2143062-DDPM-Part2

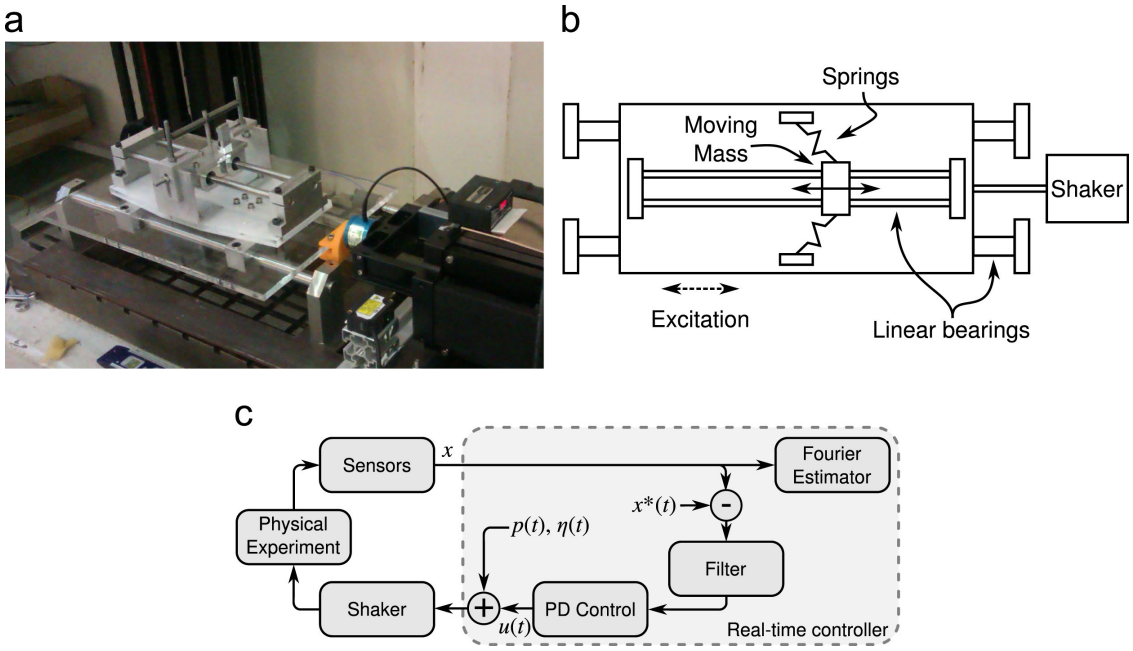
November 28, 2024

1 Data-Driven Physical Modelling (SEMTM0007) Coursework

1.0.1 Wishawin Lertnawapan 2143062

1.1 Table of Contents

1.2 0. Problem Description



> A non-linear mass-spring-damper system mounted on a shaking table. Displacements are measured using a laser displacement sensor and force is measure using a load cell.

System Identification:

The following report explores several artifical neural network (ANN) architectures in training a data-driven physical model. Data is derived from experimental oscillations of a mass-spring-damper

(NTMD) system [1], consisting of a mass supported by low-friction linear bearings, and notably two springs mounted perpendicular to the direction of motion. The mass itself is excited by a linear shaker, creating input excitations through a load cell, creating displacements to a linearly constrained mass.

The significance of the two springs is their introduction of nonlinearity, as opposed to linear mass-spring-damper systems where the spring restoring force aligns with the axis of applied motion. This behaviour is characterised to be geometric, resulting in hardening spring behaviour[2].

This nonlinear stiffness characteristic is to the justified use of ANNs. Due to both intrinsic material hardening and geometric complexities within the restoring force, parameters become nonlinear and thus cannot be represented analytically through conventional regression techniques or classical formulations. To illustrate, the restoring force of a conventional spring in a linear arrangement can be modelled using Hooke's Law [3]

$$F(x) = kx$$

where k is typically a proportionality constant, relating the restoring force F and the spring displacement x. In a perpendicular arrangement, the formulation extends to

$$F(x) = \hat{k}x\left(\frac{x}{s}\hat{i} + \frac{d}{s}\hat{j}\right)$$

where $\hat{k}$ is the nonlinear spring constant, and the terms within the bracket represent directional cosines of the spring force along the $\hat{i}$ and $\hat{j}$ bases respectively. This complexity limits techniques such as delay embedding as these rely on fundamentally linear fitting principles.

Measurement data is noted to be downsampled to 100 Hz, leading to measurement data being a discrete-time representation of a continuous dynamical system. The number of time series recorded is $n = 108$, each with a measurement size of $k = 2000$ sample data points which capture the dynamics. These measurement states can be complied as two matrices, $U(k)$, and $X(k)$ corresponding to the exication inputs and corresponding relative displacement of the mass respectively, for a given time step k.

Output Displacements (Laser Displacement Sensors):

$$X = \begin{bmatrix} x(0)_1 & x(\Delta t)_1 & x(2\Delta t)_1 & \dots & x(k\Delta t)_1 \\ x(0)_2 & x(\Delta t)_2 & x(2\Delta t)_2 & \dots & x(k\Delta t)_2 \\ & & \vdots & & \\ x(0)_n & x(\Delta t)_n & x(2\Delta t)_n & \dots & x(k\Delta t)_n \end{bmatrix}$$

Input Force (Load Cell)

$$U = \begin{bmatrix} u(0)_1 & u(\Delta t)_1 & u(2\Delta t)_1 & \dots & u(k\Delta t)_1 \\ u(0)_2 & u(\Delta t)_2 & u(2\Delta t)_2 & \dots & u(k\Delta t)_2 \\ & & \vdots & & \\ u(0)_n & u(\Delta t)_n & u(2\Delta t)_n & \dots & u(k\Delta t)_n \end{bmatrix}$$

The following ANN architectures were selected, as these architectures are particularly well suited for capturing non-autonomous temporal behavior. ESNs and RNNs retain memory of previous states with feedback mechanisms which allow periodic motion information to be captured. The major

limitation of applying ANNs in the context of the following problem is the relatively low quantity of independent time series, especially as this number decreases with splitting procedures. Assuming a split of 0.7, only 76 time series are available for training. Insufficient data can lead to poor model generalisation [4] and inability to capture nonlinear dynamics which may exhibit large variance from chaotic behaviour. On the testing set, this results in a small number of surrogates available for confident validation.

With this consideration, hyperparameters within the architectures are chosen to mitigate the risk of over-parameterisation and overfitting. Regularisation techniques and several cross-validation techniques have been implemented in the following sections to address this concern.

[1358]:
```
#Importing packages and libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import scipy.integrate as spi
from scipy.integrate import solve_ivp
from scipy.stats import spearmanr

from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as torchdata

from itertools import product

import optuna
from optuna.visualization import plot_param_importances, plot_slice, plot_contour
optuna.logging.set_verbosity(optuna.logging.ERROR)
```

1.3 1. Exploratory Data Analysis

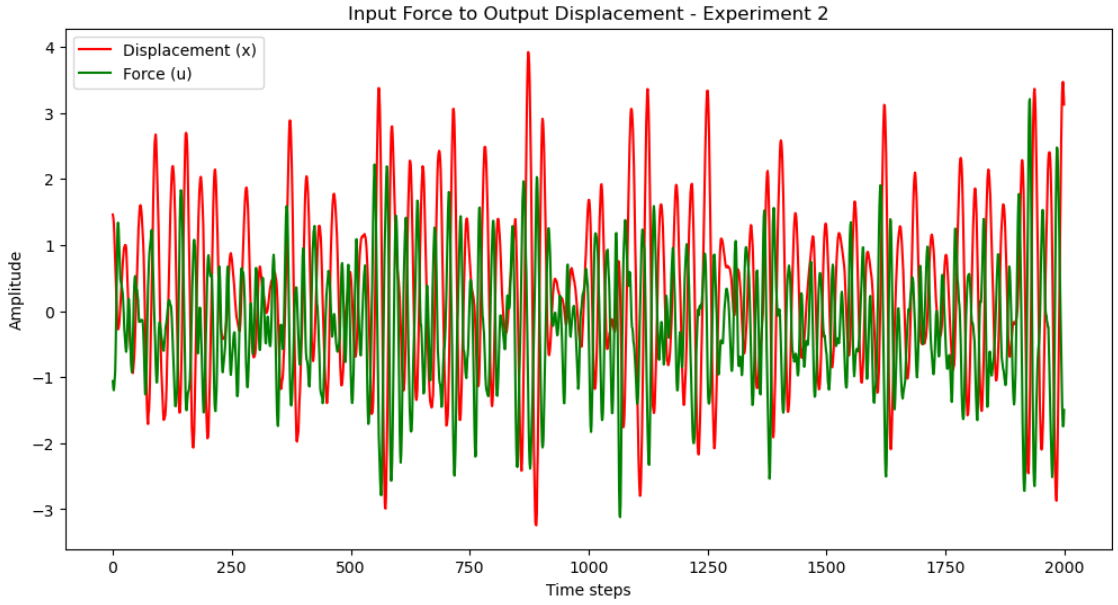
1.3.1 1.0 Time Series Plots

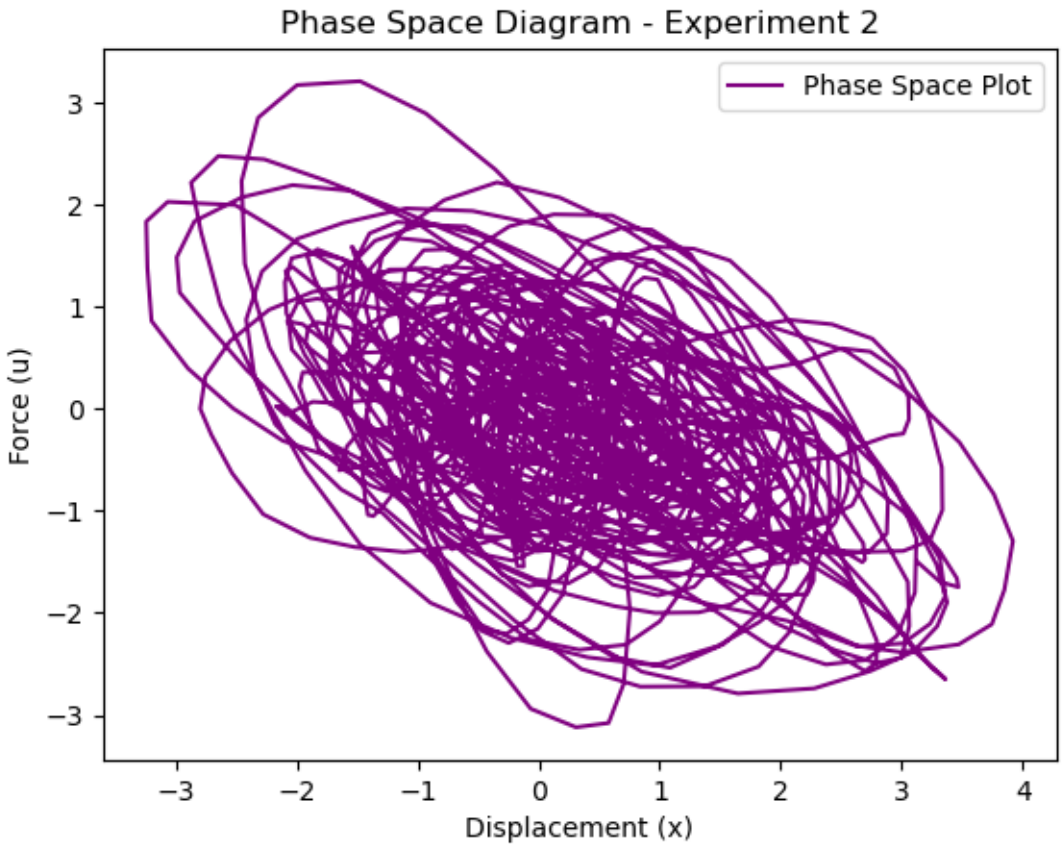
Importing the data for visualisation

[991]:
```
#Loading aata
data = np.load('coursework-2024+25-part2.npz')
x = data['x']
u = data['u']
```

[993]:
```
#-------------------------#
exp = 1 #Experiment
```

```
#--------------------------#

# Visualise data as time series plot
plt.figure(figsize=(12, 6))
plt.plot(np.arange(x.shape[1]), x[exp], label="Displacement (x)",color = "red")
plt.plot(np.arange(u.shape[1]), u[exp], label="Force (u)",color = "green")
plt.xlabel("Time steps")
plt.ylabel("Amplitude")
plt.title(f"Input Force to Output Displacement - Experiment {exp+1}")
plt.legend()
plt.show()

# Visualise data as phase space plot
plt.plot(x[exp], u[exp], label="Phase Space Plot", color='purple')
plt.xlabel("Displacement (x)")
plt.ylabel("Force (u)")
plt.title(f"Phase Space Diagram - Experiment {exp+1}")
plt.legend()
plt.show()
```





Time Series Graph:
The time-series graph exhibits oscillatory behaviour, with a clear correlation between inputs u and outputs x, and a noticeable lag between the input and output due to delayed energy dissipation. These are expected characteristics of a mass-spring-damper system. However, the system's nonlinearities are visible and manifest in the form of amplitude modulation variations.

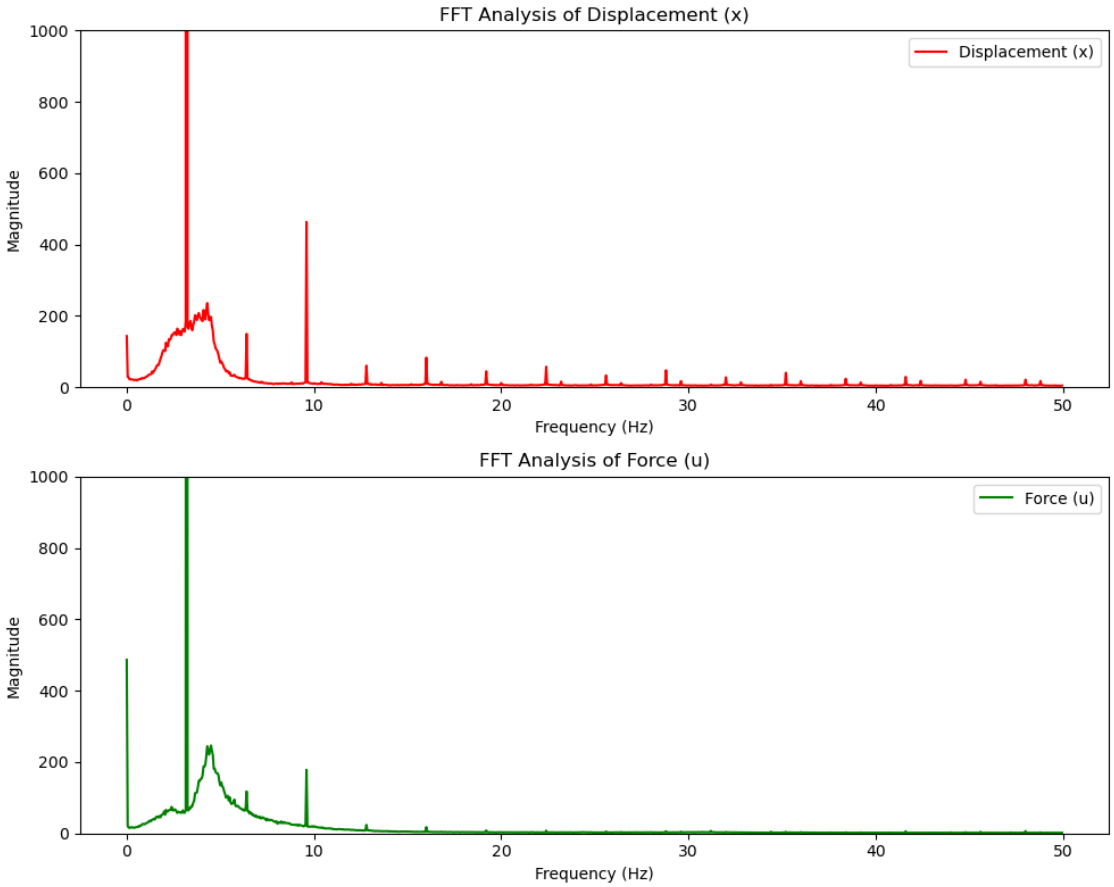
Phase Space Graph:
Phase space highlights the nonlinear and chaotic characteristics of the system, as the elliptic shapes are not fixed, and do not vary in a predictable pattern. The orientation of the semi-major axis further illustrates the phase difference between the two time series. Regions of clustering may indicate the recurrent underlying dynamics, while large variations may be due to the geometric nonlinear behaviour influenced by the spring.

Oscillations also appear affected by the presence of experimental noise, with irregularities from multi-frequency dynamics such as nonlinear energy transfer between modes through coupling. A spectral analysis was performed across all 108 tests to validate and characterise the frequencies.

