Application of Banach's Fixed Point Theorem in Machine Learning for Recurrent Neural Network Architecture

VINITA SUDHIR, DR MANISH KUMAR, DR REETA SHUKLA
BHARTI VISHWAVIDYALAYA, BALOD ROAD, CHANDKHURI,DURG
CHHATTISGARH, INDIA
PIN 491001

Abstract

To improve the stability and convergence of recurrent neural networks (RNNs) for time series forecasting applications, this research explores the use of fixed-point theory. Our goal is to enhance the generalization performance of RNN models and reduce stability problems by including a contraction mapping check into the training process. We assess the performance of RNNs with and without contraction mapping versus conventional machine learning models using the airline passenger dataset as a case study. Our tests demonstrate how contraction mapping can improve model stability and lead to higher predicted accuracy. The findings support a comprehensive strategy that combines theoretical understanding with empirical study and highlights the significance of mathematical concepts in deep learning research.

Keywords: Fixed point theorem, recurrent neural network, contraction mapping, machine learning.

1. Introduction

In machine learning, recurrent neural networks, or RNNs (see Fig. 1), have become indispensable, especially for tasks involving sequential input, such as speech recognition, time series prediction, and language modeling [1]. Because RNNs may retain a recollection of past inputs, unlike typical neural networks, they are ideally suited for processing sequences in which context is crucial. As training goes on, maintaining the stability and convergence of the hidden states is a major problem for RNNs [2]. The network may become unstable if the hidden states do not converge, resulting in subpar performance and challenging training.

To overcome the shortcomings of more conventional neural networks, including Feedforward Neural Networks (FNNs), recurrent neural networks (RNNs) were developed to process sequential input [3]. Without taking into account the context or order of previous inputs, FNN processes each input independently through a number of hidden layers. As a result, it is unable to grasp the dependencies between inputs and handle sequential data well [4,5,6]. FNNs are therefore not well adapted for sequential processing tasks like time series analysis, speech recognition, machine translation, language modeling, and many other applications that call for sequential processing [7]. RNN enters the scene to overcome the drawbacks of conventional neural networks.

By adding a recurrent link that allows data to move from one time step to the next, RNN overcomes these constraints. The network can gather information from earlier steps and apply it to the current step, allowing the model to learn temporal dependencies and handle input of variable length [8,9,10]. This recurrent connection allows RNNs to maintain internal memory, where the output of each step is fed back as an input to the next step.

By using Banach's Fixed Point Theorem, a key finding in fixed point theory, this work seeks to address this difficulty [11]. A contraction mapping over an entire metric space has a unique fixed point under certain conditions, and Banach's theorem ensures that iterative applications of the mapping will converge to this fixed point. Banach's theorem allows us to guarantee that the hidden states converge to a stable point by considering the state update equations of an RNN as a contraction mapping.

