DSA5206 Project Part 1

Muhammad Haidi Bin Azaman
A0216941E

(a) By setting x(t) = 0, we get

$$\frac{d}{dt}\begin{pmatrix} h_1(t) \\ h_2(t) \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -10 \end{pmatrix}\begin{pmatrix} h_1(t) \\ h_2(t) \end{pmatrix}$$

By matrix multiplication, we get 2 Ordinary Differential Equations (ODEs),

$$\frac{dh_1(t)}{dt} = -h_1(t)$$
$$\frac{dh_2(t)}{dt} = -10h_2(t)$$

By solving the ODEs,

$$h_1(t) = e^{-t+c_1}$$
$$h_2(t) = e^{-10t+c_2}$$

Expressed in the discrete form,

$$\begin{pmatrix} h_1(t+1) \\ h_2(t+1) \end{pmatrix} = \begin{pmatrix} e^{-1} & 0 \\ 0 & e^{-10} \end{pmatrix}\begin{pmatrix} h_1(t) \\ h_2(t) \end{pmatrix}$$

Now, we perform eigendecomposition on the matrix $\begin{pmatrix} e^{-1} & 0 \\ 0 & e^{-10} \end{pmatrix}$, using Python,

```
[28]: import math
      import numpy as np

      a = np.array([
          [math.exp(-1),0],
          [0,math.exp(-10)]
      ])

      np.linalg.eig(a)

[28]: (array([3.67879441e-01, 4.53999298e-05]),
       array([[1., 0.],
              [0., 1.]]))
```

As the eigenvalue 3.67849441e-01, which is e^{-1}, is the largest eigenvalue, the leading dynamic mode is the eigenvector corresponding to this eigenvalue.

As such the leading dynamic mode is $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$

(b)

```
import numpy as np
from tqdm import tqdm

# Define matrices A and B from eq 1.1
A = np.array([
    [-1, 0],
    [0, -10],
])

B = np.array([
    [10**(-3)],
    [10**(3)]
])


num_time_steps = 10**3
num_trajectories = 10**3
delta_t = 10**-3

# initialise trajectories
trajectories = np.zeros(shape=[num_trajectories,num_time_steps,2,1])
print(f'shape of trajectories: {trajectories.shape}')
```

```
# Iterate over trajectories
for i in tqdm(range(num_trajectories)):

    # Initialise h to 0
    h = np.zeros([num_time_steps,2,1])

    # Sample input x using normal distribution
    x = np.random.normal(0, 1, num_time_steps)

    for j in range(1,num_time_steps):
        # update formula
        h[j] = h[j-1] + delta_t * (A @ h[j-1] + B * x[j-1])

    trajectories[i] = h
```

```
# flatten the array
trajectories = trajectories.reshape(num_time_steps*num_trajectories,2)

# transpose the trajectories to pass into covariance matriz
trajectories = trajectories.T

# calculate covariance matrix
covariance_matrix = np.cov(trajectories)

# perform eigendecomposition to do POD
np.linalg.eig(covariance_matrix)
```

```
shape of trajectories: (1000, 1000, 2, 1)
100%|██████████| 1000/1000 [00:05<00:00, 173.34it/s]
(array([1.39813494e-10, 4.79946361e+01]),
 array([[-1.00000000e+00, -1.74597648e-06],
        [ 1.74597648e-06, -1.00000000e+00]]))
```

from the Python code output, we can see that the largest eigenvalue is 4.79946361e+01. As such the leading spatial mode is the eigenvector corresponding to this eigenvalue.

The leading spatial mode is $\begin{pmatrix} -1.74597648e-06 \\ -1 \end{pmatrix}$

(c) It is appropriate to use the leading (spatial) POD mode as a means to reduce the system eq (1.1) from 2 to 1 dimension. As shown in part (b) the 2nd spatial POD mode is $\begin{pmatrix} -1 \\ -1.74597648e-06 \end{pmatrix}$. As -1.74597648e-06 is very small, this spatial mode is very similar to the dynamic mode in part (a) which is $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. As such using the dynamic mode from part (a) will be suboptimal as it is just like using the 2nd spatial POD mode. Therefore, one should use the leading spatial POD mode from part (b).

(d)

$$\dot{h} = Ah(t) + Bx(t) \text{ (eq 1.3)}$$

$$\frac{dh(t)}{dt} = Ah(t) + Bx(t)$$

$$\frac{dh(t)}{dt} - Ah(t) = Bx(t)$$

Multiply both sides with e^{-At},

$$e^{-At}\frac{dh(t)}{dt} - e^{-At}Ah(t) = e^{-At}Bx(t)$$

$$\frac{d}{dt}e^{-At}h(t) = e^{-At}Bx(t)$$

Integrate both sides from 0 to t,

$$e^{-At}h(t) - e^{-A.0}h(0) = \int_0^t e^{-A\tau}Bx(\tau)d\tau$$

Let $h_* = e^{-At}h(t) - e^{-A.0}h(0)$, and using the Cayley Hamilton theorem, $e^{At} = \sum_{k=0}^{+\infty} \frac{t^k}{k!}A^k$

