keys(['CfC', 'CfC-pruned', 'Sindy-pruned', 'Sindy', 'Target'])
```

[14]:
```
pred_dict["Target"].head()
```

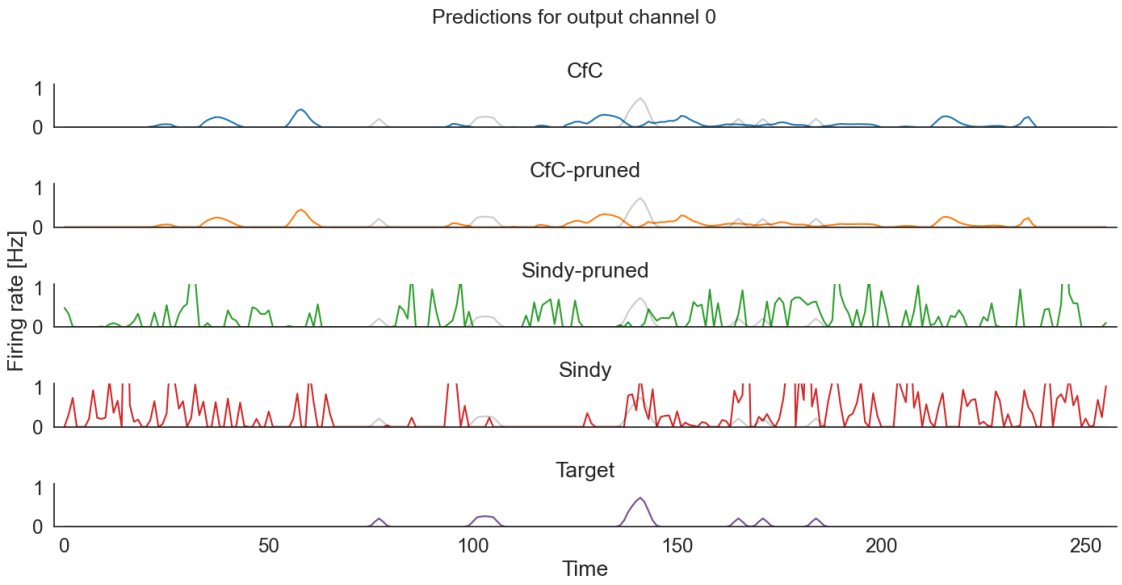
[14]:

	0	1	2	3	4	5	6	7	8
0	0.0	0.0	0.0	0.201739	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.121669	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.026943	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.002191	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.000065	0.0	0.0	0.0	0.0	0.0

[15]:
```
custom_colors = sns.color_palette("tab10")
```

[27]:
```
# Set the style
sns.set_context("paper")
# Set font_scale according to needs
sns.set(style="white", font_scale=1.6)

# Create a figure and axis
n_subplots = len(pred_dict)
fig, axes = plt.subplots(
    n_subplots,
    1,
    figsize=(
        14,
        1.5 * n_subplots,
    ),
    sharey=True,
)