[996]:
```
#Frequency Analysis function
def fft_analysis(series):
    n = len(series)
    freq = np.fft.fftfreq(n, d=1/100)
```

```
    fft_values = np.fft.fft(series)
    magnitude = np.abs(fft_values)
    return freq[:n//2], magnitude[:n//2]

x_magnitudes = []
u_magnitudes = []
# Average FFT across all time series
for series in x:
    _, magnitude = fft_analysis(series)
    x_magnitudes.append(magnitude)

for series in u:
    _, magnitude = fft_analysis(series)
    u_magnitudes.append(magnitude)

avg_x_magnitude = np.mean(x_magnitudes, axis=0)
avg_u_magnitude = np.mean(u_magnitudes, axis=0)

freq = fft_analysis(x[0])[0]
```

[998]:
```
# Visualising FFT analysis
plt.figure(figsize=(10, 8))

# Subplot x FFT
plt.subplot(2, 1, 1)
plt.plot(freq, avg_x_magnitude, label="Displacement (x)", color="red")
plt.title("FFT Analysis of Displacement (x)")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.ylim(0,1000)
plt.legend()

# Subplot x FFT
plt.subplot(2, 1, 2)
plt.plot(freq, avg_u_magnitude, label="Force (u)", color="green")
plt.title("FFT Analysis of Force (u)")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.ylim(0,1000)
plt.legend()

plt.tight_layout()
plt.show()
```



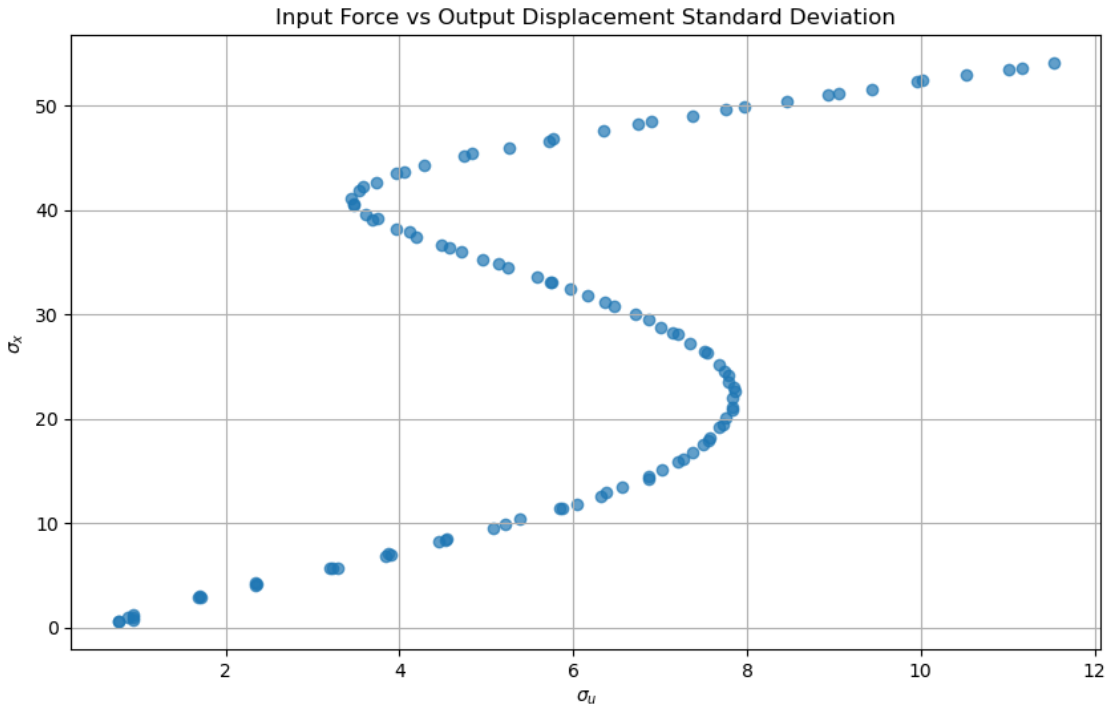
Frequency Analysis `fft_analysis()`

An FFT plot was performed for both input and displacement across all time series before averaging. The key observation is the large peak exhibited in both graphs at 4 Hz which likely corresponds to the dominant natural frequency of oscillation. Interestingly, both graphs show smaller pronounced peaks and an elevated low-frequency response band. The prior phenomena are known as harmonic peaks [5] occuring at multiples of the fundamental frequency and possibly are a consequence of nonlinearities within the system. The latter indicates energy is distributed over a range near the dominant frequency, possibly caused by damping effects within the NTMD. The presence of noise is largely negligible, which justify the lack of filtering preprocessing procedures.

1.3.2 1.1 Standard Deviation

The following step calculates the standard deviation of amplitudes for both input and output matrices. In other terms, for each experiment time series, a measure of its variation across its signal length is computed. This was then plotted to identify the relationship of each signal's amplitude, more specifically the correlation between the system's inputs σ_u, and its direct effect on the response σ_x.

[1003]:
```
x_original = x
u_original = u

# Standard deviation
std_x = np.std(x, axis=1)
std_u = np.std(u, axis=1)

# Visualise standard deviations of inputs vs outputs
plt.figure(figsize=(10, 6))
plt.scatter(std_u, std_x, alpha=0.7)
plt.xlabel("$\\sigma_u$")
plt.ylabel("$\\sigma_x$")
plt.title("Input Force vs Output Displacement Standard Deviation")
plt.grid(True)
plt.show()
```



Standard Deviation Analysis

From the visualisation, a positive correlation is shown, which is expected for coupled dynamics for which higher variability in the input force correlates to outputs with higher variablility. However, the data points do not all follow a strict line, rather an S-shaped curve indicating some level of variation in how the system responds to similar force amplitudes. This could be due to nonlinear effects in the mass-spring-damper system, where other factors influence displacement amplitude.

In context of the physical system, there can be several reasons for this shape. For small values of

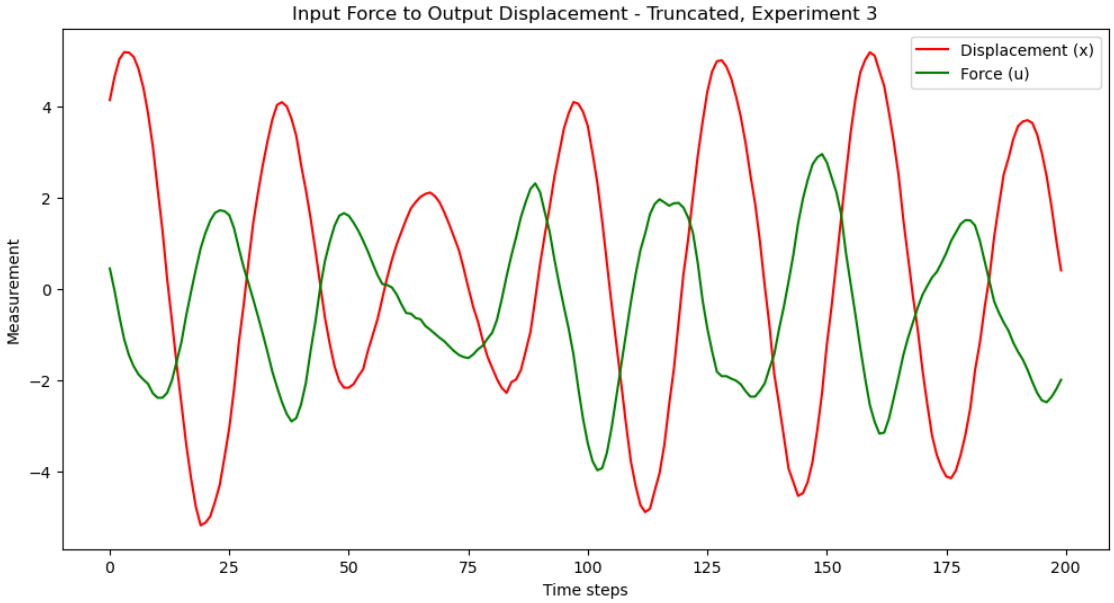
$\sigma_u < 20$, σ_x is less sensitive, which suggest nonlinear damping effects limiting variability. The same effect is also experienced at higher values where $\sigma_u > 40$, showing that the systems response is likely constrained, or limited by the stiffness of the spring being saturated at higher forces. Overall, the plot communicates clear justification for using higher complexity models in subsequent sections.

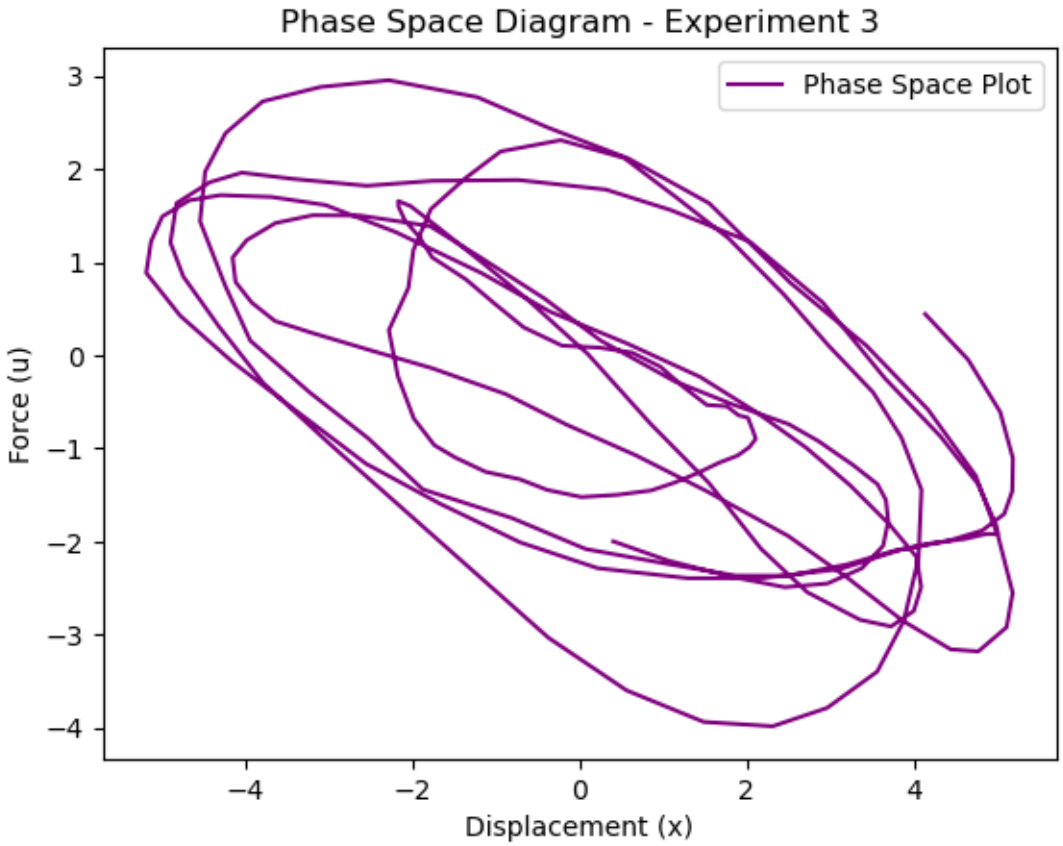
1.3.3 1.2 Truncation

Each time series was truncated to the first 200 index, purely as a means to improve computational speed. However, the report acknowledges the downstream effects of lowering complexity, especially in context of nonlinear models. By truncating to effectively a 10th of the series' original length, truncation has potential to remove major temporal dependencies which formulate the underlying dynamics the model is being trained upon.

For ESNs, RNNs and their advanced architectures, removing data will likely negatively impact learning of long-term dependencies and retention of important temporal patterns. This can lead to risk of overfitting to shorter patterns but was deemed necessary to allow efficient hyperparameter exploration.

[1368]:
```
# Truncating to 200
x = x[:, :200]
u = u[:, :200]

# Visualise truncated time series
plt.figure(figsize=(12, 6))
plt.plot(np.arange(x.shape[1]), x[exp], label="Displacement (x)",color = "red")
plt.plot(np.arange(u.shape[1]), u[exp], label="Force (u)",color = "green")
plt.xlabel("Time steps")
plt.ylabel("Measurement")
plt.title(f"Input Force to Output Displacement - Truncated, Experiment {exp+1}")
plt.legend()
plt.show()

# Visualise truncated data as phase space plot
plt.plot(x[exp], u[exp], label="Phase Space Plot", color='purple')
plt.xlabel("Displacement (x)")
plt.ylabel("Force (u)")
plt.title(f"Phase Space Diagram - Experiment {exp+1}")
plt.legend()
plt.show()
```


Input Force to Output Displacement - Truncated, Experiment 3
Displacement (x)
Force (u)
Measurement
Time steps
4
2
0
−2
−4
0
25
50
75
100
125
150
175
200


Phase Space Diagram - Experiment 3
Phase Space Plot
Force (u)
Displacement (x)
3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4

1.3.4 1.3 Normalisation

Min-Max Normalisation: `min_max_norm()`
Most ANN architecture operate effectively on appropriately scaled and normalised data. The importance of the following steps ensures experimental data is effectively preprocessed and the scale of data is consistent.

The method of normalisation applied is min-max normalisation using the function below. The function scales the data to a fixed range of (-1,1) by applying the formulation

$$x_n = 2\frac{x - x_{\min}}{x_{\max} - x_{\min}} - 1$$

As with other forms of machine learning, normalisation is an essential preprocessing step especially in the context of ANNs as these models rely on gradient-based optimisation such as ADAM. Large variations in the features can lead to numerical instability in phenomena such as exploding or vanishing gradients. Additionally, the step ensures features remain comparable regardless of scale.

The choice of min-max scaling technique over other forms of normalisation for ANNs ensures preservation of specific properties [6]. Though not universally better than other methods, it is fairly simple to implement and performs the necessary adjustments to the data range. By adhering to this range, certain properties can be satisfied; including ensuring data remains within the effective range of nonlinear activation functions, and the echo state property of ESNs.

[1370]:
```
# Min-Max Normalisation function
def min_max_norm(x):
    min_x = np.min(x)
    max_x = np.max(x)
    x_norm = 2 * (x - min_x) / (max_x - min_x) - 1
    return x_norm

x_norm = min_max_norm(x)
u_norm = min_max_norm(u)

#Print range
print(f"x Max-Min values: {np.max(x_norm)}, {np.min(x_norm)}")
print(f"u Max-Min values: {np.max(u_norm)}, {np.min(u_norm)}")
```

```
x Max-Min values: 1.0, -1.0
u Max-Min values: 1.0, -1.0
```

1.4 2. Echo State Neural Networks

Echo State Networks (ESNs) are a simple neural network architecture designed to process sequential data. The underlying principle lies in the concept of a fixed, hidden connected reservoir layer of neurones, which transforms input sequences into a high-dimensional dynamic state space. They are a considered a simplified formulation of more typical Recurrent Neural Networks (RNNs), as the reservoir remains untrained, and only the output weights W_o are optimised - typically through linear optimisation techniques. ESNs are considered appropriate for capturing the dynamical system, which leverages identification of temporal dependencies and nonlinear dynamics. The advantage of

the reservoir's stable dynamics enable the network to capture complex patterns without the need for iterative backpropagation through time [7].

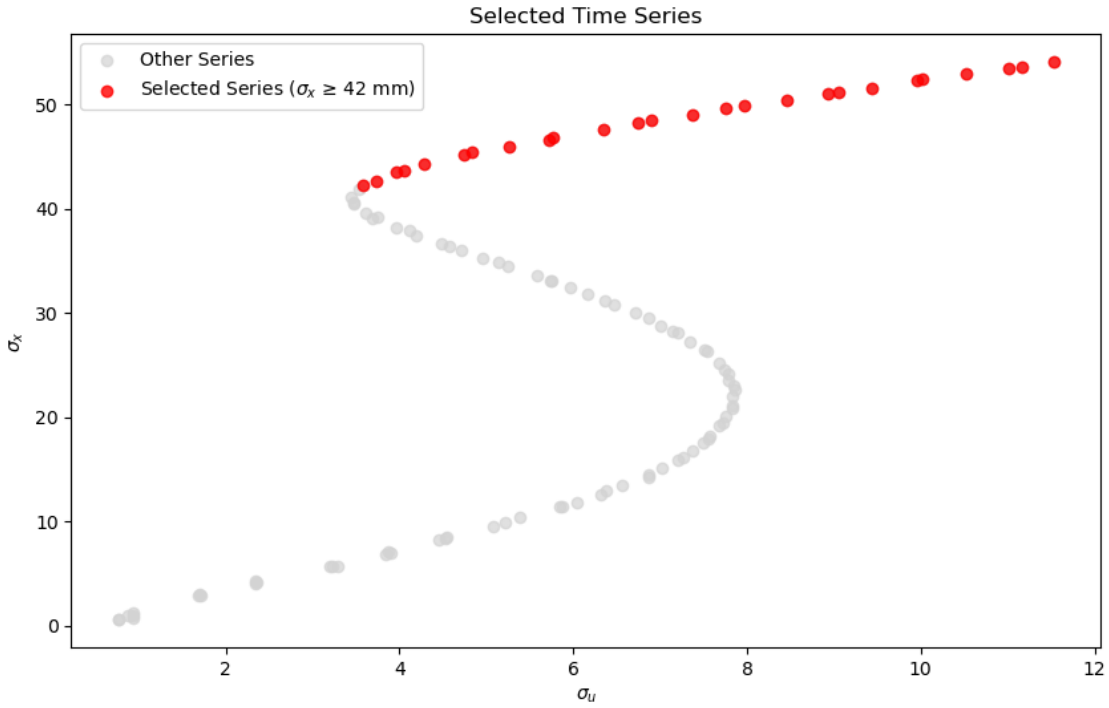
The use of ESN allows preliminary predictive modelling to be applied. Unlike standard RNNs, which require computationally expensive gradient-based training, and more hyperparameters to optimise, ESNs train only the output weights owing to their computational efficiency and less risk of vanishing/exploding gradient problems.

1.4.1 2.1 Data Separation

For exploratory processes, the ESN is applied for a select subset of time series where the standard deviation of the input signal σ_u is greater than a threshold value of 42mm.

[1450]:
```
# Criterion
selectidx = np.where(np.std(x_original, axis=1) >= 42)[0]

x_selected = x_norm[selectidx]
u_selected = u_norm[selectidx]

print("Selected series: ", len(selectidx))

# Visualise selected indices
plt.figure(figsize=(10, 6))
plt.scatter(std_u, std_x, color="lightgray", label="Other Series", alpha=0.7)
plt.scatter(std_u[selectidx], std_x[selectidx], color="red", label="Selected␣
↪Series ($\\sigma_x$  42 mm)", alpha=0.8)

plt.xlabel("$\\sigma_u$")
plt.ylabel("$\\sigma_x$")
plt.title("Selected Time Series")
plt.legend()
plt.show()
```

```
Selected series:  26
```



[1374]:
```
#Training and Testing Split
#--------------------------#
esn_split = 0.7
#--------------------------#

x_train, x_test, u_train, u_test = train_test_split(x_selected, u_selected,␣
 ↪test_size=1-esn_split)

print(f"x training data shape: {x_train.shape} ")
print(f"x testing data shape: {x_test.shape} ")
print(f"u training data shape: {u_train.shape} ")
print(f"u testing data shape: {u_test.shape} ")
```

```
x training data shape: (18, 200)
x testing data shape: (8, 200)
u training data shape: (18, 200)
u testing data shape: (8, 200)
```

1.4.2 2.2 ESN Formulation

Reservoir Creation `create_reservoir()`
Inputs: - `size`: The reservoir dimension N, which defines the number of neurones inside the hidden layer - `in_size`: Dimensionality of the input signal - `spectral radius`: The spectral radius of the reservoir weight matrix - `sparsity`: Proportion of zero elements in the reservoir

The ESN reservoir of a determined size $W \in \mathbb{R}^{N \times N}$ was initialised using this function. Within it, its weights can be randomly initialised with a uniform distribution between the ranges (-0.5, 0.5). An additional `sparsity` mask was applied to control the fraction of zero elements was an architectural choice as a form of regularisation, mimicking the sparsity-promoting action of conventional regularisation by limiting the number of possible connections. This deliberate design choice was made to prevent overfitting, especially as the data set is relatively small.