$$h_* = \int_0^t \sum_{k=0}^{\infty} \frac{(-\tau)^k}{k!}A^k Bx(\tau)d\tau$$

$$= \sum_{k=0}^{\infty} A^k B \int_0^t \frac{(-\tau)^k}{k!}x(\tau)d\tau$$

$$h_* = (B\ AB\ ...\ A^{m-1}B\ ...) \begin{pmatrix} \int_0^t \frac{(-\tau)^0}{0!}x(\tau)d\tau \\ \int_0^t \frac{(-\tau)^1}{1!}x(\tau)d\tau \\ ... \\ \int_0^t \frac{(-\tau)^{m-1}}{(m-1)!}x(\tau)d\tau \\ ... \end{pmatrix}$$

$\because rank = m$ and we can ignore terms $k \geq m$ based on Cayley-Hamilton theorem,

$$h_* = (B\ AB\ ...\ A^{m-1}B) \begin{pmatrix} \int_0^t \frac{(-\tau)^0}{0!}x(\tau)d\tau \\ \int_0^t \frac{(-\tau)^1}{1!}x(\tau)d\tau \\ ... \\ \int_0^t \frac{(-\tau)^{m-1}}{(m-1)!}x(\tau)d\tau \end{pmatrix}$$

As such, it can be seen that for the continuous-time dynamics in eq. (1.3), the controllability condition is also given by eq. (1.5)

$$R = (B\ AB\ ...\ A^{m-1}B) \text{ has full column rank m}$$

(e) For a symmetric matrix X, $X = X^T$

$$W_c(t) = \int_0^t e^{As}BB^Te^{A^Ts}\,ds \quad (1.6)$$

$$\text{let X} = e^{As}BB^Te^{A^Ts}$$

$$X^T = (e^{As}BB^Te^{A^Ts})^T = (e^{A^Ts})^T(B^T)^T(B)^T(e^{As})^T$$

$$= e^{As}BB^Te^{A^Ts} = X$$

As such, this can be extended to the controllability Gramian, showing that $W_c(t) = W_c^T(t)$. Therefore, the controllability Gramian defined in (1.6) is symmetric.

$$\text{The intergral } W_c(t) = \int_0^t e^{As}BB^Te^{A^Ts}\,ds \quad \text{is an integral over an inner product.}$$

As such, this implies that controllability Gramian is positive semidefinite. To show that it is positive definite, we need to prove that $W_c(t)$ is not $= 0$.

If $W_c(t) = 0$, then there must exist a non zero vector x such that $x^TW_c(t)x \leq 0$

$$\text{This implies that } x^Te^{As}B = \sum_{k=0}^{\infty}\frac{s^k}{k!}x^TA^kB = 0$$

$$x^TA^kB = 0 \quad for \quad 0 \leq k \leq m-1 \text{ by Cayley-Hamilton theorem.}$$

However, since the controllability matrix has full rank m, there does not exists a non-zero vector x^T that fulfills the equation $x^T(B \quad AB \quad ... \quad A^{m-1}B) = 0$
$\therefore$ by contradiction, the controllability Gramian must be positive definite for every $t > 0$.

(f) The general solution of eqn (1.3) is given as $h(t) = e^{At}h(0) + \int_0^t e^{A(t-s)}Bx(s)ds$
by subbing eqn (1.7) in to this general solution, when $t = t_*$,

$$h(t_*) = e^{At}h(0) + \int_0^{t_*} e^{A(t_*-s)}BB^Te^{A^T(t_*-s)}W_c(t_*)^{-1}h_*ds$$

$$= e^{At}h(0) + W_c(t_*)W_c(t_*)^{-1}h_* \ (\because W_c(t_*) = \int_0^{t_*} e^{A(t_*-s)}BB^Te^{A^T(t_*-s)}ds)$$

$$= h_* \ (\because t_* > 0, e^{At}h(0) = 0)$$

$$\therefore h(t_*) = h_* \text{ (shown)}$$

(g)

$$E = \int_0^{t_*} |x_*(s)|^2\, ds = \int_0^{t_*} (B^T e^{A^T(t_*-s)} W_c(t_*)^{-1} h_*)^T (B^T e^{A^T(t_*-s)} W_c(t_*)^{-1} h_*)\, ds$$

$$= \int_0^{t_*} h_*^T (W_c(t_*)^{-1})^T e^{A(t_*-s)} B B^T e^{A(t_*-s)} W_c(t_*)^{-1} h_*)\, ds$$

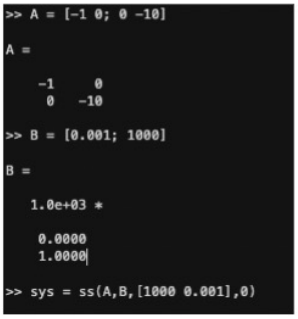
$$= h_*^T (W_c(t_*)^{-1})^T W_c(t_*) W_c(t_*)^{-1} h_* \ (\because W_c(t_*) = \int_0^{t_*} e^{A(t_*-s)} B B^T e^{A^T(t_*-s)} ds)$$