# Select output channel
output_channel = "0"

# Iterate over the dictionary
for i, (model_name, df) in enumerate(pred_dict.items()):
    # Create a scatter plot
    x_axis_values = df[output_channel].index
    sns.lineplot(
        x=x_axis_values, y=output_channel, data=df, ax=axes[i],␣
 ↪color=custom_colors[i]
    )
```

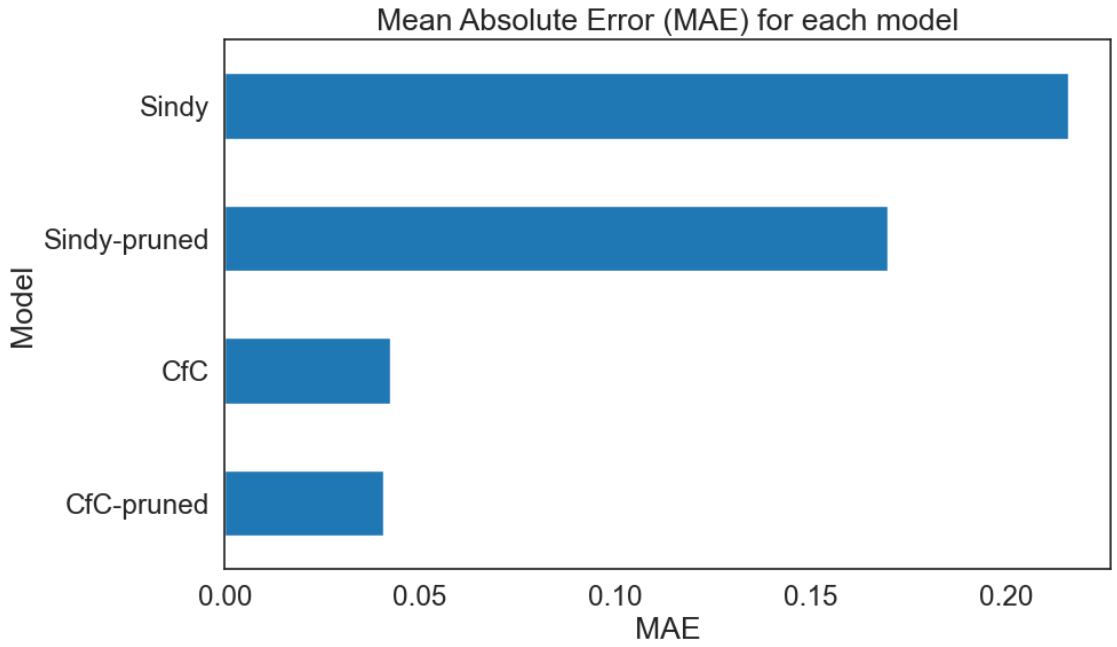
```
    # Plot the target values in background
    # Adjust data frame length
    target_df = pred_dict["Target"].iloc[: len(df)]
    sns.lineplot(
        x=target_df.index,
        y=output_channel,
        data=target_df,
        ax=axes[i],
        color="black",
        alpha=0.2,
    )
    # Add a title
    axes[i].set_title(model_name)
    axes[i].get_xaxis().set_visible(False)
    axes[i].set_ylabel("")
    axes[i].margins(x=0.01)
    axes[i].set_ylim(0, 1.1)

# Set x axis label for the last subplot
axes[-1].get_xaxis().set_visible(True)
axes[-1].set_xlabel("Time")

# Set y axis label for the figure
fig.text(-0.001, 0.5, "Firing rate [Hz]", va="center", rotation="vertical")

fig.suptitle(f"Predictions for output channel {output_channel}", fontsize=18)

sns.despine()
# Adjust the layout
plt.tight_layout()
plt.show()
```



As we can see in the plot above, the sindy prediction are noisy and do not match the target values well. The model is not able to capture the dynamics of the system, which is likely due to the complexity of the system and the limitations of the PySINDy library.

[17]:
```
# Calculate mae for each model
mae_dict = {}
for model_name, df in pred_dict.items():
    if model_name == "Target":
        continue
    # Calculate the mean absolute error
    mae = np.mean(np.abs(df - pred_dict["Target"]))
    mae_dict[model_name] = mae

# Create a DataFrame from the mae_dict
mae_df = pd.DataFrame.from_dict(mae_dict, orient="index", columns=["MAE"])
# Sort the DataFrame by MAE
mae_df = mae_df.sort_values(by="MAE", ascending=True)
```

[20]:
```
mae_df.plot(
    kind="barh",
    figsize=(10, 6),
    color=custom_colors[: len(mae_df)],
    legend=False,
    title="Mean Absolute Error (MAE) for each model",
    xlabel="MAE",
    ylabel="Model",
)
plt.show()
```



The plot above shows that the MAE of the pruned CfC model is close to the MAE of the original CfC model, which means that the pruning did not significantly affect the performance of the model. However, the MAE of the SINDy models based both on the original CfC and the pruned CfC models is significantly higher than the MAE of the CfC models. This indicates that the SINDy models are not able to capture the dynamics of the dynamical system well.

The failure of the PySINDy framework to accurately reconstruct the CfC model's dynamics can be attributed to several key factors. First, the dataset is characterized by high sparsity and biological noise, which severely impacts the reliability of derivative estimation - an essential step in the SINDy pipeline. When most values are zero and high-frequency firing events are brief and irregular, numerical differentiation becomes highly unstable, leading to inaccurate input for the symbolic regression. Second, the CfC architecture likely exhibits complex, high-dimensional, and non-smooth dynamics that are not easily captured by the low-order, interpretable equations SINDy aims to extract. Importantly, even the CfC model, which is designed to handle continuous-time, non-stationary data, struggles to fully model this challenging dynamical system, underscoring the inherent difficulty of the task. Given that SINDy attempts to approximate such dynamics with sparse and simplified differential equations, the problem becomes even more intractable. Third, the training data used for SINDy consisted of only $20,000$ samples. This limited and unbalanced sampling fails to represent the full dynamical range of the system, especially the critical high-activity responses following stimulus changes, thus restricting SINDy's ability to learn expressive and generalizable dynamics.