Additionally, the weights W are scaled to ensure its largest absolute eigenvalue $\lambda_{\max}(W)$ is mapped to the value of the defined spectral radius ρ to ensure stable dynamics [8] in the reservoir dynamics.

$$W \leftarrow W \cdot \frac{\rho}{\lambda_{\max}(W)}$$

For the Echo State Property to hold true, $\rho < 1$.

[1308]:
```
def create_reservoir(size, in_size=1, spectral_radius=0.9,sparsity = 0.2):
    # Reservoir masking
    W = np.random.rand(size, size) - 0.5
    mask = np.random.rand(size, size) < sparsity
    W *= mask

    # Scaling for spectral radius
    max_eig = np.max(np.abs(np.linalg.eigvals(W)))
    if max_eig > 0:
        W *= spectral_radius / max_eig

    # Step 4: Initialize the input weight matrix Win
    Win = np.random.rand(size, in_size) - 0.5

    return W, Win
```

State Evolution `run_reservoir()`
Inputs: - `W`: The reservoir weight matrix - `Win`: The input weight matrix - `u`: Input time series - `x`: Initial state vector of the reservoir

The state of the reservoir at time $t+1$ is evolved using this function, transforming the input sequence x to a high-dimensional state space x_t to capture temporal dependencies. Initially, this state is set to 0, but gradually modified as it is fed back into iterations and the model receives new information. Mathematically, evolution can be expressed as

$$x_{t+1} = f(W_{in}u_t + Wx_t)$$

where f is a nonlinear activation function mimicking the concept of neurones being in inactive or active states. Several logistic functions are explored in later sections. However, for the purposes of simplicity, and as the range of scaled inputs is (-1,1), the tanh function was used.

[1311]:
```
def run_reservoir(W, Win, u, x = None):
    if x is None:
        x = np.zeros(W.shape[0])

    all_x = np.zeros((u.shape[0], W.shape[0]))
```

```
    for i in range(u.shape[0]):
        x = np.tanh(np.dot(W, x) + np.dot(Win, u[i, :])) #nonlinear activation␣
↪function
        all_x[i, :] = x
    return all_x
```

Output Weight Training `train_reservoir()` - `all_x`: - `y`: Target output matrix - `ridge_alpha`:

Trains the output weights of the reservoir using linear regression methods, which aims to minimise the difference between the predicted outputs y and the updated reservoir outputs x_t

$$W_{out} = \text{argmin}_W ||y - W_x||_F^2$$

where $||\cdot||_F$ is the Frobenius norm. The additional ridge parameter α introduces a regularisation term, which is another measure to mitigate the risk of the ESN overfitting. This parameter modifies the optimisation problem into a ridge regression with closed form solution:

$$W_{out} = \text{argmin}_W ||y - W_x||_F^2 + \alpha||W||_F^2$$

using matrix notation, the code form is expressed as

$$W_{out} = (X^T X + \alpha I)^{-1} X^T y$$

[1314]:
```
def train_reservoir(all_x, y,ridge_alpha=1e-6):
    I = np.identity(all_x.shape[1])
    return np.linalg.lstsq(all_x.T @ all_x + ridge_alpha * I, all_x.T @ y,␣
↪rcond=None)[0]
```

Predicting Output `predict_reservoir()`
Inputs: - `Wout`: The output weight matrix. - `W` : The reservoir weight matrix. - `Win`:he input weight matrix. - `u`: The input time series

Used to predict the output signal y_t for the given input sequence u_t using the trained output weights W_{out} by linear combination

$$y_t = W_{out}^T x_t$$

where it is used to evaluate the ESN's ability to generalise and predict output signals for new testing input sequences

[1317]:
```
def predict_reservoir(Wout, W, Win, u):
    x = np.zeros(W.shape[0])
    y = np.zeros((u.shape[0], Wout.shape[1]))

    # Loop for all time  series
    for i in range(u.shape[0]):
        # Evaluate the reservoir at the current time step
        x = np.tanh(np.dot(W, x) + np.dot(Win, u[i, :]))
        y[i, :] = np.dot(Wout.T, x)
    return y
```

1.4.3 2.3 ESN Training

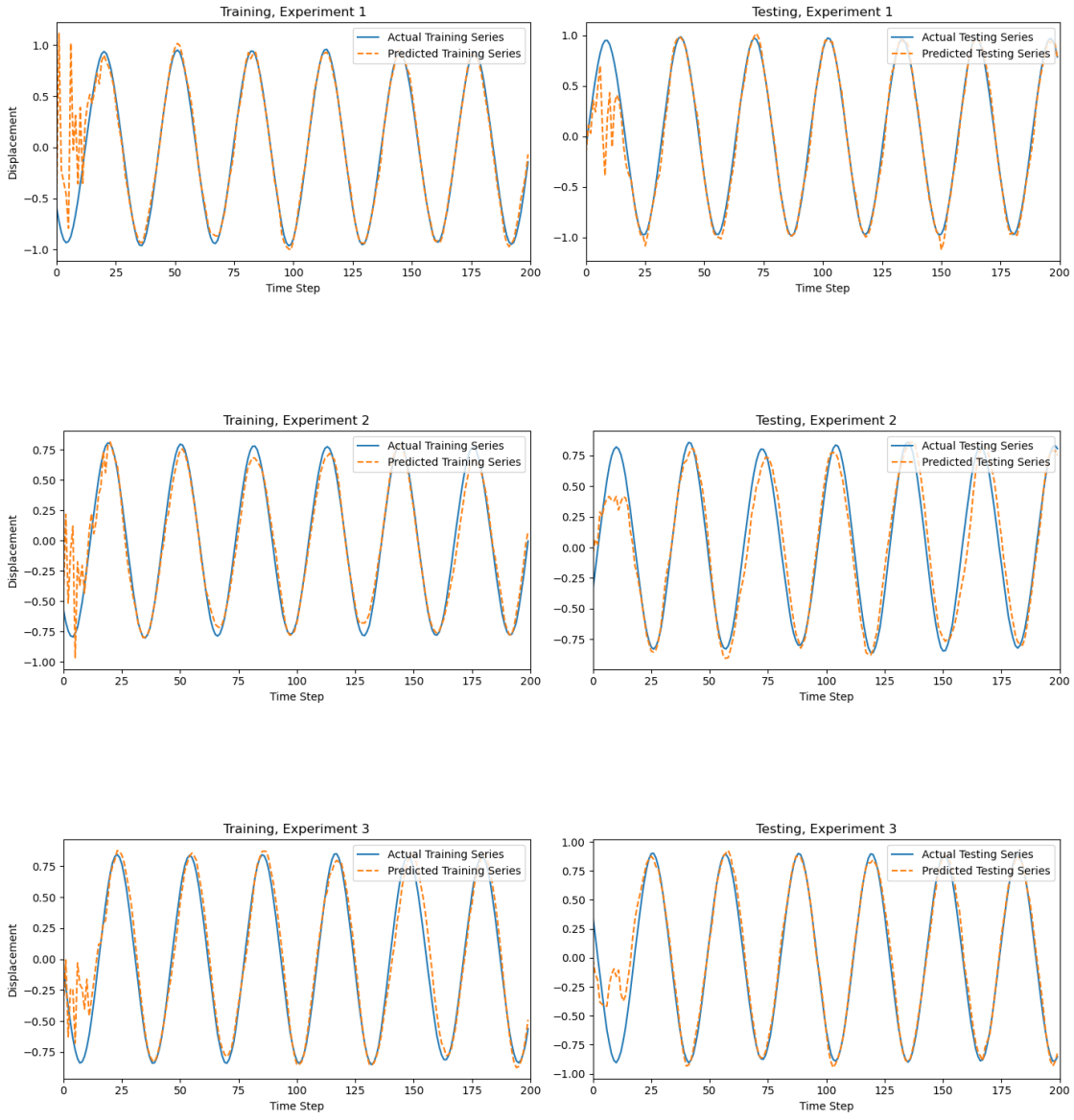
In the following seegment, the initial trained ESN model was trained to evaluate the weight matrices W_{in}, W_{out}, W. This trained ESN was then applied and the model's performance was evaluated by comparing its predicted output to the actual time series for both (a) a training set and (b) a testing set. Both the predicted and actual time series were plotted to assess the ESN's ability to capture the dynamics of the system for unseen data.

[1376]:
```
# ESN Hyperparameters
#--------------------------#
reservoir_size = 100    # Number of neurons
input_size = 1          # DImension of time series input
spectral_radius = 0.9   # Spectral Radius
washout_period = 20     # Indices to ignore for ESN transience
ridge_alpha = 1e-6      # Regularisation parameter
sparsity = 0.2          # Reservoir sparsity
#--------------------------#
```

[1378]:
```
# 1. Creating reservoir
W, Win = create_reservoir(size=reservoir_size, in_size=input_size,␣
 ↪spectral_radius=spectral_radius,sparsity = sparsity)

# 2. Training loop for all u series
train_states = []
for i in range(u_train.shape[0]):
    states = run_reservoir(W, Win, u_train[i].reshape(-1, 1))
    train_states.append(states[washout_period:])

all_x_train = np.vstack(train_states)
y_train = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_train])

# 3. Run ESN Training
Wout = train_reservoir(all_x_train, y_train,ridge_alpha=ridge_alpha)
```

[1380]:
```
#Predictions for a set of ESN experiments
for exp in range(0,3):
    train_pred = predict_reservoir(Wout, W, Win, u_train[exp].reshape(-1, 1))
    test_pred = predict_reservoir(Wout, W, Win, u_test[exp].reshape(-1, 1))

    # Visualise training predictions
    plt.figure(figsize=(14, 4))
    plt.subplot(1, 2, 1)
    plt.plot(x_train[exp], label="Actual Training Series")
    plt.plot(train_pred, label="Predicted Training Series", linestyle='--')
    plt.xlabel("Time Step")
    plt.ylabel("Displacement")
    plt.xlim(0,200)
    plt.title(f"Training, Experiment {exp+1}")
```

```
    plt.legend(loc='upper right')

    #Testing
    plt.subplot(1, 2, 2)
    plt.plot(x_test[exp], label="Actual Testing Series")
    plt.plot(test_pred, label="Predicted Testing Series", linestyle='--')
    plt.xlabel("Time Step")
    plt.xlim(0,200)
    plt.title(f"Testing, Experiment {exp+1}")
    plt.legend(loc='upper right')

    plt.tight_layout()
    plt.show()
```



The plots above illustrate the performance of the ESN model, compared against time series for both training and testing datasets for sample time series. The training plots (left column) show a confident fit, with the predicted series closely aligning with the actual data, indicating strong learning capability. In the testing plots (right column), while the model generalises well, minor deviations occur at the start of the time series. This region is explained by the ESN being in a transient stage, for which the reservoir has not fully stabilised. Overall, the ESN demonstrates robust predictive accuracy across multiple experiments, maintaining phase and amplitude consistency in both datasets.

Evaluating the performance of the ESN was done was calculating the Mean Squared Error (MSE) for all training and testing sets separately. The MSE is formulated as [9]

$$\text{MSE} = \frac{1}{T-k}\sum_{t=k}^{T-1}(x_t - \hat{x}_t)^2$$

for each time series, where k is the washout period, subtracted for the total entries in a time series T. The MSE acts as a quantitive measure of performance and generalisability. Note the use of the `washout_period` parameter, which is the index upto which calculation of the error is excluded to account for the transient initiation phase.

[1382]:
```
# MSE over training set
train_mse_list = []
for i in range(u_train.shape[0]):
    train_pred_series = predict_reservoir(Wout, W, Win, u_train[i].reshape(-1,␣
 ↪1))
    train_mse_list.append(mean_squared_error(x_train[i][washout_period:],␣
 ↪train_pred_series[washout_period:]))

train_mse = np.mean(train_mse_list)
train_mse_std = np.std(train_mse_list)


# MSE over testing set
test_mse_list = []
for i in range(u_test.shape[0]):
    test_pred_series = predict_reservoir(Wout, W, Win, u_test[i].reshape(-1, 1))
    test_mse_list.append(mean_squared_error(x_test[i][washout_period:],␣
 ↪test_pred_series[washout_period:]))

test_mse = np.mean(test_mse_list)
test_mse_std = np.std(test_mse_list)

print(f"Average Training series MSE: {train_mse},Standard Deviation:␣
 ↪{train_mse_std}")
print(f"Average Testing series MSE: {test_mse}, Standard Deviation:␣
 ↪{test_mse_std}")
```

```
Average Training series MSE: 0.007058611022131908,Standard Deviation:
0.005774009066660622
Average Testing series MSE: 0.006241305055424697, Standard Deviation:
0.005521492013747539
```

The similar average MSE values between training and testing data implies that the current ESN configuration generalises well to the unseen test data and suggests that the model is able to effectively predict unseen sequences.

The low MSE values indicate that the ESN captures the underlying dynamics of the system accurately, and despite the nonlinear nature of the NTMD, the model can represent these dynamics accurately, with the current ESN configuration. The relatively low reservoir size and introduction of regularisation have likely contributed to the lack of overfitting, despite the limited quantity of data.

1.4.4 2.4 ESN Validation

One method to validate the weights generated by the ESN and its ability to generalise is to calculate the MSE across different subsets of the data set, in a way that parallels bootstrapping techniques. This is known as K-fold validation, for which the procedure is as follows [10]

1. The data is divided into k number of fold for training, for which the remainder is used for validation purposes
2. For each training fold, reservoir creation and ESN training is performed and the MSE is computed
3. Predictions made by this model are used to compute an MSE value for the validation (testing) fold to assess generalisability

For nonlinear systems, this validation serves to rigorously assess quantitatively the ESN's robustness to variations between u and x.

[1387]:
```
# K-Fold Cross Validation
#--------------------------#
folds = 5  # Number of folds
#--------------------------#

kf = KFold(n_splits=folds, shuffle=True, random_state=2143062)

train_mse_folds = []
val_mse_folds = []

# Cross validation iteration
for train_index, val_index in kf.split(u_train):
    # Split data into training and validation sets for this fold
    u_train_fold, u_val_fold = u_train[train_index], u_train[val_index]
    x_train_fold, x_val_fold = x_train[train_index], x_train[val_index]

    # Create reservoir with current parameters
    W, Win = create_reservoir(size=reservoir_size, in_size=1,␣
 ↪spectral_radius=spectral_radius)
```

```
    # Train the ESN on the training fold
    train_states = []
    for i in range(u_train_fold.shape[0]):
        states = run_reservoir(W, Win, u_train_fold[i].reshape(-1, 1))
        train_states.append(states[washout_period:])  # Apply washout
    all_x_train = np.vstack(train_states)
    y_train = np.vstack([x[washout_period:].reshape(-1, 1) for x in␣
↪x_train_fold])

    # Train output weights
    Wout = train_reservoir(all_x_train, y_train)

    # Training MSE for fold
    train_pred = np.vstack([predict_reservoir(Wout, W, Win, u.reshape(-1,␣
↪1))[washout_period:] for u in u_train_fold])
    train_mse = mean_squared_error(y_train, train_pred)
    train_mse_folds.append(train_mse)

    # Testing MSE for fold to validate
    val_pred = np.vstack([predict_reservoir(Wout, W, Win, u.reshape(-1,␣
↪1))[washout_period:] for u in u_val_fold])
    y_val = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_val_fold])
    val_mse = mean_squared_error(y_val, val_pred)
    val_mse_folds.append(val_mse)

# Calculate average training and validation MSE across all folds
avg_train_mse = np.mean(train_mse_folds)
std_train_mse = np.std(train_mse_folds)
avg_val_mse = np.mean(val_mse_folds)
std_val_mse = np.std(val_mse_folds)

print(f"Average Training MSE: {avg_train_mse} (Std Dev: {std_train_mse})")
print(f"Average Validation MSE: {avg_val_mse} (Std Dev: {std_val_mse})")
```

```
Average Training MSE: 0.005927219976210901 (Std Dev: 0.0014716041982470746)
Average Validation MSE: 0.008378108074088158 (Std Dev: 0.002498474566999234)
```

Training MSE reflects how well the model fits the data it was trained on in each fold, while validation MSE reflects how well the model performs on unseen data within each fold. The relatively small values indicate the ESN effectively captures the NTMD system's nonlinear dynamics with low training error, while the slightly higher validation error suggests good but not perfect generalisation.