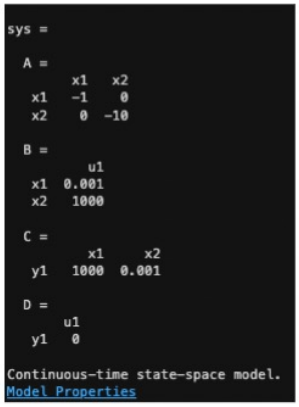
$$= h_*^T (W_c(t_*)^{-1})^T h_*$$

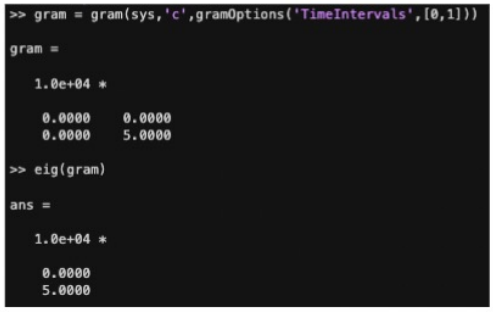
$$= h_*^T W_c(t_*)^{-1} h_* \ (\because \text{the gramian matrix is symmetric (part e)},\ \therefore (W_c(t_*)^{-1})^T = W_c(t_*)^{-1})$$

$$\therefore E = h_*^T W_c(t_*)^{-1} h_* \text{ (shown)}$$

(h)



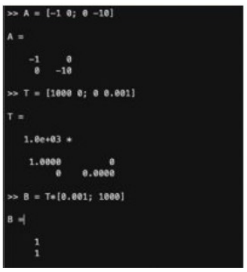


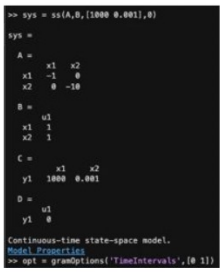


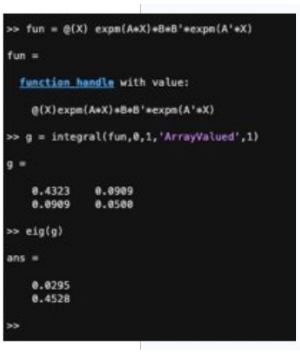
From the Matlab output, above, the controllability Gramian is computed as $\begin{pmatrix} 0 & 0 \\ 0 & 50000 \end{pmatrix}$

The eigenvalues are 0 and 50000.

(i)







From the Matlab output, above, the controllability Gramian is computed as $\begin{pmatrix} 0.4323 & 0.0909 \\ 0.0909 & 0.0500 \end{pmatrix}$

The eigenvalues are 0.0295 and 0.4528.

It can be observed that these eigenvalues are much closer to each other than in part (h) where the eigenvalues were very different from each other (0 and 5000).

From part (g), it was stated that if W_c has small eigenvalues, then the work required to steer the system along the direction of the corresponding eigenvectors will be very large. Hence, the eigenvalues of give a measure of how easy it is to control the system.

As such, this means that in part (h), it will be very easy to control the system along the eigenvector corresponding to the eigenvalue 5000, but you will have no control over the other eigenvector which correponds to the eigenvalue of 0.

For part (i), the eigenvalues are closer to each other in magnitude, so you can control the system in both directions, though more difficult than the case with large eigenvalue 5000.

DSA5206 Project Part 2

Is Non-Linear Always Better?

Exploring the effects of linear vs non-linear dimensionality reduction

on downstream time series forecasting tasks

Muhammad Haidi Bin Azaman
A0216941E

May 1, 2024

1 Introduction

With the advancement of the field of machine learning, we see models of high complexities achieving state-of-the-art results on various domains. This trend implies that increased complexity often leads to superior results, particularly through the integration of highly non-linear approaches. However, a fundamental question arises: Are non-linear methods always the optimal choice? Can linear techniques outperform their non-linear counterparts? This paper delves into this topic by investigating the effectiveness of both linear and non-linear dimensionality reduction methods in subsequent time series forecasting tasks.

Time series prediction is a critical task in various domains such as finance, healthcare, and environmental science [1]. With the increasing availability of high-dimensional time series data, the need for effective dimensionality reduction techniques has become paramount. Dimensionality reduction methods aim to extract essential features from high-dimensional data into a lower dimensional space so as to reduce computational complexity.