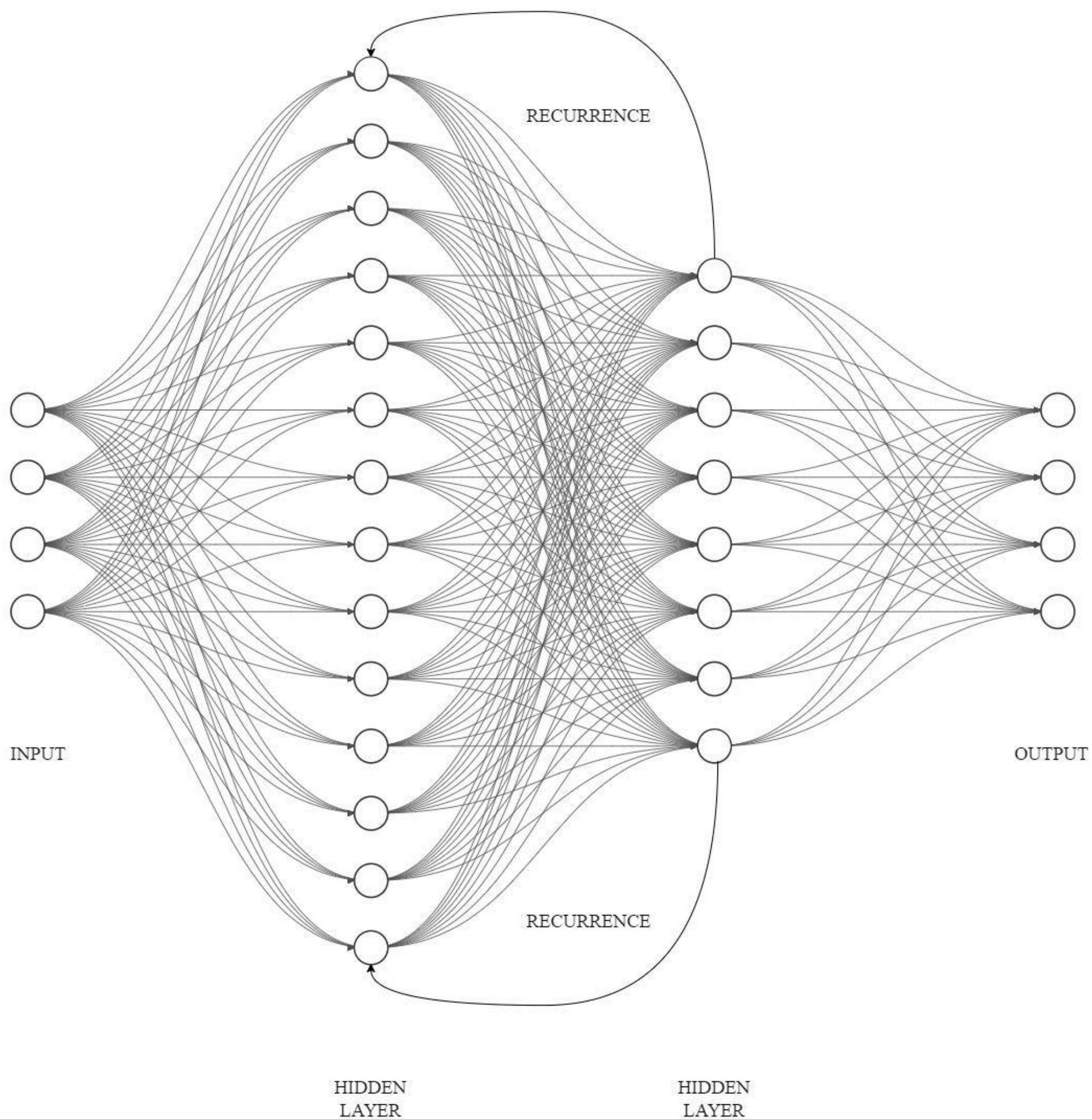


Fig. 1 Recurrent Neural Network architecture diagram

https://doi.org/10.5281/zenodo.14742173

The first step in applying Banach's Fixed Point Theorem to RNNs is to rewrite the state update equation. Typically, an RNN's hidden state h_t at time step t is updated according to the equation:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \quad (1)$$

Where W_h, W_x are weight matrices, b is the bias vector and σ is a non-linear activation function. It is ensured that this update equation produces a contraction mapping, i.e., for any two hidden states, h_1 and h_2, the distance between their images under the mapping is smaller than k times the distance between h_1 and h_2. To do this, we must choose a constant k, such that $0 \leq k < 1$.

Banach's theorem can be utilized to ensure that the concealed states will converge once it has been established that the state update equation is a contraction mapping. This entails creating an algorithm based on Python to confirm the contraction condition and including it in the RNN's training procedure. We can increase the stability of the hidden states and thus improve training and inference performance by making sure that the contraction condition is satisfied. This study uses a sequence prediction challenge to evaluate this method. Specifically, the RNN is trained to predict the subsequent value in a succession of data points, e.g., the subsequent time series value or letter in a text sequence [12,13,14,15]. This illustrates the usefulness of using Banach's Fixed Point Theorem by contrasting the RNN's performance with and without the contraction condition check. The findings demonstrate that training becomes more consistent and prediction accuracy improves when the contraction check is included.

In conclusion, this work offers a novel strategy for utilizing Banach's Fixed Point Theorem to enhance the stability and convergence of RNNs. We present a strong framework for stable RNN training by reformulating the RNN state update equations as a contraction mapping and ensuring the contraction condition is satisfied. This paper offers a fresh approach to tackling one of the main problems in RNN training by bridging the gap between mathematical theory and real-world machine learning applications.