1.5 3. Hyperparameter Tuning

1.5.1 3.1 Hyperparameters for ESNs

Hyperparameter tuning is critical for optimising the performance of any ANN, as their flexible architecture increases its dependence on proper parameter selection as they govern the behaviour

of models which are specific to the characteristics of the system they are intended to predict. For more complex networks, a large number of parameters must be optimised, in both the architecture and within training (as explicitly shown in Section 4). For illustration purposes and demonstration of proven techniques, a comparison of four of the following ESN parameters was assessed:

`reservoir_size` (N): Dimension of neurones in the hidden layer. - Low: Fewer neurones in the reservoir, leading to reduced capacity to capture complex nonlinear dynamics, potentially underfitting the data - High: Larger reservoir enables greater expressive power and capacity to model such patterns but increases computational cost and risk of overfitting, especially with limited training data

`spectral_radius` (ρ): Largest eigenvalue of the reservoir weight matrix - Low: Reservoir dynamics fade quickly, reducing memory and the ability to capture long-term dependencies - High: Reservoir may become unstable, leading to chaotic behaviour and loss of predictive accuracy

`ridge_alpha` (α): Regularisation parameter for the ridge regression used in training the output weights - Low: Minimal regularisation, potentially overfitting the training data, especially with noisy datasets or small training sets - High: Strong regularisation, reducing overfitting but potentially underfitting if the model is overly constrained

`sparsity` (S: Fraction of nonzero connections in the reservoir weight matrix. - Low: Fewer connections, leading to a simpler reservoir with reduced capacity to capture complex relationships - High: Dense reservoir, increasing capacity but risking redundancy and higher computational costs

1.5.2 3.2 Optimisation

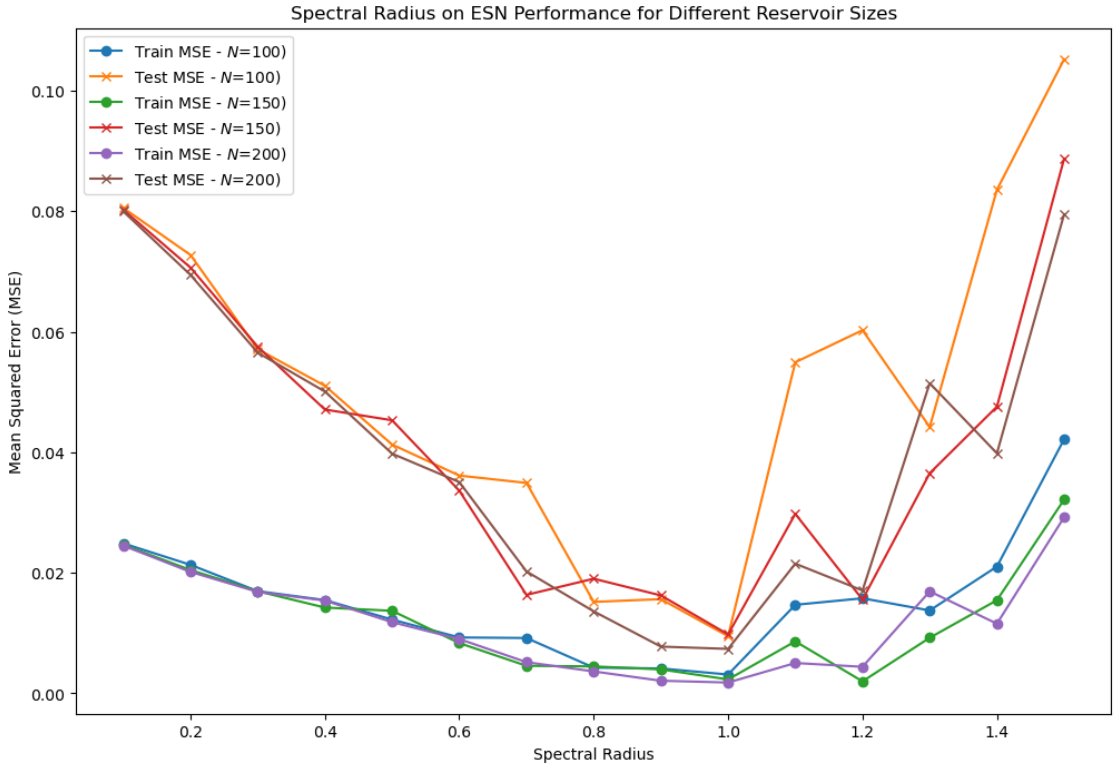
As a preliminary investigation, the impacts of `reservoir_size` and `spectral_radius` were compared. These two were chosen due to their large impact on model performance [11] . The visual below was formed using conventional manual search training and testing iterations over a range of specified parameter lists.

[1336]:
```
# Optimisation of Reservoir sizes against Spectral Radius
#-------------------------#
reservoir_sizes_list = [ 100, 150,200]
spectral_radius_list =  np.linspace(0.1,1.5,15)
#-------------------------#

train_mse_list_size = []
test_mse_list_size = []

# Iterate over reservoir sizes
for reservoir_size in reservoir_sizes_list:
    train_mse_list_res = []
    test_mse_list_res = []


    for spectral_radius in spectral_radius_list:
        # Create reservoir
        W_res, Win_res = create_reservoir(size=reservoir_size,␣
↪in_size=input_size, spectral_radius=spectral_radius, sparsity=sparsity)
```

```
        # Train
        train_states_res = []
        for i in range(len(u_train)):
            states_res = run_reservoir(W_res, Win_res, u_train[i].reshape(-1, 1))
            train_states_res.append(states_res[washout_period:])
        all_x_train_res = np.vstack(train_states_res)
        y_train_res = np.vstack([x[washout_period:].reshape(-1, 1) for x in␣
↪x_train])

        # Train reservoir output weights
        Wout_res = train_reservoir(all_x_train_res, y_train_res,␣
↪ridge_alpha=ridge_alpha)

        # Evaluate on training data
        train_mse_res = []
        for i in range(len(u_train)):
            train_pred_series_res = predict_reservoir(Wout_res, W_res, Win_res,␣
↪u_train[i].reshape(-1, 1))
            train_mse_res.append(mean_squared_error(x_train[i][washout_period:],␣
↪train_pred_series_res[washout_period:]))
        train_mse_list_res.append(np.mean(train_mse_res))

        # Evaluate on testing data
        test_mse_res = []
        for i in range(len(u_test)):
            test_pred_series_res = predict_reservoir(Wout_res, W_res, Win_res,␣
↪u_test[i].reshape(-1, 1))
            test_mse_res.append(mean_squared_error(x_test[i][washout_period:],␣
↪test_pred_series_res[washout_period:]))
        test_mse_list_res.append(np.mean(test_mse_res))

    # Store results for the current reservoir size
    train_mse_list_size.append(train_mse_list_res)
    test_mse_list_size.append(test_mse_list_res)
```

[1337]:
```
# Visualisation of MSE performance for reservoir sizes and spectral radii
plt.figure(figsize=(12, 8))
for i, reservoir_size in enumerate(reservoir_sizes_list):
    plt.plot(spectral_radius_list, train_mse_list_size[i], label=f"Train MSE -␣
↪$N$={reservoir_size})", marker='o')
    plt.plot(spectral_radius_list, test_mse_list_size[i], label=f"Test MSE -␣
↪$N$={reservoir_size})", marker='x')

plt.xlabel("Spectral Radius")
plt.ylabel("Mean Squared Error (MSE)")
```

```
plt.title("Spectral Radius on ESN Performance for Different Reservoir Sizes")
plt.legend()
plt.show()
```



The relationship of the MSE against spectral radius ρ is consistent and evident across all reservoir sizes N. As the radius increases, the error decreases initially as the model performance improves, reaching an optimal at $0.8 < \rho < 1$. Beyond 1.0, performance quickly decreases as expected due to loss of the Echo state property, resulting in chaotic reservoir dynamics and instability.
Comparing reservoir sizes show optimal performance with higher reservoir sizes of $N = 200$, which shows an increase leads to the ESN generally achieving a lower MSE.

Grid Search

Manual hyperparameter tuning is time-consuming and prone to human error, and often suboptimal due to its inability to systematically explore the parameter space. The following section applies an automated search process known as grid search [12], evaluating the performance over a specified range of parameters systematically.

The `product()` function was applied, generating the Cartesian product of the specified parameter values below. The ESN model was then trained for each combination of hyperparameters and the MSE for both training and testing sets was evaluated for comparison.

[1353]:
```
# Parameter List
#--------------------------#
```

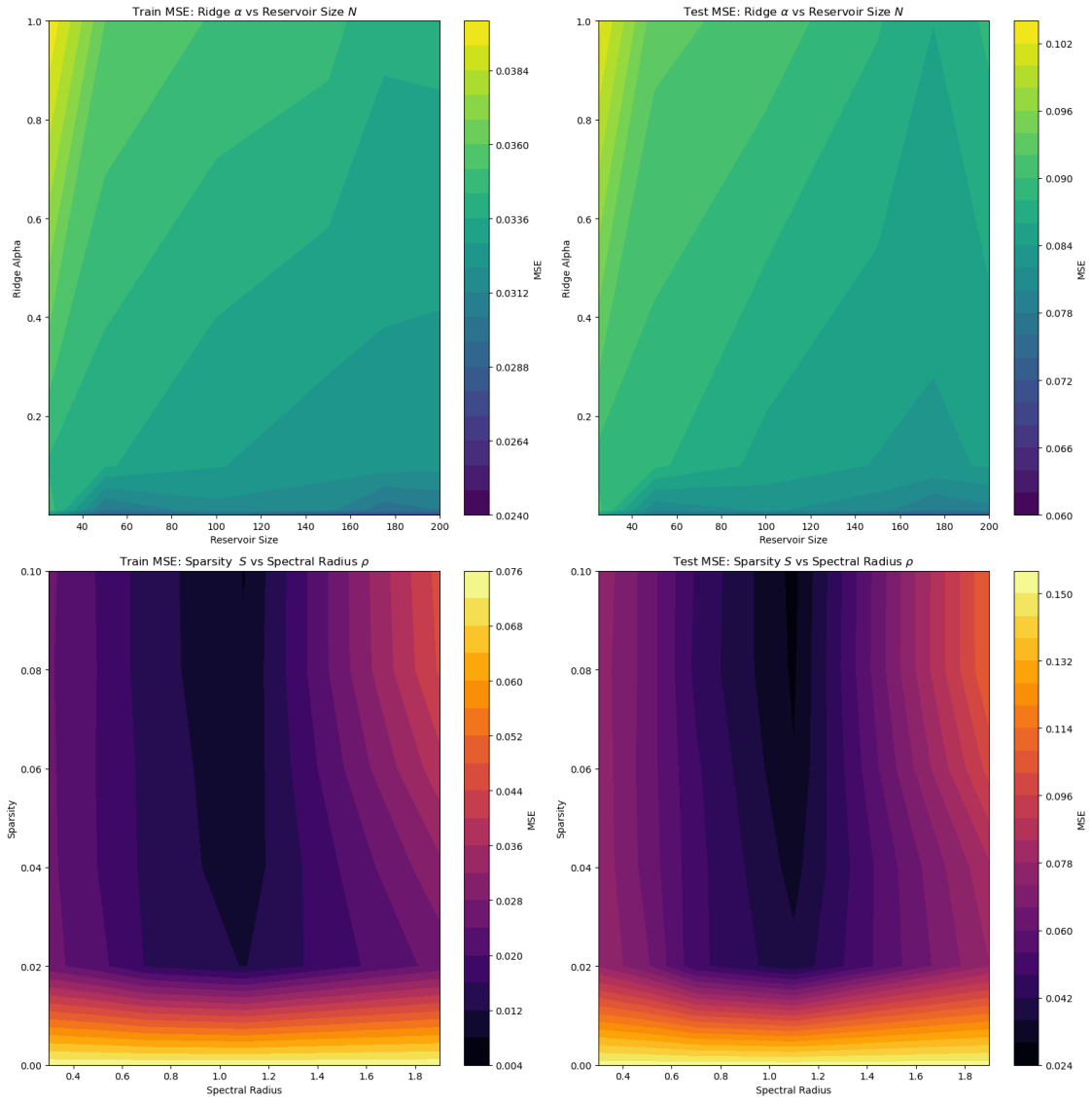
```
reservoir_sizes = [25,50, 100, 150,175,200]
spectral_radii = [0.3,0.5,0.7, 1.9, 1.1]
ridge_alphas = [1e-8,1e-6, 1e-4, 1e-2,1e-1,1e0]
sparsities = [0,0.02, 0.04, 0.06, 0.08,0.1]
#--------------------------#
```

[1342]:
```
# Grid Search for hyperparameter optimisation
optimisation_results = []

for size, radius, alpha, sparsity in product(reservoir_sizes, spectral_radii,␣
↪ridge_alphas, sparsities):
    # Create reservoir
    W_op, Win_op = create_reservoir(size=size, in_size=input_size,␣
↪spectral_radius=radius, sparsity=sparsity)

    # Training ESN
    train_states_op = []
    for i in range(len(u_train)):
        states_op = run_reservoir(W_op, Win_op, u_train[i].reshape(-1, 1))
        train_states_op.append(states_op[washout_period:])
    all_x_train_op = np.vstack(train_states_op)
    y_train_op = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_train])

    Wout_op = train_reservoir(all_x_train_op, y_train_op, ridge_alpha=alpha)

    # MSE Train
    train_mse_list_op = []
    for i in range(len(u_train)):
        train_pred_series_op = predict_reservoir(Wout_op, W_op,␣
↪Win_op,u_train[i].reshape(-1, 1))
        train_mse_list_op.append(mean_squared_error(x_train[i][washout_period:],␣
↪train_pred_series_op[washout_period:]))
    train_mse_op = np.mean(train_mse_list_op)

    # MSE Test
    test_mse_list_op = []
    for i in range(len(u_test)):
        test_pred_series_op = predict_reservoir(Wout_op, W_op, Win_op, u_test[i].
↪reshape(-1, 1))
        test_mse_list_op.append(mean_squared_error(x_test[i][washout_period:],␣
↪test_pred_series_op[washout_period:]))
    test_mse_op = np.mean(test_mse_list_op)

    # Record results
    optimisation_results.append({
        "reservoir_size": size,
        "spectral_radius": radius,
```

```
        "ridge_alpha": alpha,
        "sparsity": sparsity,
        "train_mse": train_mse_op,
        "test_mse": test_mse_op
    })

# Convert to pd dataframe
results_df = pd.DataFrame(optimisation_results)
```

[1419]:
```
# Visualisation of heatmaps for various parameters
fig, axs = plt.subplots(2, 2, figsize=(16, 16))

# Train MSE - ridge_alpha vs reservoir_size
pivot_table_train_1 = results_df.pivot_table(index="ridge_alpha",␣
 ↪columns="reservoir_size", values="train_mse")
train_contour_1 = axs[0, 0].contourf(
    pivot_table_train_1.columns,
    pivot_table_train_1.index,
    pivot_table_train_1.values,
    cmap="viridis",
    levels=20
)
cbar_train_1 = fig.colorbar(train_contour_1, ax=axs[0, 0])
cbar_train_1.set_label("MSE")
axs[0, 0].set_title("Train MSE: Ridge $\\alpha$ vs Reservoir Size $N$")
axs[0, 0].set_xlabel("Reservoir Size")
axs[0, 0].set_ylabel("Ridge Alpha")

# Test MSE - ridge_alpha vs reservoir_size
pivot_table_test_1 = results_df.pivot_table(index="ridge_alpha",␣
 ↪columns="reservoir_size", values="test_mse")
test_contour_1 = axs[0, 1].contourf(
    pivot_table_test_1.columns,
    pivot_table_test_1.index,
    pivot_table_test_1.values,
    cmap="viridis",
    levels=20
)
cbar_test_1 = fig.colorbar(test_contour_1, ax=axs[0, 1])
cbar_test_1.set_label("MSE")
axs[0, 1].set_title("Test MSE: Ridge $\\alpha$ vs Reservoir Size $N$")
axs[0, 1].set_xlabel("Reservoir Size")
axs[0, 1].set_ylabel("Ridge Alpha")

# Train MSE - sparsity vs spectral_radius
pivot_table_train_2 = results_df.pivot_table(index="sparsity",␣
 ↪columns="spectral_radius", values="train_mse")
```

```
train_contour_2 = axs[1, 0].contourf(
    pivot_table_train_2.columns,
    pivot_table_train_2.index,
    pivot_table_train_2.values,
    cmap="inferno",
    levels=20
)
cbar_train_2 = fig.colorbar(train_contour_2, ax=axs[1, 0])
cbar_train_2.set_label("MSE")
axs[1, 0].set_title("Train MSE: Sparsity  $S$ vs Spectral Radius $\\rho$")
axs[1, 0].set_xlabel("Spectral Radius")
axs[1, 0].set_ylabel("Sparsity")

# Test MSE - sparsity vs spectral_radius
pivot_table_test_2 = results_df.pivot_table(index="sparsity",␣
 ↪columns="spectral_radius", values="test_mse")
test_contour_2 = axs[1, 1].contourf(
    pivot_table_test_2.columns,
    pivot_table_test_2.index,
    pivot_table_test_2.values,
    cmap="inferno",
    levels=20
)
cbar_test_2 = fig.colorbar(test_contour_2, ax=axs[1, 1])
cbar_test_2.set_label("MSE")
axs[1, 1].set_title("Test MSE: Sparsity $S$ vs Spectral Radius $\\rho$")
axs[1, 1].set_xlabel("Spectral Radius")
axs[1, 1].set_ylabel("Sparsity")

plt.tight_layout()
plt.show()
```



The heatmaps above illustrate the influence of hyperparameters on the training and testing MSE performance for the ESN. Increasing the reservoir size generally reduces MSE, behaviour consistent with one demonstrated in the earlier plot with diminishing returns after a certain size due to overfitting on the training set. Similarly, lower values of $\alpha < 0.2$ improve performance, indicating minimal regularisation is optimal for this setup. The relationship between sparsity and spectral radius is interesting, and highlights that low $S < 0.05$ combined with a $\rho \approx 1$ yields the best results, aligning with the Echo State Property. However, very high $\rho > 1.2$ values degrade performance due to instability.

Bayesian Optimisation

Though grid search is useful for evaluating large ranges, specific evaluation can become increasingly computationally expensive even for the ESN model. As an exploratory method to explore the effects of hyperparameters, it is a good visualisation tool, however the grid will exhaustively search and

ignore redundant combinations - combining explorative and exploitative searching.

In contrast, Bayesian Optimisation methods utilise probabilistic models to estimate the objective function $f(x)$ to search the parameter space adaptively [13] which reduces computational overhead. This was implemented using the `Optuna` framework. First, an objective function `objective()` was defined using suggested trial values within defined ranges for each parameter. The ESN is constructed using sampled hyperparameters as before, with the aim of minimising the MSE. Over `n_trials`, the objective function was run to return the optimal hyperparameters found.