The dataset used for this project is the Jena Climate dataset, obtained from the Weather Station of the Max Planck Institute for Biogeochemistry. It covers the period from January 1st, 2009, to December 31st, 2016. The dataset encompasses 14 variables such as air temperature, atmospheric pressure, humidity, and wind direction, among others, recorded at 10-minute intervals. The target variable for forecasting is Temperature in degree Celcius or denoted as T (deg C) in the csv file. The full time series for Temperature is shown in Figure 1

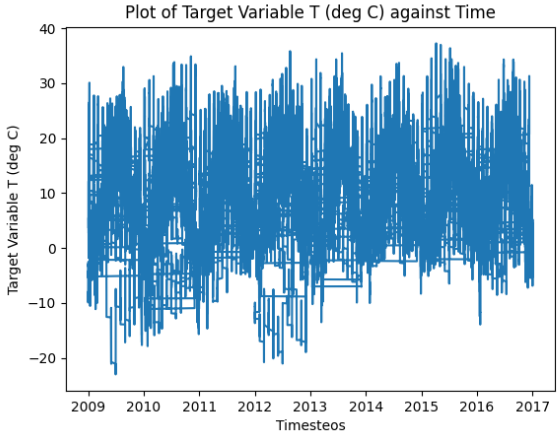


Figure 1: Plot of Temperature (deg C) against time.

2 Methods

In this study, four sets of experiments were conducted to explore different approaches to forecasting temperature using the Jena Climate dataset.

1. Baseline. Utilises the full set of 14 features for forecasting.

2. Linear Principal Component Analysis (PCA). PCA was applied to the dataset to reduce its dimensionality to two principal components, which were then used for forecasting.

3. Non-linear Autoencoder. Two autoencoder models were developed from scratch and trained, a Fully Connected Network (FCN) and a Long Short-Term Memory (LSTM) autoencoder. They were used to extract

nonlinear representations from the data's latent space, which were subsequently used for forecasting.

4. Linear PCA with three components. To investigate if an additional principal component would lead to improvement in forecasting results.

For each set of experiments, three different deep learning architectures, namely Recurrent Neural Network (RNN), LSTM, and Gated Recurrent Unit (GRU), were trained and evaluated to assess their predictive capabilities after dimensionality reduction.

2.1 Data Preprocessing

The dataset was divided into chunks, with each chunk representing a contiguous sequence of data points over a specified lag period / lookback. The lookback determines the number of previous time steps used as input features for predicting the target variable. For instance, if the lag period is set to 10, each chunk consists of 10 consecutive data points, with the target value representing the subsequent data point after the last lag value.

The dataset was split into train, validation and test sets using a 70-20-10 split. It is important to note that the dataset is partitioned in order without random shuffling to maintain the temporal integrity of each set. Normalisation was applied to scale the input features between -1 and 1. This avoids the issue of features with larger magnitudes from dominating the learning process.

2.2 Baseline Experiments

In the first approach, the entire set of 14 features, including atmospheric pressure, humidity, wind direction, and others, were used directly for forecasting temperature. These baseline experiments are used as a benchmark comparison for the other experiments. For these experiments, the features X are of shape (512, 10, 14) as the batch size is 512, lookback window is 10, and there are 14 features in the dataset. The target variable (temperature), y is of shape (512,1). This is also consistent for the other sets of experiments utilising dimensionality reduction, except the number of features is equal to number of reduced features.

2.3 Linear Principal Component Analysis

PCA is a linear dimensionality reduction technique commonly used in machine learning and data analysis [2]. It aims to transform high-dimensional data into a lower-dimensional space while preserving as much of the original data variance as possible. PCA was performed on the normalised dataset to extract the principal components accounting for the maximum variance. The first two principal components, representing the most significant sources of variation in the data, were retained for training the forecasting model.

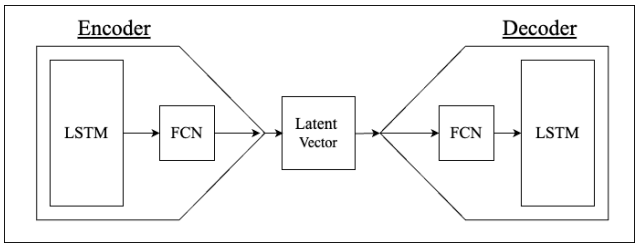


Figure 2: Network architecture of the LSTM autoencoder.

2.4 Non-linear Autoencoder

In the third approach, two autoencoder models were trained to learn a low-dimensional representation of the input features. The autoencoder architecture comprises an encoder component, which maps the high-dimensional input features to a lower-dimensional latent space, and a decoder component, which reconstructs the input features from the latent representation.

Two different autoencoder architectures were trained. They are the FCN and LSTM autoencoder architectures [3]. Figure 2 shows the network architecture for the LSTM autoencoder architecture. The encoder consists of a LSTM module with 4 hidden states. This module reduces the input dataset of 14 features to 4. Following which, the output is passed to a shallow FCN that further reduces the output to a hidden representation of shape (512, 10, 2). In other words, the encoder encodes the input data to two features, similar to the PCA method implemented. The decoder is the inverse of the encoder. It attempts to reconstruct the original time series input. The latent representation is learned by minimising the reconstruction loss of the autoencoder decoder output.

In contrast, the FCN autoencoder employs a different architecture. Instead of incorporating an LSTM module, it integrates another fully connected layer with a higher number of hidden states. This configuration facilitates the sequential reduction of dimensionality from input to latent space as the time series traverses through the network.