2. Research Gap

There is a significant research gap in the quickly developing field of deep learning, specifically in the area of recurrent neural networks (RNNs) for time series forecasting, concerning the incorporation of mathematical concepts to improve model stability and convergence. RNNs are prone to stability problems during training, which can impair their performance and generalization capacity, despite their amazing ability to capture temporal dependencies and patterns in sequential data. The research gaps are explained below:

1. The absence of systematic inquiry into the use of fixed-point theory in RNN training is one of the main research gaps. The fixed-point theory is very pertinent to the optimization process in deep learning models, as it offers a mathematical framework for examining the convergence of iterative algorithms. Nevertheless, few research has used fixed-point theory to enhance the stability and convergence characteristics of RNNs, in spite of its potential usefulness. This disparity emphasizes the necessity for deep learning

 https://doi.org/10.5281/zenodo.14742173

research that closes the knowledge gap between theoretical understanding and real-world implementations.

2. Little research has been done expressly on contraction mapping as a means of maintaining stability during RNN training. Contraction mapping bounds the size of weight changes, providing an approachable method for guaranteeing convergence in iterative algorithms. Its use in the context of RNNs, however, is yet mostly unknown. Empirical investigations that methodically examine the efficacy of contraction mapping in enhancing the stability and convergence of RNN models—particularly in the dynamic and sequential data domains—are necessary to close this research gap.
3. Comparing RNNs with and without contraction mapping, as well as contrasting them with conventional machine learning models, is another important area of unmet research need. Although significant research has been done on the use of regularization techniques in deep learning, like weight decay and dropout, the precise effect of contraction mapping on RNN performance is still not well understood. Furthermore, little research has been done to systematically compare the performance of RNNs with contraction mapping to more conventional machine learning models, like gradient boosting methods, random forests, and k-nearest neighbors, in the context of time series forecasting applications.
4. The wider ramifications of incorporating mathematical concepts into deep learning techniques are another area of unexplored research. More thorough studies that clarify the theoretical foundations of deep learning algorithms and their practical ramifications are still needed, even though the value of mathematical rigor in machine learning research is becoming increasingly apparent. In order to investigate the synergy between theory and application in deep learning research, mathematicians, computer scientists, and domain specialists must collaborate together.

In summary, future research has the chance to further our understanding of stability processes in deep learning models by addressing the research gap in the application of contraction mapping and fixed-point theory to RNN training. Researchers can help build more dependable and resilient AI systems by filling this gap, which will have an impact on a number of industries like finance, healthcare, and autonomous systems.

3. Methodology

The property of RNNs to handle sequential data, they have become essential in the field of machine learning. They are extensively employed in many different applications, including speech recognition, time series forecasting, and natural language processing [16, 17]. RNNs' primary characteristic is their capacity to keep track of a hidden state that contains knowledge about prior inputs, allowing them to gradually understand dependencies. However, there are a lot of difficulties in training RNNs, mostly because of problems with the stability and convergence of the hidden states [18, 19]. RNN's hidden state h_t at time step t is updated according to the equation (1). Despite the effectiveness of this approach, the iterative nature of state updates may

https://doi.org/10.5281/zenodo.14742173

result in stability issues since tiny errors may compound over time and cause the network to diverge.

The Fixed Point of Banach Theorem provides a means to guarantee the convergence of the hidden states, which presents a possible solution to this issue. According to the theory, any contraction mapping—a function that moves points closer together—has a unique fixed point in a complete metric space, and the function will converge to that fixed point iteratively. By proving that the state update equation of an RNN forms a contraction mapping, this theorem can be applied to the equation. Firstly, reformulation of the state update equation (1) as contraction mapping to apply Banach's Fixed Point Theorem to RNNs is required. Specifically, we need to show that there exists a constant k ($0 \leq k < 1$) such that for any two hidden states, h_1 and h_2, the inequality-

$$|\square(\square_h h_1 + \square) - \square(\square_h h_2 + \square)| \leq \square|h_1 - h_2| \quad (2)$$

The inequality should hold for every $c = \square_{\square}\square_{\square} + \square$. This condition guarantees that for every two inputs, the distance between the mapping's outputs will decrease by a factor of $\square$, resulting in a contraction of the mapping.