[1389]:
```
# Optuna objective function
def objective(trial):
    # Suggest values for each hyperparameter
    reservoir_size = trial.suggest_int("reservoir_size", 25, 300)
    spectral_radius = trial.suggest_float("spectral_radius", 0.1, 1.5)
    ridge_alpha = trial.suggest_float("ridge_alpha", 1e-12, 1e0, log=True)
    sparsity = trial.suggest_float("sparsity", 0.0, 0.4)

    # Create reservoir
    W, Win = create_reservoir(size=reservoir_size, in_size=input_size,␣
 ↪spectral_radius=spectral_radius, sparsity=sparsity)

    # Train
    train_states = []
    for i in range(len(u_train)):
        states = run_reservoir(W, Win, u_train[i].reshape(-1, 1))
        train_states.append(states[washout_period:])
    all_x_train = np.vstack(train_states)
    y_train = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_train])

    # Train weights
    Wout = train_reservoir(all_x_train, y_train, ridge_alpha=ridge_alpha)

    # MSE Test
    test_mse = []
    for i in range(len(u_test)):
        test_pred_series = predict_reservoir(Wout, W, Win, u_test[i].reshape(-1,␣
 ↪1))
        test_mse.append(mean_squared_error(x_test[i][washout_period:],␣
 ↪test_pred_series[washout_period:]))

    return np.mean(test_mse)

# Create study
study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=50, timeout=600)
best_params = study.best_params
best_value = study.best_value
```

```
print("Optimal Hyperparameters:")
for param, value in best_params.items():
    print(f"{param}: {value}")
print(f"Optimal Test MSE: {best_value}")
```

```
Optimal Hyperparameters:
reservoir_size: 277
spectral_radius: 1.0712450776351856
ridge_alpha: 1.6821893517096605e-06
sparsity: 0.12894353169268413
Optimal Test MSE: 0.0018412699440084927
```

Generalisation Approaches

In addition to optimal hyperparameter tuning, other methods to imrpove the generalisability of the ESN at the higher level include:

Input noise augmentation:
Small Gaussian noise can be added to the input x during training, to simulate experimental variability and aid the ESN in recognising underlying patterns consistent over data sets. This approach parallels dropout regularisation, as both apply the concept of stochastic regularisation, which forces the ESN to focus on identifying the underlying structure of the data as opposed to on noise-free inputs.

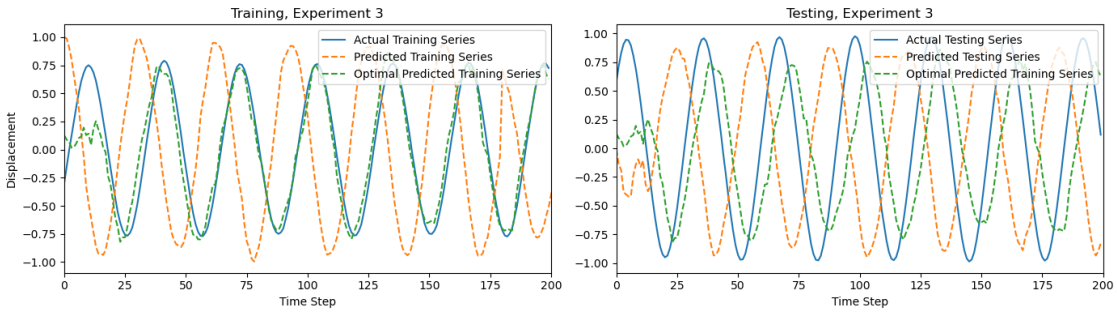
Ensemble ESNs:
Training an ensemble of ESNs involves creating multiple networks with slightly varied initialisations or parameter settings and averaging their predictions. This method can be applied to reduce intrinsic variance in relying on a single ESN model, and leverages the learning patterns of multiple models, making the overall prediction more robust and less prone to overfitting caused by individual ESN configurations. This can also extend to training separate ESNs or tailoring reservoirs for high- and low-amplitude series, before aggregating their predictions based on the characteristics of the input data.

1.5.3 3.3 Performance

These optimised hyperparameters were tested for both sample prediction test sequences, and for the full normalised data set.

[1421]:
```
exp_op = 6
# Training with optimised parameters
W_x, Win_x = create_reservoir(size=best_params["reservoir_size"], in_size=1,␣
 ↪spectral_radius=best_params["spectral_radius"],␣
 ↪sparsity=best_params["sparsity"])
train_states_x = []
for i in range(u_train.shape[0]):
    states_x = run_reservoir(W_x, Win_x, u_train[i].reshape(-1, 1))
    train_states_x.append(states_x[washout_period:])

all_x_train_x = np.vstack(train_states_x)
```

```
y_train_x = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_train])

Wout_x = train_reservoir(all_x_train_x,␣
↪y_train_x,ridge_alpha=best_params["ridge_alpha"])

train_pred_x = predict_reservoir(Wout_x, W_x, Win_x, u_train[exp_op].reshape(-1,␣
↪1))
test_pred_x = predict_reservoir(Wout_x, W_x, Win_x, u_test[exp_op].reshape(-1,␣
↪1))

# Visualise training predictions
plt.figure(figsize=(14, 4))
plt.subplot(1, 2, 1)
plt.plot(x_train[exp_op], label="Actual Training Series")
plt.plot(train_pred, label="Predicted Training Series", linestyle='--')
plt.plot(train_pred_x, label="Optimal Predicted Training Series", linestyle='--')
plt.xlabel("Time Step")
plt.ylabel("Displacement")
plt.xlim(0,200)
plt.title(f"Training, Experiment {exp+1}")
plt.legend(loc = "upper right")

#Testing
plt.subplot(1, 2, 2)
plt.plot(x_test[exp_op], label="Actual Testing Series")
plt.plot(test_pred, label="Predicted Testing Series", linestyle='--')
plt.plot(train_pred_x, label="Optimal Predicted Training Series", linestyle='--')
plt.xlabel("Time Step")
plt.xlim(0,200)
plt.title(f"Testing, Experiment {exp+1}")
plt.legend(loc = "upper right")

plt.tight_layout()
plt.show()
```



[1425]:
```
# Training on entire data set
full_W, full_Win = create_reservoir(size=best_params["reservoir_size"],␣
↪in_size=input_size, spectral_radius=best_params["spectral_radius"],sparsity =␣
↪best_params["sparsity"])

all_states_full = []
for i in range(u_norm.shape[0]):
    states_full = run_reservoir(full_W, full_Win, u_norm[i].reshape(-1, 1))
    all_states_full.append(states_full[washout_period:])

all_x_full = np.vstack(all_states_full)
y_full = np.vstack([x[i][washout_period:].reshape(-1, 1) for i in range(x_norm.
↪shape[0])])

full_Wout = train_reservoir(all_x_full,␣
↪y_full,ridge_alpha=best_params["ridge_alpha"])

mse_list_full = []
for i in range(u.shape[0]):
    pred_series_full = predict_reservoir(full_Wout, full_W, full_Win, u[i].
↪reshape(-1, 1))
    mse_list_full.append(mean_squared_error(x_norm[i][washout_period:],␣
↪pred_series_full[washout_period:]))
overall_mse_full = np.mean(mse_list_full)

print(f"Average MSE: {overall_mse_full}")
```

```
Average MSE: 3604652.1501764115
```

Performance Evaluation

On the full data set, the average MSE is substantially higher indicating the model is unable to generalise to the entire data set. There are some possible explanations for this, stemming for the nonlinear behaviours exhibited by the data set variances explored in the data from Section 1.

Fundamentally, the NTMD system is nonlinear in nature, leading to different beheaviours across different inputs and initial conditions which have been identified into several regimes. The ESN was trained on exclusively high variance data of $\sigma_u > 42$, the model was only exposed to specific high-energy dynamics - areas exhibiting large oscillations and, as identified, spring saturation.

As ESNs are sensitive to scale and variance of input signals even after normlisation, this tailors the model to only capture behaviour within the scope. When trained on low variance data which are likely dominated by damping effects, which are governed by different dynamics and thus map x to u through different interactions. When the ESN is trained on the entire dataset dynamics differ significantly from those learned during training on high-variance data appearing as "outliers" or less relevant states to the reservoir. This likely causes dilution in the reservoir's pattern recognitiion, adapting to all noninear regimes resulting in an overall poor performance.

1.6 4. Recurrent Neural Networks

1.6.1 4.1 Data Separation

[1054]:
```
#Training and Testing Split
#-------------------------#
rnn_split = 0.7
#-------------------------#

x_train, x_test, u_train, u_test = train_test_split(x_norm, u_norm, test_size=1␣
 ↪- rnn_split, random_state=42)

print(f"x training data shape: {x_train.shape} ")
print(f"x testing data shape: {x_test.shape} ")
print(f"u training data shape: {u_train.shape} ")
print(f"u testing data shape: {u_test.shape} ")
```

```
x training data shape: (75, 200)
x testing data shape: (33, 200)
u training data shape: (75, 200)
u testing data shape: (33, 200)
```

Tensor Conversion

Post train and test splitting, the matrices X and U are formatted into a single input state to be processed by an RNN. The concatenation step ensures the model is able to learn the coupled dynamics of the system. By converting to PyTorch tensors from NumPy arrays, the transformed structure is compatible with gradient-based optimisation algorithms and tensor operations which require an additional feature dimension for each time step. The operation `unsqueeze()` is used to create an additional feature dimension for both matrices.

[1060]:
```
#Tensor Conversion
#Train
x_train_tensor = torch.tensor(x_train, dtype=torch.float32).unsqueeze(-1)
u_train_tensor = torch.tensor(u_train, dtype=torch.float32).unsqueeze(-1)
input_data_train = torch.cat((x_train_tensor, u_train_tensor), dim=-1)
output_data_train = torch.tensor(x_train, dtype=torch.float32).unsqueeze(-1)

#Test
x_test_tensor = torch.tensor(x_test, dtype=torch.float32).unsqueeze(-1)
u_test_tensor = torch.tensor(u_test, dtype=torch.float32).unsqueeze(-1)

#Combining u and x into an input
input_data_test = torch.cat((x_test_tensor, u_test_tensor), dim=-1)
output_data_test = torch.tensor(x_test, dtype=torch.float32).unsqueeze(-1)

print("Training input data shape:", input_data_train.shape)
print("Testing input data shape:", input_data_test.shape)
```

```
Training input data shape: torch.Size([75, 200, 2])
```

```
Testing input data shape: torch.Size([33, 200, 2])
```

1.6.2 4.2 RNN Formulation

Recurrent Neural Networks (RNNs) are a class of ANNs which, in similar design to ESNs are used to model sequential data by maintaining a memory state which evolves in time. This hidden layer is similar to ESNs in which both possess a large number of neurones dynamic to the input states, and each containing recurrent neural connections, allowing information to persist. More generally, RNNs extend concepts found in ESNs by learning the recurrent connections through backpropagation through time (BPTT), rather than fixing the internal reservoir.

As RNNs are a more general form of architecture, this flexibility is encapsulated through the use of a class `RNN()`, with the following methods

Initialisation `__init__()` The RNN class is initialised with three main components: - Input Layer, which accepts the input states (u, x) for each subsequent time step - The recurrent layer models hidden state evolution, taking the input and previous hidden state to produce the current iteration, determined by

$$h_t = \sigma(W_{ih}x_t + W_{hh}h_{t-1} + b_h)$$

where σ is a nonlinear activation function, W_{hh} are recurrent connection weights (hidden-hidden), W_{ih} are input-hidden weights, and b_h contains bias values for the hidden layer. - The output layer or prediction layer, which linearly maps the hidden state to the output to calculate predictions

Forward Propagation `forward()`
This function dictates the data flow through the network over the input sequence. It is then passed to a nonlinear dropout function, before being processed by the output layer - mapped by the function

$$y_t = W_{ho}h_t + b_o$$

for W_{ho} being hidden-output weights, to compute predictions y_t.

Hidden State Initialisation `init_hidden()`
The hidden state initialisation method enhances the flexibility of the `RNN()` class, allowing initialisation of either a single hidden state zero tensor based on the batch size used in RNN and GRU architectures

$$h_0 = 0$$

, or a multiple cell and hidden state initialisation for LSTMs.

$$(h_0, c_0) = (0, 0)$$

For this architecture, the report chose to also explore nonlinear methods to improve complex temporal relationship learning, notably through activation functions σ in the forward method. The one applied here is a common function known as the Rectified Linear Unit (ReLU)

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

for sparse activation and improved gradient stability [14].
A dropout regulariser was also employed, which led to increased model generalisability through prior testing. The dropout layer is parameterised by a weight p, which acts to randomly deactivate a proportion of hidden states. This introduces stochastic variability and is a form of regularisation.

[1097]:
```
# RNN Setup
class RNN(nn.Module):
    # Initialisation
    def __init__(self, input_size=2, hidden_size=50, output_size=1,␣
 ↪rnn_layers=1,dropout_p = 0.3):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn_layers = rnn_layers
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True,␣
 ↪num_layers=rnn_layers)
        self.fc = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()
        self.dropout = nn.Dropout(p=dropout_p)

    # Forward method
    def forward(self, input_data, hidden):
        out, hidden = self.rnn(input_data, hidden)
        out = self.dropout(out)
        out = self.fc(out)
        #out = self.activation(out)
        return out, hidden

    # Hidden state initialisation
    def init_hidden(self, batch_size):
        if isinstance(self.rnn, nn.LSTM):
            # LSTM
            return (torch.zeros(self.rnn_layers, batch_size, self.hidden_size),
                    torch.zeros(self.rnn_layers, batch_size, self.hidden_size))
        else:
            # RNN and GRU
            return torch.zeros(self.rnn_layers, batch_size, self.hidden_size)
```

[1164]:
```
# Model Hyperparameters
#--------------------------#
input_size = 2                          # Dimension of the concatenated inputs␣
 ↪(u,x)
hidden_size = 64                        # Number of neurons in the hidden layer
output_size = 1                         # Output size - predicting a single␣
 ↪displacement value at each step
rnn_layers = 1                          # Number of stacked RNN layers
dropout_p = 0.3                         # Dropout weight
batch_size = input_data_train.shape[0] # Number of sequences processed␣
 ↪simultaneously
#--------------------------#

# RNN Model
```

```
model = RNN(input_size=input_size, hidden_size=hidden_size,␣
 ↪output_size=output_size, rnn_layers=rnn_layers)

# Hidden state initialisation
hidden = model.init_hidden(batch_size)
```

1.6.3 4.3 RNN Training

Hyperparameters

Hyperparameters are crucial or any ANN, as they directly influence the model's ability to learn and generalise from sequential data. Unlike ESNs, which rely on fixed internal dynamics and only train the output weight W_o, RNNs are fully flexible, making hyperparameter tuning even more critical.

Key hyperparameters determine the network's capacity, convergence rate, and generalisability. For instance, increasing the hidden size enhances the model's expressiveness but also raises the risk of overfitting and raises computational overhead. Similarly, a small learning rate ensures stable convergence but can slow down training, while higher rates may lead to instability. The flexible nature of RNNs makes them highly adaptable to various tasks, but also necessitates careful balancing of hyperparameters to prevent underfitting, overfitting, or inefficiencies during training. While this report explores the significance of these hyperparameters and their roles, excessive optimisation was not performed due to computational limitations, though theoretically can be executed in a similar manner as outlined Section 3.

[1105]:
```
# Training Hyperparameters
#--------------------------#
learning_rate = 0.0005                                                    ␣
 ↪          # Step size for updating model weights
n_epochs = 50                                                             ␣
 ↪          # Number of full passes over training set
criterion = nn.MSELoss()                                                  ␣
 ↪          # Loss function used - MSE
optimizer = optim.Adam(model.parameters(), lr=learning_rate)              ␣
 ↪          # Adam optimiser to update model weights
step_size = 30                                                            ␣
 ↪          # Scheduler steps before reducing learning rate
gamma = 0.25                                                              ␣
 ↪          # Decay factor for reducing learning rate
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=step_size,␣
 ↪gamma=gamma)  # Dynamically adjusts learning rate
lag_weight = 0.9                                                          ␣
 ↪          # Lag weight to smoothen dynamics and phase
#--------------------------#
```

Training Loop

The training loop implements an iterative process to optimise the RNN parameters for predicting sequential data. As follows;

1. The model is set to training mode using `model.train()`, and the hidden state is initialised at the start of each epoch to $h_0 = 0$ For each epoch up to `n_epochs`;
2. Each time series in the data set is analysed. The RNN processes the input, using this to update the hidden state using
$$h_t, y_t = RNN(x_t, h_{t-1})$$
where y_t is the predicted output and h_t the updated hidden state at time t
3. The MSE is calculated between the predicted and target sequences as the primary loss function, using least squares as shown below
$$\mathcal{L}_{\text{MSE}} = \frac{1}{T}\sum_{t=1}^{T} ||y_t - \hat{y}_t||_F^2$$
4. As a phase discrepancy was consistently apparent in training iterations, a lag penalty was introduced, using a regularisation formulation calculated as [15]
$$\mathcal{L}_{\text{lag}} = \frac{1}{T-1}\sum_{t=1}^{T-1} ||y_t - \hat{y}_{t+1}||_F^2$$
such that this is combined with the lag penalty weight, λ. Thus,
$$\mathcal{L} = \mathcal{L}_{\text{MSE}} + \lambda\mathcal{L}_{\text{lag}}$$
5. Gradients are computed for all model parameters $\nabla\mathcal{L}$ by backpropagation. These values are then clipped as per specification, to a maximum norm of 1:
$$||\hat{\nabla}_p|| = \min(||\nabla_p||, 1)$$
where p represents the gradients of parameter p
6. Parameters are then updated using the chosen optimiser - for the purposes of this report, the Adam optimiser was used [16]. A learning rate scheduler was also implemented to adjust the `learning_rate` by a specified `step_size` by the parameter γ.
7. The average gradient norm is calculated, and the total loss for the epoch is accumulated and returned - which is a measure of the RNN convergence over consecutive epochs

To monitor gradient clipping during training, average gradient norms `AGN` values were computed before and after clipping to identify the significance of the gradient being clipped. This also provided insight into the training stability and convergence while validating the model. Gradient norms were calculated using the formula
$$||\nabla|| = \sqrt{\sum_p ||\nabla_p||_F^2}$$

[1108]:
```
#Training RNN
def train_rnn(model, input_data, target_data, n_epochs=n_epochs,␣
↪learning_rate=learning_rate,criterion=criterion,optimizer=optimizer,scheduler=scheduler,␣
↪lag_weight=lag_weight):
```