2.5 Forecasting model 1: RNN

The Recurrent Neural Network (RNN) is adept at analyzing sequential data, making it great for time series forecasting. With recurrent units maintaining hidden states [4], the RNN captures temporal dependencies by combining current observations with past states. However, standard RNNs can struggle with long-term dependencies, also known as the vanishing or exploding gradient problem [5]. This limitation prompted the development of LSTM and GRU net-

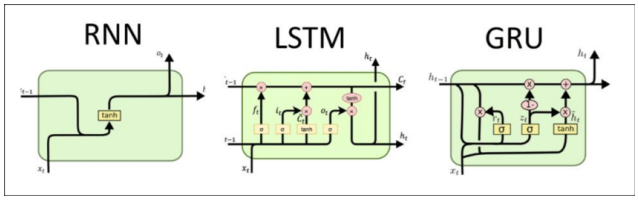


Figure 3: Architectures of a RNN, LSTM and GRU module.

works, which can mitigate this issue to a certain extent. Figure 3 shows the architecture of a RNN cell. The figure shows that the RNN only employs one hidden state which is used for computation at each timestep, serving as short-term memory.

2.6 Forecasting model 2: LSTM

The Long Short-Term Memory (LSTM) network [6], a variant of the RNN, addresses the shortcomings of standard RNNs when it comes to capturing long-term dependencies. Like the RNN, LSTM is designed to analyze sequential data and is particularly well-suited for time series forecasting tasks. While traditional RNNs struggle with the vanishing or exploding gradient problem, LSTM networks introduce specialized memory cells that allow for more effective handling of long-term dependencies as shown in Figure 3. These memory cells contain gating mechanisms, including input, forget, and output gates, which regulate the flow of information within the network [7]. This enables LSTM networks to retain and selectively update information over multiple time steps, facilitating the capture of both short-term and long-term patterns in the data. In contrast to the single hidden state used in standard RNNs, LSTM networks maintain multiple internal states, including cell states and hidden states, enhancing their capacity to store and process information across different time steps.

2.7 Forecasting model 3: GRU

The Gated Recurrent Unit (GRU) is another variant of the RNN architecture, designed to address the challenges of capturing long-term dependencies in sequential data. Similar to LSTM, GRU networks incorporate gating mechanisms to regulate the flow of information within the network and facilitate learning across multiple time steps [8]. GRU units combine the strengths of LSTM networks while simplifying their architecture, making them computationally more efficient [7]. Unlike LSTM, which maintains separate cell states and hidden states, GRU units utilize a single hidden state, streamlining the model architecture as seen in Figure 3. This simplicity allows GRU networks to learn representations of sequential data more efficiently while still capturing both short-term and long-term dependencies relatively effectively. As such, the GRU can be seen as an intermediate between RNNs and LSTMs.

2.8 Metrics

The metrics used are Mean Absolute Error (MAE) and Mean Squared Error (MSE). MAE represents the average magnitude of absolute errors between predicted values and actual values, thereby offering a robust measure that is relatively less influenced by outliers. Conversely, MSE calculates the average of squared deviations between predicted and actual values, consequently imposing greater emphasis on larger errors, due to the square term.

$$\text{MAE} = \sum_{i=1}^{D} |x_i - y_i|, \quad \text{MSE} = \sum_{i=1}^{D} (x_i - y_i)^2$$

3 Results

For all experiments, the leanest model architectures are used for training. For example, for the first set of experiments on baseline models, the hidden_size and num_layers for the RNN, LSTM, GRU models are set to 1. For the models in the other sets of experiments, the hidden_size is set to 2 or 3 depending on the dimensionality of the reduced data (for PCA) or latent space (for Autoencoder output).

A learning rate scheduler, ReduceLROnPlateau, was used during training. The ReduceLROnPlateau scheduler is used to dynamically adjust the learning rate during training [9]. Its primary purpose is to reduce the learning rate when the validation loss has stopped improving, typically indicating that the model may be stuck in a local minimum or a plateau. Due to limitations on compute resources, all forecasting models were trained for 10 epochs only. Surprisingly, promising results could be produced despite training for only a small number of epochs.

3.1 Experiments 1: Baseline Experiments

After data preprocessing, the full set of 14 features were passed to the three forecasting models. Table 1 shows the results of the baseline experiments. Surprisingly, the MSE and MAE values are very small. This indicates that the three models were able to learn very well from the full dataset of 14 features. Thus, this will serve as a strong baseline for the next set of experiments.

Table 1: Experiments 1, using full set of features.

Models	Mean Sqr. Err.	Mean Abs. Err.
RNN	0.000683	0.017895
LSTM	0.000245	0.012197
GRU	0.001361	0.027916

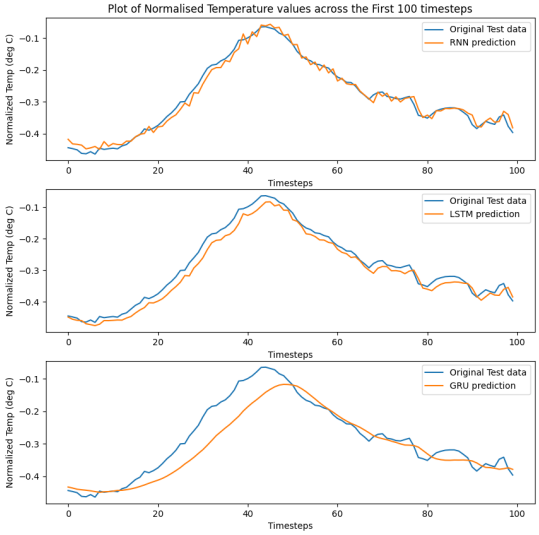


Figure 4: The test set prediction of the RNN, LSTM, GRU models for the baseline experiments using the full set of features.

Figure 4 shows the comparison of the models' test prediction with the actual groundtruth for the first 100 timesteps of the test dataset, across the RNN, LSTM and GRU models. It can be seen that the LSTM and RNN models are able to model the groundtruth well. However, the RNN prediction has more fluctuations which could have resulted in the higher MSE and MAE values in Table 1 as compared to the LSTM model. On the other hand, the GRU model seems to predict a time series that is quite far away from the groundtruth as compared to the RNN and LSTM models, consistent with the results in Table 1.

3.2 Linear Principal Component Analysis

The explained_variance_ratio_ attribute of the PCA object revealed that the first principal component explains approximately 57.89% of the variance in the data, while the second principal component explains about 21.59%. Together, these two principal components capture approximately 79.48% of the total variance in the original data. This reduction in dimensionality allows training on two features instead of 14.

Figure 5 depicts the scatter plot of the original dataset projected onto the first two principal components. Notably, the first component of the PCA reveals a substantial amount of variance, evidenced by a clear colour gradient from left to right along the horizontal axis of the first principal component. This gradient corresponds to the range of target values within the dataset, with brighter colors transitioning to

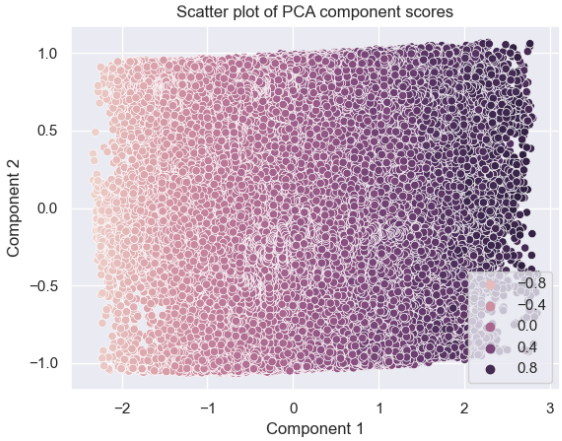


Figure 5: The scatter plot of the first two principal components of the dataset after PCA.

darker shades. Such a gradient shows that the two principal components effectively capture the variance inherent in the dataset.

3.3 Experiments 2: Forecasting after PCA

By projecting the original dataset onto the first two principal components, we get a lower dimensional representation of the data that was then used to train the forecasting models. The results of the second set of experiments are shown in Table 2. In these experiments, the LSTM and GRU have comparable performance. Their MSE and MAE values are similar whereas the RNN seems to have struggled with the reduction of data.

Table 2: Experiments 2, PCA(n=2).

Models	Mean Sqr. Err.	Mean Abs. Err.
RNN	0.002553	0.03913
LSTM	0.002469	0.036635
GRU	0.002454	0.036878

Comparing with the results of experiments 1 in Table 1, it is observed that the model performance across all three models have deteriorated. This can be explained by the percentage of total variance that was captured by the first two principal components. As these two principal components could only capture approximately 79.48% of the total variance in the original data, it is clear to see that the drop in model performance could be attributed to the fact that the reduced dataset was not a sufficiently good representation of the original dataset.

Perhaps more principal components would be necessary to capture significantly more of the total variance in the dataset. In the fourth set of experiments, the effect of an

additional principal component on eventual forecast values will be investigated. The hypothesis is that with this added principal component, the new reduced transformed data will be an improved representation of the input time series, thus leading to better forecast results.

3.4 Non-linear Autoencoders

Both the FCN and LSTM autoencoders were trained for 100 epochs to reduce the input time series to latent spaces of the same dimensionality for fair comparison. As seen in Table 3, the performance of the LSTM autoencoder far outweighs that of the FCN autoencoder. Both the MSE and MAE loss values for the LSTM are an order of magnitude or more lower than that of the FCN autoencoder. The difference in results can be attributed to the strength and importance of the LSTM module in extracting meaningful temporal patterns from the existing input time series.

With the FCN autoencoder, the input time series is immediately flattened into a 1D vector. This could potentially cause issues as the temporal significance of the time series is lost. As such, by using the LSTM, the temporal patterns are maintained and useful features are learnt [10]. This can be likened to the case of Convolutional Neural Networks (CNNs) in the application of image classification. CNNs outperform basic FCNs as the convolution operation is very useful in capturing spatial relationships [11]. This is similar to the case of LSTMs for time series modelling in capturing temporal dependencies.