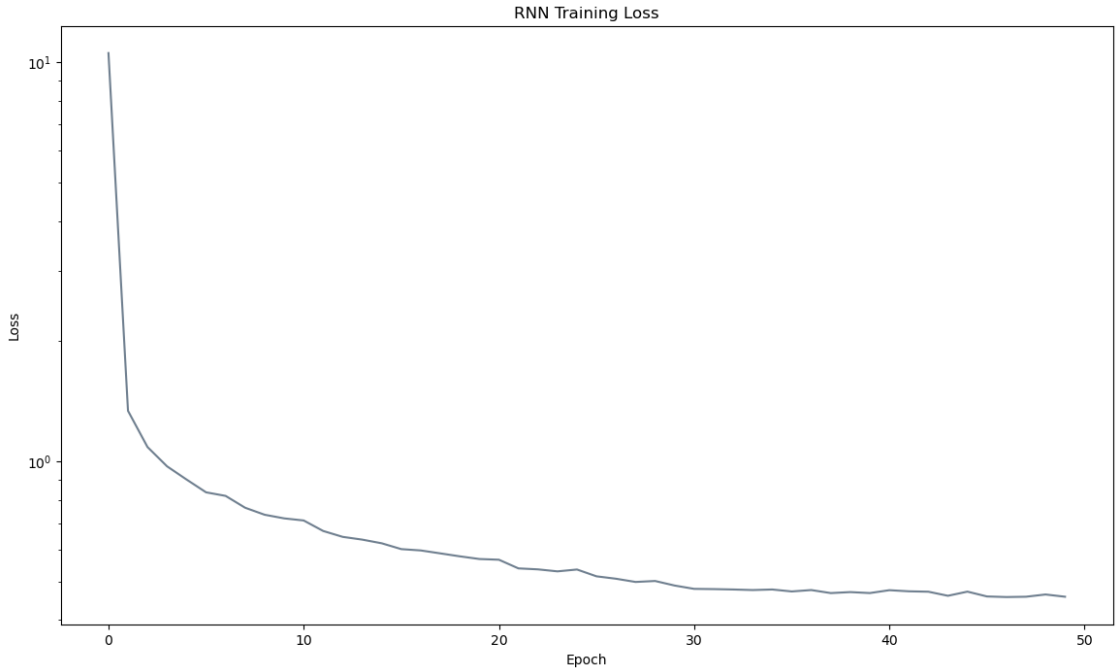
```
    # Track losses over epochs
    losses = []
    GN_before = []
    GN_after = []

    # Training loop
    for epoch in range(n_epochs):
        model.train()  # Set the model to training mode
        hidden = model.init_hidden(input_data.size(0))  # Initialize hidden␣
↪state for each epoch
        epoch_loss = 0.0

        for i in range(input_data.shape[0]):

            input_sequence = input_data[i].unsqueeze(0)
            target_sequence = target_data[i].unsqueeze(0)

            if i == 0:
                hidden = model.init_hidden(batch_size=1)

            # Forward pass
            output_pred, hidden = model(input_sequence, hidden)
            loss = criterion(output_pred, target_sequence)

            # Lag penalty
            if target_sequence.shape[1] > 1:
                lag_penalty = torch.mean((output_pred[:, :-1, :] -␣
↪target_sequence[:, 1:, :]) ** 2)
                total_loss = loss + lag_weight * lag_penalty
            else:
                total_loss = loss

            # Detach hidden state
            if isinstance(hidden, tuple):
                hidden = tuple(h.detach() for h in hidden)
            else:
                hidden = hidden.detach()

            # Backward pass
            optimizer.zero_grad()
            total_loss.backward()

            total_GN_before = torch.norm(torch.stack([torch.norm(p.grad.
↪detach()) for p in model.parameters() if p.grad is not None]))
            GN_before.append(total_GN_before.item())

            # Apply gradient clipping
```

```
            nn.utils.clip_grad_norm_(model.parameters(), max_norm=1)

            total_GN_after = torch.norm(torch.stack([torch.norm(p.grad.detach())␣
↪for p in model.parameters() if p.grad is not None]))
            GN_after.append(total_GN_after.item())

            # Update weights
            optimizer.step()

            # Accumulate loss for this sequence
            epoch_loss += total_loss.item()

        # Step the learning rate scheduler
        scheduler.step()

        losses.append(epoch_loss)

        AGN_before = sum(GN_before)/len(GN_before)
        AGN_after = sum(GN_after)/len(GN_after)

        print(f"Epoch {epoch+1}/{n_epochs}, Loss: {epoch_loss:.4f}, AGN Before:␣
↪{AGN_before:.4f}, AGN After: {AGN_after:.4f}")

        GN_before.clear()
        GN_after.clear()

    return losses

# Train RNN
losses = train_rnn(model, input_data_train,␣
↪output_data_train,learning_rate=learning_rate,criterion=criterion,optimizer=optimizer,schedul
```

```
Epoch 1/50, Loss: 10.5353, AGN Before: 0.6179, AGN After: 0.4746
Epoch 2/50, Loss: 1.3374, AGN Before: 0.2181, AGN After: 0.2181
Epoch 3/50, Loss: 1.0851, AGN Before: 0.1495, AGN After: 0.1495
Epoch 4/50, Loss: 0.9709, AGN Before: 0.1307, AGN After: 0.1307
Epoch 5/50, Loss: 0.8996, AGN Before: 0.1076, AGN After: 0.1076
Epoch 6/50, Loss: 0.8355, AGN Before: 0.1077, AGN After: 0.1077
Epoch 7/50, Loss: 0.8186, AGN Before: 0.1250, AGN After: 0.1250
Epoch 8/50, Loss: 0.7645, AGN Before: 0.1061, AGN After: 0.1061
Epoch 9/50, Loss: 0.7342, AGN Before: 0.0965, AGN After: 0.0965
Epoch 10/50, Loss: 0.7190, AGN Before: 0.0925, AGN After: 0.0925
Epoch 11/50, Loss: 0.7103, AGN Before: 0.1187, AGN After: 0.1187
Epoch 12/50, Loss: 0.6685, AGN Before: 0.0764, AGN After: 0.0764
Epoch 13/50, Loss: 0.6462, AGN Before: 0.0961, AGN After: 0.0961
Epoch 14/50, Loss: 0.6360, AGN Before: 0.0736, AGN After: 0.0736
Epoch 15/50, Loss: 0.6227, AGN Before: 0.0820, AGN After: 0.0820
```

```
Epoch 16/50, Loss: 0.6021, AGN Before: 0.0855, AGN After: 0.0855
Epoch 17/50, Loss: 0.5975, AGN Before: 0.0751, AGN After: 0.0751
Epoch 18/50, Loss: 0.5875, AGN Before: 0.0653, AGN After: 0.0653
Epoch 19/50, Loss: 0.5776, AGN Before: 0.0816, AGN After: 0.0816
Epoch 20/50, Loss: 0.5687, AGN Before: 0.0633, AGN After: 0.0633
Epoch 21/50, Loss: 0.5664, AGN Before: 0.0649, AGN After: 0.0649
Epoch 22/50, Loss: 0.5387, AGN Before: 0.0575, AGN After: 0.0575
Epoch 23/50, Loss: 0.5357, AGN Before: 0.0605, AGN After: 0.0605
Epoch 24/50, Loss: 0.5296, AGN Before: 0.0721, AGN After: 0.0721
Epoch 25/50, Loss: 0.5353, AGN Before: 0.0717, AGN After: 0.0717
Epoch 26/50, Loss: 0.5146, AGN Before: 0.0549, AGN After: 0.0549
Epoch 27/50, Loss: 0.5075, AGN Before: 0.0601, AGN After: 0.0601
Epoch 28/50, Loss: 0.4982, AGN Before: 0.0501, AGN After: 0.0501
Epoch 29/50, Loss: 0.5010, AGN Before: 0.0490, AGN After: 0.0490
Epoch 30/50, Loss: 0.4880, AGN Before: 0.0508, AGN After: 0.0508
Epoch 31/50, Loss: 0.4787, AGN Before: 0.0412, AGN After: 0.0412
Epoch 32/50, Loss: 0.4781, AGN Before: 0.0413, AGN After: 0.0413
Epoch 33/50, Loss: 0.4770, AGN Before: 0.0404, AGN After: 0.0404
Epoch 34/50, Loss: 0.4755, AGN Before: 0.0412, AGN After: 0.0412
Epoch 35/50, Loss: 0.4770, AGN Before: 0.0385, AGN After: 0.0385
Epoch 36/50, Loss: 0.4716, AGN Before: 0.0395, AGN After: 0.0395
Epoch 37/50, Loss: 0.4755, AGN Before: 0.0396, AGN After: 0.0396
Epoch 38/50, Loss: 0.4672, AGN Before: 0.0400, AGN After: 0.0400
Epoch 39/50, Loss: 0.4699, AGN Before: 0.0367, AGN After: 0.0367
Epoch 40/50, Loss: 0.4673, AGN Before: 0.0406, AGN After: 0.0406
Epoch 41/50, Loss: 0.4751, AGN Before: 0.0364, AGN After: 0.0364
Epoch 42/50, Loss: 0.4720, AGN Before: 0.0384, AGN After: 0.0384
Epoch 43/50, Loss: 0.4709, AGN Before: 0.0410, AGN After: 0.0410
Epoch 44/50, Loss: 0.4600, AGN Before: 0.0362, AGN After: 0.0362
Epoch 45/50, Loss: 0.4711, AGN Before: 0.0433, AGN After: 0.0433
Epoch 46/50, Loss: 0.4582, AGN Before: 0.0389, AGN After: 0.0389
Epoch 47/50, Loss: 0.4566, AGN Before: 0.0362, AGN After: 0.0362
Epoch 48/50, Loss: 0.4575, AGN Before: 0.0395, AGN After: 0.0395
Epoch 49/50, Loss: 0.4634, AGN Before: 0.0380, AGN After: 0.0380
Epoch 50/50, Loss: 0.4575, AGN Before: 0.0373, AGN After: 0.0373
```

[1149]:
```python
#Visualisation of Loss against Epochs - RNN
plt.figure(figsize=(14, 8))
plt.axes().set_yscale('log')
plt.plot(range(n_epochs),losses, linestyle='-',color='slategrey')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("RNN Training Loss")
plt.show()
```



The plot of MSE loss against the number of epochs for an RNN is shown above. The loss decreases rapidly in the initial 2-3 epochs, indicating efficient learning of the underlying dynamics. This is followed by a gradual reduction and indication of stabilisation, suggesting convergence to an optimal solution. The absence of fluctuations in the later epochs implies consistent stable learning and minimal overfitting.

1.6.4 4.4 RNN Testing

The RNN performance was be tested by repeatedly refeeding output trajectories into models as inputs - referred to as the process of "rolling out" the model, while retaining memory of states in previous iterations [17]. These predicted trajectories can be compared against the original model and its loss can be validated.

Testing Hyperparameters
Two testing hyperparameters were specified for the unrolling procedure;

- `n_warmup` dictates the number of initial time steps to process known data. As the hidden state is initialised as zero tensors, this can negatively affect the outputs. As a solution, the RNN model hidden states are initialised using known states using the known test sequence data before transitioning to autoregressive means
- `alpha` is a weight factor, added for this to improve the stability of the model and show improvements in predictability by weighing the contribution of predicted and actual input values by the function [18]

$$\text{Input} = \alpha(\text{actual}) + (1 - \alpha)(\text{predicted})$$

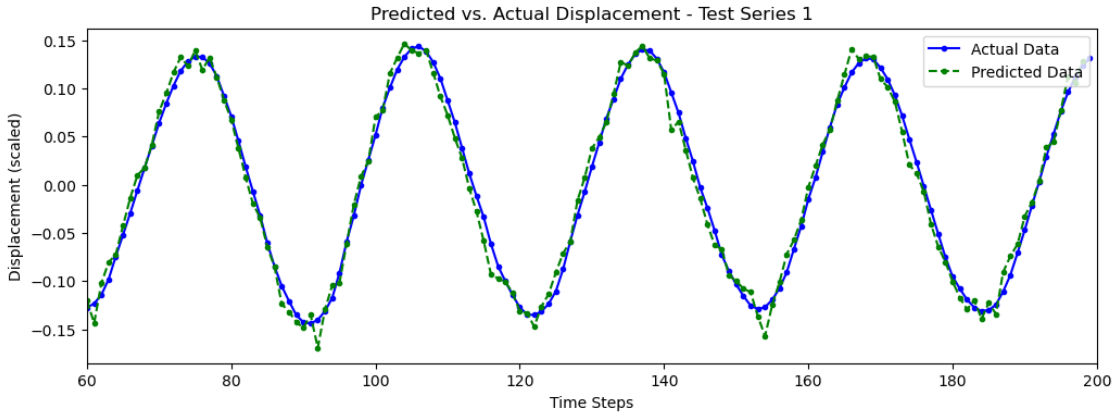
[1226]:
```
# Testing Parameters
#--------------------------#
n_warmup = 60         # Number of time steps to warm-up the model
alpha = 0.2           # Weight factor
#--------------------------#
```

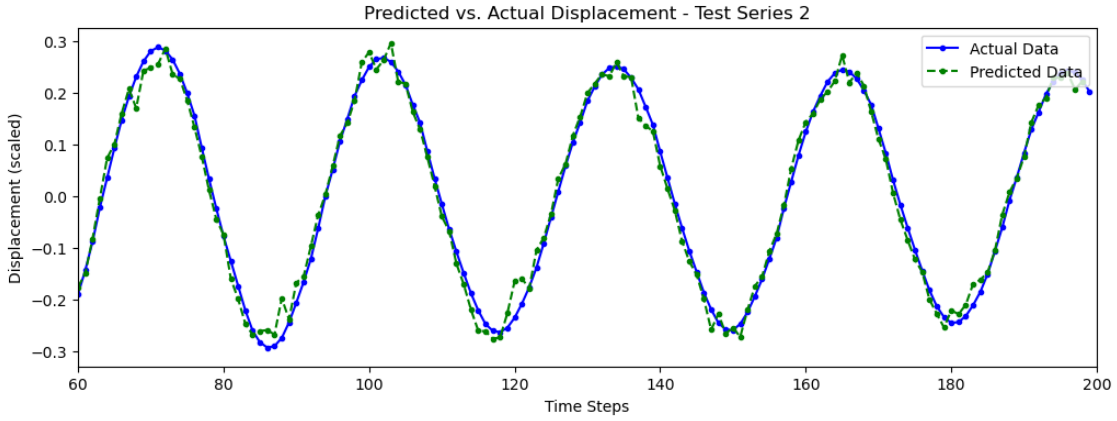
[1115]:
```
# Predict loops and time steps by iterating and feeding back predictions
all_predictions = []
mse_list = []
timeseries = []

with torch.no_grad():

    # Iteration over testing sets
    for i in range(input_data_test.shape[0]):
        test_sequence = input_data_test[i]
        test_sequence = test_sequence.clone().detach()
        actual_data = output_data_test[i]

        warmup_input = test_sequence[:n_warmup].unsqueeze(0)
        timeseries = []

        # Hidden state initialisation
        hidden = model.init_hidden(batch_size=1)

        # Warm-up phase: Initialize hidden state
        next_state, hidden = model(warmup_input, hidden)
        input = next_state[:, -1, :].unsqueeze(0)

        # Predict remaining time steps
        for i in range(len(actual_data) - n_warmup - 1):
            # Extract u feature for the current time step
            u_feature = test_sequence[n_warmup + i, 1].view(1, 1, 1)

            # (x,u) into input
            input = alpha * torch.cat((test_sequence[n_warmup + i, 0].view(1, 1,␣
↪1), u_feature), dim=-1) + (1 - alpha) * input

            # Predict the next state and update hidden state
            input, hidden = model(input, hidden)

            # Append prediction to timeseries
            timeseries.append(input.squeeze(0).numpy())

        timeseries = np.array(timeseries)
        all_predictions.append(timeseries)
```

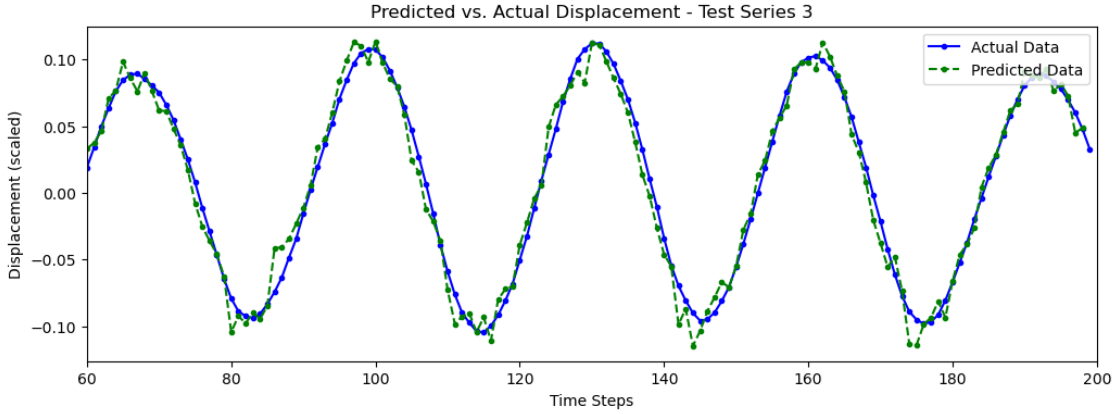
```
        # MSE
        mse = np.mean((actual_data[n_warmup:-1].numpy() - timeseries[:, 0]) ** 2)
        mse_list.append(mse)

# Average MSE over all test sequences
average_mse = np.mean(mse_list)
print(f"Average MSE on Test Set: {average_mse:.6f}")
```

```
Average MSE on Test Set: 0.001790
```

[1222]:
```
# Plot the predicted vs actual time series for the first test sequence
i_data = [0,1,2]  # Index of the sequence to plot
for i in i_data:
    plt.figure(figsize=(12, 4))
    t = np.arange(len(output_data_test[i]))
    plt.plot(t, output_data_test[i].numpy(), label="Actual Data", color="blue",␣
 ↪linestyle="-", marker=".")
    plt.plot(t[n_warmup:n_warmup + len(all_predictions[i])], all_predictions[i][:
 ↪, 0] / scaling, label="Predicted Data", color="green", linestyle="--",␣
 ↪marker=".")
    plt.xlabel("Time Steps")
    plt.ylabel("Displacement (scaled)")
    plt.legend(loc="upper right")
    plt.xlim(n_warmup, len(actual_data))
    plt.title(f"Predicted vs. Actual Displacement - Test Series {i+1}")
    plt.show()
```







The RNN was evaluated on the dataset for a subset of example test series. The model achieved an average MSE of 0.00179 on the test set, indicating good predictive accuracy. The model accurately captures the oscillatory behaviour, including amplitude, frequency, and phase alignment, even when given unseen sequences. Minor discrepancies, such as slight phase shifts or small amplitude mismatches, are negligible and do not significantly affect the overall performance. The RNN effectively learns the nonlinear temporal dependencies inherent in the NTMD system and maintains generalisability over the displayed data sets.