Table 3: Autoencoder Reconstruction Loss.

Autoencoder	Mean Sqr. Err.	Mean Abs. Err.
FCN	0.1572244	0.2567081
LSTM	0.0059954	0.0386229

3.5 Experiments 3: Forecasting using Autoencoder latent states

The LSTM autoencoder was used to generate a lower dimensional latent representation of the input time series. The decoder was discarded and the encoder weights were frozen such that during the forecasting task, the LSTM autoencoder was only used for inference for the latent state output. This latent state was then used as the input time series for the forecasting task.

The results in Table 4 show that the RNN and LSTM models have similar performance whereas the GRU model performed the worst. Comparing to the results of the first set of experiments (Table 1) and second set of experiments (Table 2, these models using LSTM autoencoder hidden states performed the worst. This suggests the presence of strong linear relationships between the dataset's features and the target variable. Despite a prolonged training period of 100 epochs, the latent representation fails to outperform the PCA-transformed data, signifying the latent states' limited efficacy as a reduced dataset representation.

Table 4: Experiments 3, LSTM Autoencoder.

Models	Mean Sqr. Err.	Mean Abs. Err.
RNN	0.009703	0.07853
LSTM	0.009764	0.077963
GRU	0.010535	0.081512

4 Further Results

4.1 Experiments 4: Forecasting using PCA(n=3)

Comparing the results of the second set of experiments (Table 2) and the third (Table 4), it is shown that the linear dimensionality reduction method of PCA produced better results on the forecasting task as compared to the non-linear method of LSTM autoencoder.

Recall that for the second set of experiments, only two principal components were used. These components only contribute to a total of 79.48% of the total variance of the dataset. In this fourth set of experiments, this is further investigated by performing PCA with three principal components instead.

Together with the third principal component, the percentage of total variance captured by all three principal components increases to 95.50%. This is a major improvement.

Table 5: Comparison of Forecasting MSE values between models of PCA(n=2) and PCA(n=3).

Models	PCA(n=2)	PCA(n=3)
RNN	0.002553	0.00109
LSTM	0.002469	0.002389
GRU	0.002454	0.001552

The values in Table 5 show the difference in MSE loss values between the model using the reduced data input from PCA(n=2)and PCA(n=3). The results validate the previous hypothesis of the third principal component potentially improving forecasting results, as all three forecasts models show improvements when trained using the reduced representation from PCA(n=3). Therefore, this shows the importance of hyperparameter tuning when doing PCA on a dataset. It is crucial to check the total explained_variance_ratio across the N components selected. It has been shown in this fourth set of experiments how a difference of just one additional principal component greatly affects the outcome of forecast values.

5 Conclusion

In this paper, linear and non-linear dimensionality reduction techniques were investigated. The reduced representation output coming from each of these methods were then used to train three different deep learning architectures: RNN, LSTM, GRU. The results have shown that the performance of the forecasting models using the PCA outputs were better than the LSTM autoencoder method. This prompts us to reconsider the original question at the beginning of the paper: Are non-linear methods always better? Contrary to expectations, the findings demonstrate that linear methods can be very effective and efficient when applied correctly.

The primary limitations of this study center around hyperparameter tuning. Since the deep learning models were trained for only 10 epochs in the main experiments, there is a possibility of uncovering more insightful findings by extending the training duration. This suggests potential future research avenues, including experimenting with varying numbers of epochs, layers and hidden states for each model. Moreover, to further validate the results, it would be beneficial to explore the training of alternative time series datasets with higher dimensionality.

While none of the experiments in this paper managed to outperform the baseline models leveraging the entire set of dataset features, it is intriguing to observe the promising results obtained through dimensionality reduction methods. Extending this insight to datasets with very high dimensionality, especially those containing more than 100 features, it becomes clear why these dimensionality reduction methods are so important. The experiments in this paper underscores the potential effectiveness of a simple linear PCA method in yielding robust outcomes despite the utilization of a restricted number of input features.

References

[1] B. Lim and S. Zohren, "Time-series forecasting with deep learning: A survey," *Philosophical Transactions of the Royal Society A*, vol. 379, no. 2194, p. 20 200 209, 2021.

[2] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[3] H. Homayouni, S. Ghosh, I. Ray, S. Gondalia, J. Duggan, and M. G. Kahn, "An autocorrelation-based lstm-autoencoder for anomaly detection on time-series data," in *2020 IEEE international conference on big data (big data)*, IEEE, 2020, pp. 5068–5077.

[4] J. T. Connor, R. D. Martin, and L. E. Atlas, "Recurrent neural networks and robust time series prediction," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 240–254, 1994.

[5] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[6] Y. Hua, Z. Zhao, R. Li, X. Chen, Z. Liu, and H. Zhang, "Deep learning with long short-term memory for time series prediction," *IEEE Communications Magazine*, vol. 57, no. 6, pp. 114–119, 2019.