It should be noted that due to the large warmup and α hyperparameters chosen and truncated data set, the prediction states are relatively small sequences which may be too small to demonstrate significant nonlinearities, with significant influence from existing signal states.

1.6.5 4.5 GRU

`nn.GRU()`

The Gated Retrieval Unit (GRU) architecture enhances the standard RNN by introducing gates to manage information flow, improving its ability to handle long-term dependencies and mitigating vanishing gradient problems. The key difference in architecture is the use of gates, which abstractively perform selective memory updates of relevant information. Standard RNNs update

hidden states directly. GRUs however, possess two different gates: - reset gates: $r_t = σ(W_r ·[h_{t−1}, x_t] + b_r)$ − $updategatesz_t = σ(W_z · [h_{t−1}, x_t] + b_z)$
where $σ$ is the sigmoid activation function, and $[h_{t−1}, x_t]$ denotes a concatenation, and b_r, b_z are biases. These gates are integral in the formulation of the hidden state update for h_t [19].

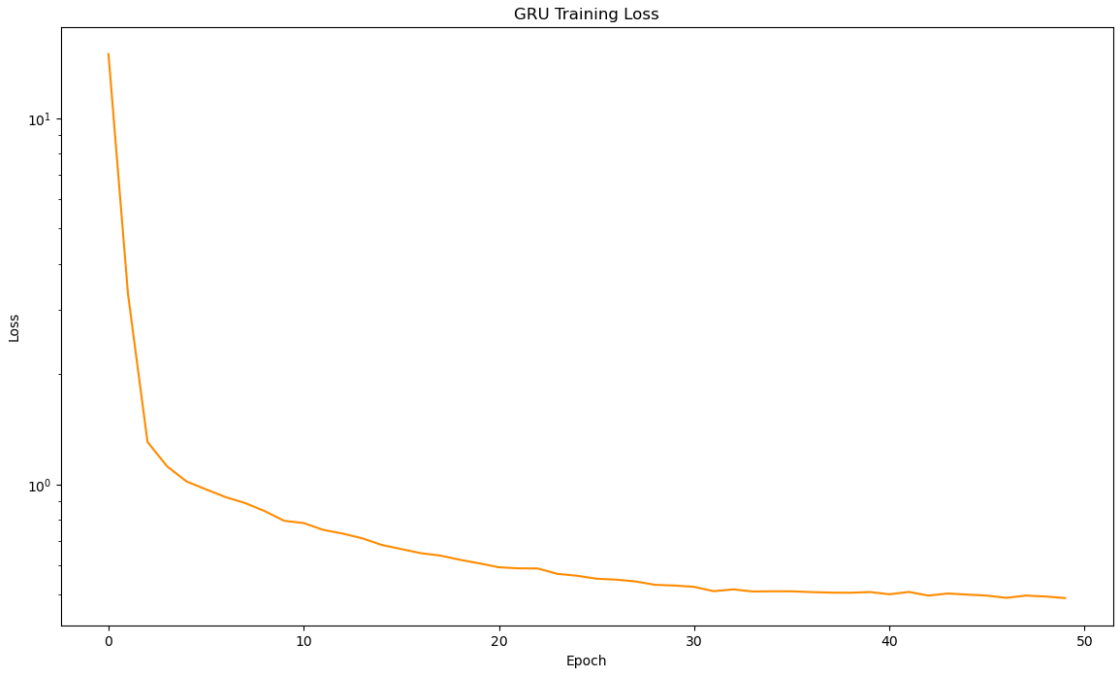
[1157]:
```
# GRU Setup
class GRU(nn.Module):
    # Initialisation
    def __init__(self, input_size=2, hidden_size=50, output_size=1,␣
↪rnn_layers=1,dropout_p = 0.3):
        super(GRU, self).__init__()
        self.hidden_size = hidden_size
        self.rnn_layers = rnn_layers
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True,␣
↪num_layers=rnn_layers)
        self.fc = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()
        self.dropout = nn.Dropout(p=dropout_p)

    # Forward method
    def forward(self, input_data, hidden):
        out, hidden = self.gru(input_data, hidden)
        out = self.dropout(out)
        out = self.fc(out)
        # out = self.activation(out)
        return out, hidden

    # Hidden state initialisation
    def init_hidden(self, batch_size):
        return torch.zeros(self.rnn_layers, batch_size, self.hidden_size)
```

[1125]:
```
# Initialise the GRU model
gru_model = GRU(input_size=input_size, hidden_size=hidden_size,␣
↪output_size=output_size, rnn_layers=rnn_layers)

# Use the same hyperparameters as the RNN
gru_optimizer = optim.Adam(gru_model.parameters(), lr=learning_rate)
gru_scheduler = torch.optim.lr_scheduler.StepLR(gru_optimizer,␣
↪step_size=step_size, gamma=gamma)

# Train GRU
gru_losses = train_rnn(gru_model, input_data_train, output_data_train,
                       n_epochs=n_epochs, learning_rate=learning_rate,
                       criterion=criterion, optimizer=gru_optimizer,
                       scheduler=gru_scheduler, lag_weight=lag_weight)
```

```
Epoch 1/50, Loss: 14.9677, AGN Before: 0.5163, AGN After: 0.4621
Epoch 2/50, Loss: 3.3076, AGN Before: 0.2860, AGN After: 0.2860
```

```
Epoch 3/50, Loss: 1.3048, AGN Before: 0.1263, AGN After: 0.1263
Epoch 4/50, Loss: 1.1192, AGN Before: 0.0948, AGN After: 0.0948
Epoch 5/50, Loss: 1.0175, AGN Before: 0.0835, AGN After: 0.0835
Epoch 6/50, Loss: 0.9679, AGN Before: 0.0791, AGN After: 0.0791
Epoch 7/50, Loss: 0.9221, AGN Before: 0.0754, AGN After: 0.0754
Epoch 8/50, Loss: 0.8882, AGN Before: 0.0620, AGN After: 0.0620
Epoch 9/50, Loss: 0.8446, AGN Before: 0.0632, AGN After: 0.0632
Epoch 10/50, Loss: 0.7945, AGN Before: 0.0671, AGN After: 0.0671
Epoch 11/50, Loss: 0.7831, AGN Before: 0.0634, AGN After: 0.0634
Epoch 12/50, Loss: 0.7504, AGN Before: 0.0636, AGN After: 0.0636
Epoch 13/50, Loss: 0.7330, AGN Before: 0.0699, AGN After: 0.0699
Epoch 14/50, Loss: 0.7114, AGN Before: 0.0577, AGN After: 0.0577
Epoch 15/50, Loss: 0.6824, AGN Before: 0.0584, AGN After: 0.0584
Epoch 16/50, Loss: 0.6649, AGN Before: 0.0605, AGN After: 0.0605
Epoch 17/50, Loss: 0.6480, AGN Before: 0.0580, AGN After: 0.0580
Epoch 18/50, Loss: 0.6381, AGN Before: 0.0581, AGN After: 0.0581
Epoch 19/50, Loss: 0.6216, AGN Before: 0.0612, AGN After: 0.0612
Epoch 20/50, Loss: 0.6078, AGN Before: 0.0583, AGN After: 0.0583
Epoch 21/50, Loss: 0.5930, AGN Before: 0.0581, AGN After: 0.0581
Epoch 22/50, Loss: 0.5891, AGN Before: 0.0544, AGN After: 0.0544
Epoch 23/50, Loss: 0.5884, AGN Before: 0.0563, AGN After: 0.0563
Epoch 24/50, Loss: 0.5688, AGN Before: 0.0632, AGN After: 0.0632
Epoch 25/50, Loss: 0.5622, AGN Before: 0.0552, AGN After: 0.0552
Epoch 26/50, Loss: 0.5518, AGN Before: 0.0424, AGN After: 0.0424
Epoch 27/50, Loss: 0.5486, AGN Before: 0.0442, AGN After: 0.0442
Epoch 28/50, Loss: 0.5424, AGN Before: 0.0572, AGN After: 0.0572
Epoch 29/50, Loss: 0.5310, AGN Before: 0.0479, AGN After: 0.0479
Epoch 30/50, Loss: 0.5287, AGN Before: 0.0522, AGN After: 0.0522
Epoch 31/50, Loss: 0.5243, AGN Before: 0.0339, AGN After: 0.0339
Epoch 32/50, Loss: 0.5101, AGN Before: 0.0380, AGN After: 0.0380
Epoch 33/50, Loss: 0.5160, AGN Before: 0.0370, AGN After: 0.0370
Epoch 34/50, Loss: 0.5091, AGN Before: 0.0329, AGN After: 0.0329
Epoch 35/50, Loss: 0.5098, AGN Before: 0.0338, AGN After: 0.0338
Epoch 36/50, Loss: 0.5097, AGN Before: 0.0324, AGN After: 0.0324
Epoch 37/50, Loss: 0.5073, AGN Before: 0.0364, AGN After: 0.0364
Epoch 38/50, Loss: 0.5057, AGN Before: 0.0331, AGN After: 0.0331
Epoch 39/50, Loss: 0.5051, AGN Before: 0.0343, AGN After: 0.0343
Epoch 40/50, Loss: 0.5074, AGN Before: 0.0351, AGN After: 0.0351
Epoch 41/50, Loss: 0.5004, AGN Before: 0.0368, AGN After: 0.0368
Epoch 42/50, Loss: 0.5078, AGN Before: 0.0367, AGN After: 0.0367
Epoch 43/50, Loss: 0.4962, AGN Before: 0.0348, AGN After: 0.0348
Epoch 44/50, Loss: 0.5028, AGN Before: 0.0377, AGN After: 0.0377
Epoch 45/50, Loss: 0.4992, AGN Before: 0.0346, AGN After: 0.0346
Epoch 46/50, Loss: 0.4961, AGN Before: 0.0350, AGN After: 0.0350
Epoch 47/50, Loss: 0.4892, AGN Before: 0.0349, AGN After: 0.0349
Epoch 48/50, Loss: 0.4963, AGN Before: 0.0367, AGN After: 0.0367
Epoch 49/50, Loss: 0.4933, AGN Before: 0.0320, AGN After: 0.0320
Epoch 50/50, Loss: 0.4886, AGN Before: 0.0335, AGN After: 0.0335
```

[1147]:
```
#Visualisation of Loss against Epochs - GRU
plt.figure(figsize=(14, 8))
plt.axes().set_yscale('log')
plt.plot(range(n_epochs),gru_losses, linestyle='-',color = 'darkorange')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("GRU Training Loss")
plt.show()
```



[1162]:
```
# Evaluate GRU performance on the test set
gru_all_predictions = []
gru_mse_list = []

with torch.no_grad():
    # Loop for data set
    for i in range(input_data_test.shape[0]):
        test_sequence = input_data_test[i]
        test_sequence = scaling * test_sequence.clone().detach()
        actual_data = output_data_test[i]

        # Warmup phase
        warmup_input = test_sequence[:n_warmup].unsqueeze(0)

        # Hidden state initialisation
        gru_timeseries = []
```

```
        hidden = gru_model.init_hidden(batch_size=1)

        # Warm-up phase
        next_state, hidden = gru_model(warmup_input, hidden)
        input = next_state[:, -1, :].unsqueeze(0)

        # Predict remaining time steps
        for i in range(len(actual_data) - n_warmup - 1):
            u_feature = test_sequence[n_warmup + i, 1].view(1, 1, 1)
            input = alpha * torch.cat((test_sequence[n_warmup + i, 0].view(1, 1,␣
↪1), u_feature), dim=-1) + (1 - alpha) * input
            input, hidden = gru_model(input, hidden)
            gru_timeseries.append(input.squeeze(0).numpy())

        gru_timeseries = np.array(gru_timeseries)
        gru_all_predictions.append(gru_timeseries)

        # MSE
        mse = np.mean((actual_data[n_warmup:-1].numpy() - gru_timeseries[:, 0])␣
↪** 2)
        gru_mse_list.append(mse)

# Average MSE for GRU
gru_average_mse = np.mean(gru_mse_list)
```

1.6.6 4.6 LSTM

`nn.LSTM()`
In similar fashion to the GRU, the Long Short Term Memory Unit (LSTM) networks are designed to address issues like vanishing and exploding gradients while effectively learning long-term dependencies. It is more computationally expensive than both GRUs and traditional RNNs, owing to their use of three different gates which also serve to selectively regulate information flow: - Input gates: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$ $- Forget gates f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$ - Output gates: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
in addition to gates, LSTM differs from GRU by maintaining a hidden state and an internal cell state to increase memory retention - which is evolved by

$$c_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

The modification to the `init_hidden()` method accounts for the cell state being initialised as another tuple of zero tensors. where σ is the sigmoid activation function, and $[h_{t-1}, x_t]$ denotes a concatenation, and b_i, b_f, b_o are biases[19]

[1131]:
```
# LSTM Setup
class LSTM(nn.Module):
    # Initialisation
    def __init__(self, input_size=2, hidden_size=50, output_size=1,␣
↪rnn_layers=1,dropout_p = 0.3):
```

```
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.rnn_layers = rnn_layers
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True,␣
↪num_layers=rnn_layers)  # LSTM layer
        self.fc = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()
        self.dropout = nn.Dropout(p=dropout_p)

    # Forward method
    def forward(self, input_data, hidden):
        out, hidden = self.lstm(input_data, hidden)
        out = self.dropout(out)
        out = self.fc(out)
        # out = self.activation(out)
        return out, hidden

    # Hidden and cell state initialisation
    def init_hidden(self, batch_size):

        return (torch.zeros(self.rnn_layers, batch_size, self.hidden_size),
                torch.zeros(self.rnn_layers, batch_size, self.hidden_size))
```

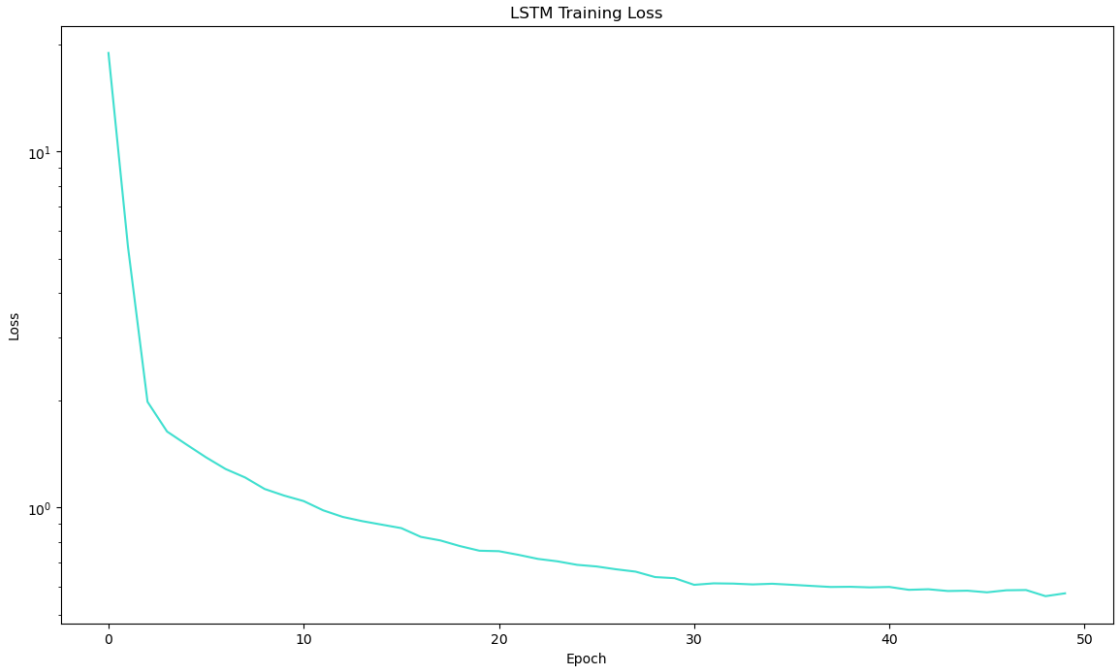
[1133]:
```
# Initialise the LSTM model
lstm_model = LSTM(input_size=input_size, hidden_size=hidden_size,␣
↪output_size=output_size, rnn_layers=rnn_layers)

# Use the same hyperparameters as the RNN
lstm_optimizer = optim.Adam(lstm_model.parameters(), lr=learning_rate)
lstm_scheduler = torch.optim.lr_scheduler.StepLR(lstm_optimizer,␣
↪step_size=step_size, gamma=gamma)

# Train LSTM
lstm_losses = train_rnn(lstm_model, input_data_train, output_data_train,
                        n_epochs=n_epochs, learning_rate=learning_rate,
                        criterion=criterion, optimizer=lstm_optimizer,
                        scheduler=lstm_scheduler, lag_weight=lag_weight)
```

```
Epoch 1/50, Loss: 18.8997, AGN Before: 0.3556, AGN After: 0.3556
Epoch 2/50, Loss: 5.3927, AGN Before: 0.4113, AGN After: 0.3796
Epoch 3/50, Loss: 1.9808, AGN Before: 0.2036, AGN After: 0.2036
Epoch 4/50, Loss: 1.6342, AGN Before: 0.1369, AGN After: 0.1369
Epoch 5/50, Loss: 1.5019, AGN Before: 0.1255, AGN After: 0.1255
Epoch 6/50, Loss: 1.3827, AGN Before: 0.1200, AGN After: 0.1200
Epoch 7/50, Loss: 1.2822, AGN Before: 0.1097, AGN After: 0.1097
Epoch 8/50, Loss: 1.2139, AGN Before: 0.1051, AGN After: 0.1051
Epoch 9/50, Loss: 1.1270, AGN Before: 0.0941, AGN After: 0.0941
```

```
Epoch 10/50, Loss: 1.0798, AGN Before: 0.1050, AGN After: 0.1050
Epoch 11/50, Loss: 1.0421, AGN Before: 0.0994, AGN After: 0.0994
Epoch 12/50, Loss: 0.9816, AGN Before: 0.0929, AGN After: 0.0929
Epoch 13/50, Loss: 0.9408, AGN Before: 0.0977, AGN After: 0.0977
Epoch 14/50, Loss: 0.9154, AGN Before: 0.1042, AGN After: 0.1042
Epoch 15/50, Loss: 0.8949, AGN Before: 0.1046, AGN After: 0.1046
Epoch 16/50, Loss: 0.8750, AGN Before: 0.0945, AGN After: 0.0945
Epoch 17/50, Loss: 0.8268, AGN Before: 0.0862, AGN After: 0.0862
Epoch 18/50, Loss: 0.8079, AGN Before: 0.0914, AGN After: 0.0914
Epoch 19/50, Loss: 0.7791, AGN Before: 0.0880, AGN After: 0.0880
Epoch 20/50, Loss: 0.7560, AGN Before: 0.0881, AGN After: 0.0881
Epoch 21/50, Loss: 0.7537, AGN Before: 0.0952, AGN After: 0.0952
Epoch 22/50, Loss: 0.7362, AGN Before: 0.0921, AGN After: 0.0921
Epoch 23/50, Loss: 0.7170, AGN Before: 0.0891, AGN After: 0.0891
Epoch 24/50, Loss: 0.7058, AGN Before: 0.0833, AGN After: 0.0833
Epoch 25/50, Loss: 0.6901, AGN Before: 0.0917, AGN After: 0.0917
Epoch 26/50, Loss: 0.6828, AGN Before: 0.0891, AGN After: 0.0891
Epoch 27/50, Loss: 0.6705, AGN Before: 0.0805, AGN After: 0.0805
Epoch 28/50, Loss: 0.6603, AGN Before: 0.0766, AGN After: 0.0766
Epoch 29/50, Loss: 0.6375, AGN Before: 0.0717, AGN After: 0.0717
Epoch 30/50, Loss: 0.6330, AGN Before: 0.0716, AGN After: 0.0716
Epoch 31/50, Loss: 0.6064, AGN Before: 0.0495, AGN After: 0.0495
Epoch 32/50, Loss: 0.6122, AGN Before: 0.0493, AGN After: 0.0493
Epoch 33/50, Loss: 0.6112, AGN Before: 0.0569, AGN After: 0.0569
Epoch 34/50, Loss: 0.6079, AGN Before: 0.0520, AGN After: 0.0520
Epoch 35/50, Loss: 0.6106, AGN Before: 0.0591, AGN After: 0.0591
Epoch 36/50, Loss: 0.6065, AGN Before: 0.0507, AGN After: 0.0507
Epoch 37/50, Loss: 0.6022, AGN Before: 0.0546, AGN After: 0.0546
Epoch 38/50, Loss: 0.5980, AGN Before: 0.0473, AGN After: 0.0473
Epoch 39/50, Loss: 0.5987, AGN Before: 0.0509, AGN After: 0.0509
Epoch 40/50, Loss: 0.5964, AGN Before: 0.0553, AGN After: 0.0553
Epoch 41/50, Loss: 0.5981, AGN Before: 0.0525, AGN After: 0.0525
Epoch 42/50, Loss: 0.5871, AGN Before: 0.0508, AGN After: 0.0508
Epoch 43/50, Loss: 0.5893, AGN Before: 0.0550, AGN After: 0.0550
Epoch 44/50, Loss: 0.5826, AGN Before: 0.0524, AGN After: 0.0524
Epoch 45/50, Loss: 0.5838, AGN Before: 0.0573, AGN After: 0.0573
Epoch 46/50, Loss: 0.5777, AGN Before: 0.0520, AGN After: 0.0520
Epoch 47/50, Loss: 0.5854, AGN Before: 0.0570, AGN After: 0.0570
Epoch 48/50, Loss: 0.5863, AGN Before: 0.0545, AGN After: 0.0545
Epoch 49/50, Loss: 0.5635, AGN Before: 0.0514, AGN After: 0.0514
Epoch 50/50, Loss: 0.5739, AGN Before: 0.0548, AGN After: 0.0548
```