[7] Y. Hu, A. Huber, J. Anumula, and S.-C. Liu, "Overcoming the vanishing gradient problem in plain recurrent networks," *arXiv preprint arXiv:1801.06105*, 2018.

[8] Z. Niu, Z. Yu, W. Tang, Q. Wu, and M. Reformat, "Wind power forecasting using attention-based gated recurrent unit network," *Energy*, vol. 196, p. 117 081, 2020.

[9] A. Lewkowycz, "How to decay your learning rate," *arXiv preprint arXiv:2103.12682*, 2021.

[10] G. Lai, W.-C. Chang, Y. Yang, and H. Liu, "Modeling long-and short-term temporal patterns with deep neural networks," in *The 41st international ACM SIGIR conference on research & development in information retrieval*, 2018, pp. 95–104.

[11] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial intelligence review*, vol. 53, pp. 5455–5516, 2020.

A Appendix

The full code for this paper can be found at my github repo here: `https://github.com/haidiazaman/dsa5206-project/tree/main`.

If the link is not accessible, please email me at `e0540498@u.nus.edu`.

```python
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_stacked_layers):
        # input size is number of expected features in the input x
        super().__init__()
        self.hidden_size = hidden_size
        self.num_stacked_layers = num_stacked_layers

        self.rnn = nn.RNN(input_size, hidden_size, num_stacked_layers,batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        batch_size = x.size(0)
        h0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device) # rnn

        out, _ = self.rnn(x, h0) # output is (output, h_n) but dont need to manually keep track o
        out = self.fc(out[:, -1, :]) # -1 to take the final value of the time series, this is the
        return out
```

Figure 6: Model code for RNN.

```python
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_stacked_layers):
        # input size is number of expected features in the input x
        super().__init__()
        self.hidden_size = hidden_size
        self.num_stacked_layers = num_stacked_layers

        self.lstm = nn.LSTM(input_size, hidden_size, num_stacked_layers,batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        batch_size = x.size(0) # input must be torch tensor
        h0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device) # short term memory
        c0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device) # long term memory

        out, _ = self.lstm(x, (h0, c0)) # output is output, (h_n, c_n), but we dont need to track the next state
        # out shape is (batch_size, num_timesteps inclusive of predicted day, num_features_in_hidden_state)
        out = self.fc(out[:, -1, :]) # -1 to take the final value of the time series, this is the prediction fo
        return out
```

Figure 7: Model code for LSTM.

```python
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, num_stacked_layers):
        # input size is number of expected features in the input x
        super().__init__()
        self.hidden_size = hidden_size
        self.num_stacked_layers = num_stacked_layers

        self.gru = nn.GRU(input_size, hidden_size, num_stacked_layers,batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        batch_size = x.size(0)
        h0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device)

        out, _ = self.gru(x, h0) # output is (output, h_n) but dont need to manually keep t
        out = self.fc(out[:, -1, :]) # -1 to take the final value of the time series, this
        return out
```

Figure 8: Model code for GRU.

```python
class Autoencoder(nn.Module):
    def __init__(self, input_features, lookback, hidden_size):
        super(Autoencoder, self).__init__()
        self.input_features = input_features
        self.lookback = lookback
        self.encoder = nn.Sequential(
            nn.Linear(input_features*lookback, 64),
            nn.ReLU(),
            nn.Linear(64, hidden_size),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(hidden_size, 64),
            nn.ReLU(),
            nn.Linear(64, input_features*lookback),
            nn.ReLU()
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten input tensor
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        decoded = decoded.view(x.size(0), self.lookback, self.input_features)
        return decoded
```

Figure 9: Model code for the FCN autoencoder.

```python
class LSTM_Autoencoder(nn.Module):
    def __init__(self, input_size, hidden_size, num_stacked_layers):
        # input size is number of expected features in the input x
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_stacked_layers = num_stacked_layers

        self.lstm_encoder = nn.LSTM(input_size, hidden_size, num_stacked_layers,batch_first=True)
        self.fc_encoder = nn.Linear(hidden_size, 2)

        self.fc_decoder = nn.Linear(2, hidden_size)
        self.lstm_decoder = nn.LSTM(hidden_size, input_size, num_stacked_layers,batch_first=True)

    def forward(self, x):
        batch_size = x.size(0) # input must be torch tensor

        # ENCODE
        h0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device) # short term memory
        c0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device) # long term memory
        out, _ = self.lstm_encoder(x, (h0, c0)) # output is output, (h_n, c_n), but we dont need to track the
        # out shape is (batch_size, num_timesteps inclusive of predicted day, num_features_in_hidden_state)
        out = self.fc_encoder(out)

        # DECODE
        out = self.fc_decoder(out)
        h0 = torch.zeros(self.num_stacked_layers, batch_size, self.input_size).to(device) # short term memory
        c0 = torch.zeros(self.num_stacked_layers, batch_size, self.input_size).to(device) # long term memory
        out, _ = self.lstm_decoder(out, (h0, c0)) # output is output, (h_n, c_n), but we dont need to track the

        return out
```

Figure 10: Model code for LSTM autoencoder.