[1145]:
```
#Visualisation of Loss against Epochs - LSTM
plt.figure(figsize=(14, 8))
plt.axes().set_yscale('log')
plt.plot(range(n_epochs),lstm_losses, linestyle='-',color = 'turquoise')
plt.xlabel("Epoch")
```

```
plt.ylabel("Loss")
plt.title("LSTM Training Loss")
plt.show()
```



[1137]:
```
# Evaluate LSTM performance on the test set
lstm_all_predictions = []
lstm_mse_list = []

with torch.no_grad():
    # Loop for data set
    for i in range(input_data_test.shape[0]):
        test_sequence = input_data_test[i]
        test_sequence = scaling * test_sequence.clone().detach()
        actual_data = output_data_test[i]

        # Warmup phase
        warmup_input = test_sequence[:n_warmup].unsqueeze(0)

        # Hidden and cell state initialisation
        hidden = lstm_model.init_hidden(batch_size=1)

        # Warm-up phase
        next_state, hidden = lstm_model(warmup_input, hidden)
        input = next_state[:, -1, :].unsqueeze(0)
```

```
        # Predict remaining time steps
        lstm_timeseries = []
        for i in range(len(actual_data) - n_warmup - 1):
            u_feature = test_sequence[n_warmup + i, 1].view(1, 1, 1)
            input = alpha * torch.cat((test_sequence[n_warmup + i, 0].view(1, 1,␣
↪1), u_feature), dim=-1) + (1 - alpha) * input
            input, hidden = lstm_model(input, hidden)
            lstm_timeseries.append(input.squeeze(0).numpy())

        lstm_timeseries = np.array(lstm_timeseries)
        lstm_all_predictions.append(lstm_timeseries)

        # MSE
        mse = np.mean((actual_data[n_warmup:-1].numpy() - lstm_timeseries[:, 0])␣
↪** 2)
        lstm_mse_list.append(mse)

# Average MSE for LSTM
lstm_average_mse = np.mean(lstm_mse_list)
```

1.6.7 4.7 Performance Validation

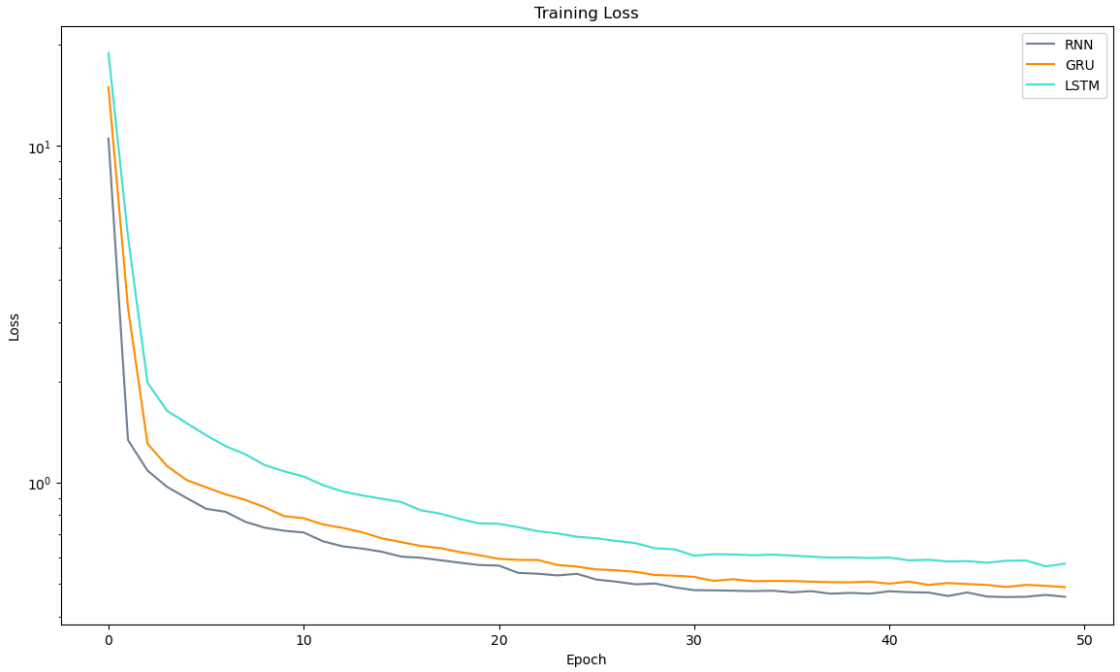
Comparison of RNN Architectures

The section below comprehensively compares the performance of the three RNN architectures by their testing prediction MSE and convergence plots.

[324]:
```
print(f"RNN Average MSE on Test Set: {average_mse:.6f}")
print(f"GRU Average MSE on Test Set: {gru_average_mse:.6f}")
print(f"LSTM Average MSE on Test Set: {lstm_average_mse:.6f}")
```

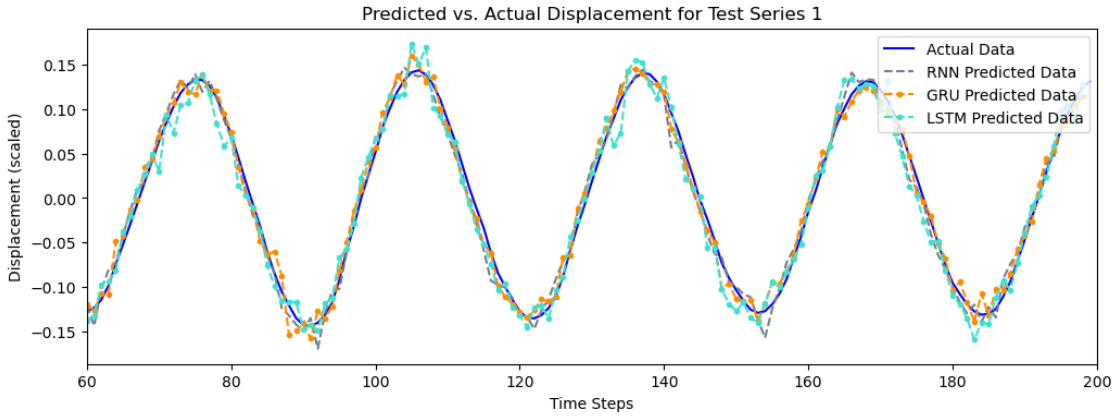
```
RNN Average MSE on Test Set: 0.001019
GRU Average MSE on Test Set: 0.000920
LSTM Average MSE on Test Set: 0.001274
```

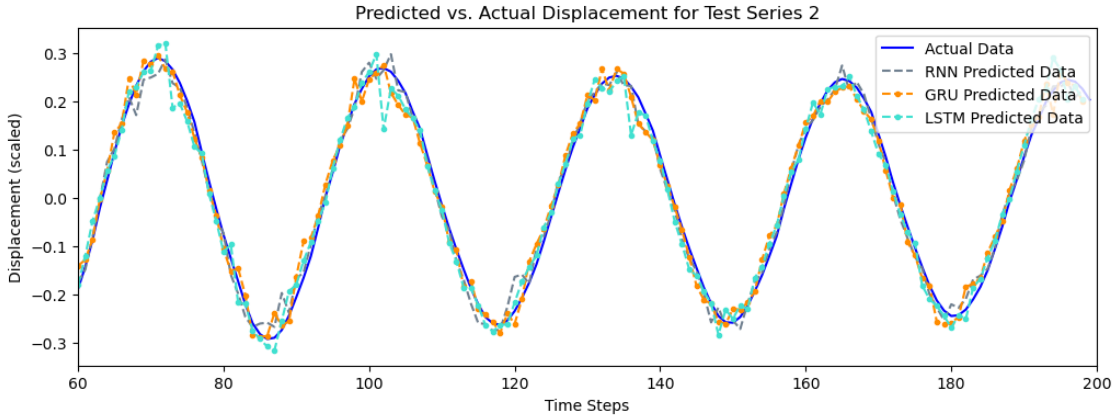
[1239]:
```
#Visualisation of Loss against Epochs
plt.figure(figsize=(14, 8))
plt.axes().set_yscale('log')
plt.plot(range(n_epochs),losses, linestyle='-',color ='slategrey',label = "RNN")
plt.plot(range(n_epochs),gru_losses, linestyle='-',color ='darkorange',label =␣
↪"GRU")
plt.plot(range(n_epochs),lstm_losses, linestyle='-',color = 'turquoise',label␣
↪="LSTM")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title("Training Loss")
plt.show()
```

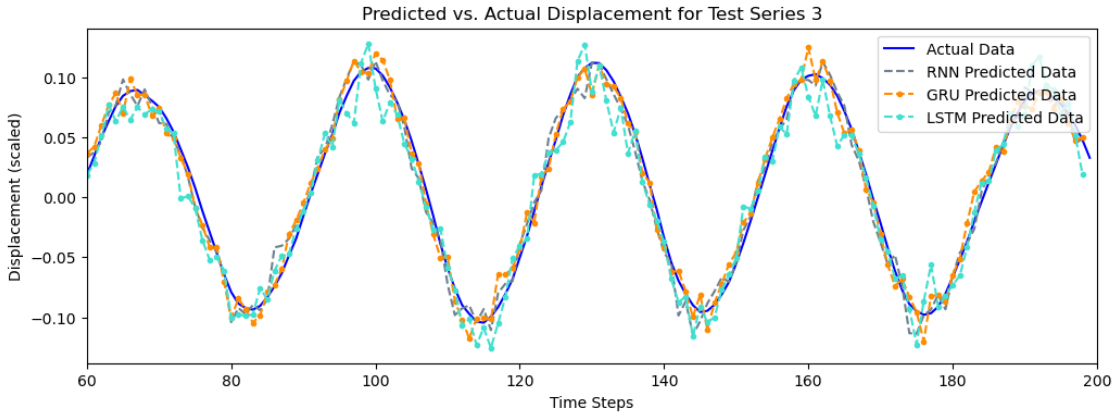


[1261]:
```
# Visualise prediction test for RNN architectures
for i in i_data:
    plt.figure(figsize=(12, 4))
    t = np.arange(len(output_data_test[i]))

    # Plot actual data
    plt.plot(t, output_data_test[i].numpy(), label="Actual Data", color="blue",␣
↪linestyle="-")

    # Plot RNN predictions
    plt.plot(t[n_warmup:n_warmup + len(all_predictions[i])], all_predictions[i][:
↪, 0] / scaling,
             label="RNN Predicted Data", linestyle="--", color='slategrey')

    # Plot GRU predictions
    plt.plot(t[n_warmup:n_warmup + len(gru_all_predictions[i])],␣
↪gru_all_predictions[i][:, 0] / scaling,
             label="GRU Predicted Data", linestyle="--", marker=".",␣
↪color='darkorange')

    # Plot LSTM predictions
    plt.plot(t[n_warmup:n_warmup + len(lstm_all_predictions[i])],␣
↪lstm_all_predictions[i][:, 0] / scaling,
             label="LSTM Predicted Data", linestyle="--", marker=".",␣
↪color='turquoise')
```

```
    plt.xlabel("Time Steps")
    plt.ylabel("Displacement (scaled)")
    plt.legend(loc="upper right")
    plt.xlim(n_warmup, len(actual_data))
    plt.title(f"Predicted vs. Actual Displacement for Test Series {i+1}")
    plt.show()
```







The average MSE scores for predictions show that the GRU (0.000920) slightly outperforms both the RNN (0.001019) and the LSTM (0.001274), though the differences are marginal.

For the convergence plot, Aal models exhibit a steep decline in loss during the early epochs and stablise as training progresses as the expected behaviour. The RNN stabilises at the lowest loss, followed by the GRU, and then the LSTM. This is unexpected behaviour, as LSTM and GRU architectures are intended to perform better while trading an increase in computational overhead. A possible implication can be due to the short sequence of the truncated data, which can negatively counteract GRU and LSTM's intended purpose for mitigating gradients and memory retention over long-term sequences leading to overfitting; while the tradtiional RNN maintains viable in capturing short-term dynamics. Further study with the untruncated data should be performed to validate this.

While the LSTM captures the same overall trends in predictions, the higher variability and phase misalignment in the predictions suggest the complex gating mechanisms are less appropriate for generalising short sequences.

1.7 5. Neural Ordinary Differential Equations (NODEs)

Not completed

1.8 References

[1] Szalai, RS. (n.d.). University of Bristol SEMTM0007. © 1997-2024 Blackboard Inc. All Rights Reserved. US Patent No. 7,493,396 and 7,558,853. Additional Patents Pending. https://www.ole.bris.ac.uk/ultra/courses/_259188_1/
[2] Barton, D. A. (2016). Control-based continuation: Bifurcation and stability analysis for physical experiments. Mechanical Systems and Signal Processing, 84, 54–64. https://doi.org/10.1016/j.ymssp.2015.12.039
[3] Chrysafides, A., & Blanchard, P. A. (2002). On the Influence of Material Hardening on the Fatigue Life of Springs. Journal of Engineering Materials and Technology, 124(2), 226-231
[4] Yao, X., & Liu, Y. (2004). Modeling Nonlinear Time Series with Artificial Neural Networks: A Comparison of Different Architectures. Neurocomputing, 57(1-4), 213-228
[5] Jones, R. L., & McDonald, M. (2000). Nonlinear Vibrations in Mechanical Systems: Harmonics

and Bifurcations. Journal of Sound and Vibration, 234(3), 539-558
[6] Yang, Y., & Liu, Y. (2018). A Comprehensive Comparison of Normalization Methods in Machine Learning Applications. Proceedings of the 2018 International Conference on Data Science and Machine Learning
[7] Jaeger, H. (2001). The "Echo State" Approach to Analysing and Training Recurrent Neural Networks. GMD Report 148, German National Research Center for Information Technology
[8] Jaeger, H. (2002). Adaptive Nonlinear System Identification with Echo State Networks. Proceedings of the International Workshop on Self-Organization and Autonomous Machines, 71-74
[9] Lukoševičius, M., & Jaeger, H. (2009). Reservoir Computing Approaches to Recurrent Neural Network Training. Computer Science Review, 3(3), 127-149
[10] Sklansky, J. (1989). Cluster Validation Using Cross-Validation. Pattern Recognition Letters, 9(1), 35-39
[11] Buonomo, I., & Manfredi, L. (2011). Echo State Networks: A Comparative Study on Reservoir Hyperparameters and Nonlinearities. Neurocomputing, 74(11), 1845-1853
[12] Bergstra, J., & Bengio, Y. (2012). Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research, 13(1), 281-305
[13] Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian Optimization of Machine Learning Algorithms. Advances in Neural Information Processing Systems (NeurIPS), 25, 2951-2959
[15] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9(8), 1735-1780
[16] Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations (ICLR)
[17] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning Representations by Back-propagating Errors. Nature, 323(6088), 533-536
[18] Williams, R. J., & Zipser, D. (1989). A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. Neural Computation, 1(2), 270-280
[19] Cho, K., Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. Empirical Methods in Natural Language Processing (EMNLP), 1724-1734