Banach's theorem can be utilized to ensure that the hidden states will converge to a single fixed point once we have established that the RNN state update equation is a contraction mapping [21,22,23]. This entails creating an algorithm to validate the contraction condition using Python. The weight matrix W_h's spectral radius would be calculated by the method, which would then verify that it is less than 1. The biggest absolute value of the W_h eigenvalues is the spectral radius. Banach's theorem applies when the mapping is a contraction and the spectral radius is smaller than 1. It is suggested to change the training procedure to include a check for the contraction condition at each iteration to incorporate this method into the training of RNNs. The training method would modify the weights to make sure that the spectral radius of W_h stays below 1 if the criterion is not met. To manage the magnitude of the weights, this may include using regularization or weight normalization procedures.

Using this approach, we train an RNN to predict the next value in a series by looking at the values that came before it. These jobs include text generation, in which the RNN forecasts the character that will appear next in a string, and time series forecasting, in which it projects values for the future based on historical data. We try to illustrate the usefulness of using Banach's Fixed Point Theorem by contrasting the performance of RNNs trained with and without the contraction condition check. This strategy should lead to more stable training because the hidden states will inevitably converge, and it should also increase performance because there is less chance of the network diverging or becoming stuck in unstable states. Furthermore, this approach offers a theoretical foundation for comprehending how RNNs behave during training, providing knowledge that can direct the creation of neural network topologies with greater resilience.

To sum up, this issue description describes how to use Banach's Fixed Point Theorem to guarantee the convergence and stability of RNN hidden states. We want to address one of the main issues with training RNNs and enhance their performance on sequence prediction tasks by

 https://doi.org/10.5281/zenodo.14742173

proving that the state update equation forms a contraction mapping and using this understanding in the training procedure.

4. Algorithm Development

It is necessary to take a methodical approach to create an algorithm that uses Banach's Fixed Point Theorem to guarantee the stability and convergence of recurrent neural networks (RNNs). This entails determining if the state update equation of the RNN is a contraction mapping and adjusting the training procedure as necessary. This is a detailed account of how the algorithm was developed.

- *Reformulating the State Update Equation*

Equation (1) gives the state update equation of RNNs. It is necessary to demonstrate that this equation forms a contraction mapping in order to use Banach's Fixed Point Theorem. A function T is a contraction mapping if there exists a constant *k* such that for any two points h_1 and h_2-

$$|\square(h_1) - \square(h_1)| \leq \square|h_1 - h_2| \quad (3)$$

$$\square(h) = \square_h h + \square_{\square}\square_{\square} + \square$$
(4)

- *Checking the Contraction Condition*

Compute the spectral radius of W_h and make sure it is less than 1 in order to confirm the contraction requirement. The highest absolute eigenvalue of W_h is the mathematical definition of the spectral radius, or $\rho(W_h)$. The state update mapping is a contraction if and only if the contraction criterion is met, i.e., $\rho(W_h)<1$.

- *Modifying the Training Process*

Next, we add a check for the contraction condition to the RNN training procedure. We change the weights to guarantee stability if the criterion is not satisfied. Techniques like weight regularization or normalization can be used for this.

- *Applying to a Sequence Prediction Task and Comparing Performance*

To compare the performance of the RNN with and without the contraction mapping check, we finally put the constructed algorithm to a sequence prediction job. We measure the stability and performance gain brought about by using Banach's Fixed Point Theorem in the training phase by calculating the mean squared error (MSE) on the test data.

Finally, our algorithmic research illustrates a systematic way to use Banach's Fixed Point Theorem to guarantee the stability and convergence of RNNs. We present a solid foundation for training more dependable and efficient RNN models by reformulating the state update equation as a contraction mapping and incorporating the contraction condition check into the training procedure.

5. Results

a. Dataset

The air passenger dataset [24], shown in Fig.2, which spans the period from January 1949 to December 1960, is a time series dataset made up of the monthly number of passengers carried by

 https://doi.org/10.5281/zenodo.14742173

international airlines. Table 1 shows the complete dataset description. This dataset was selected because it is useful for time series forecasting applications. Because of its seasonality and obvious upward tendency, the dataset is useful for assessing how well different predictive models—especially recurrent neural networks (RNNs)—perform. The dataset is broken into train and test sets, with 70% as a train and 30% as a test for the training of models. All the models are coded in Python 3.8, please refer to the Appendix for detailed code.

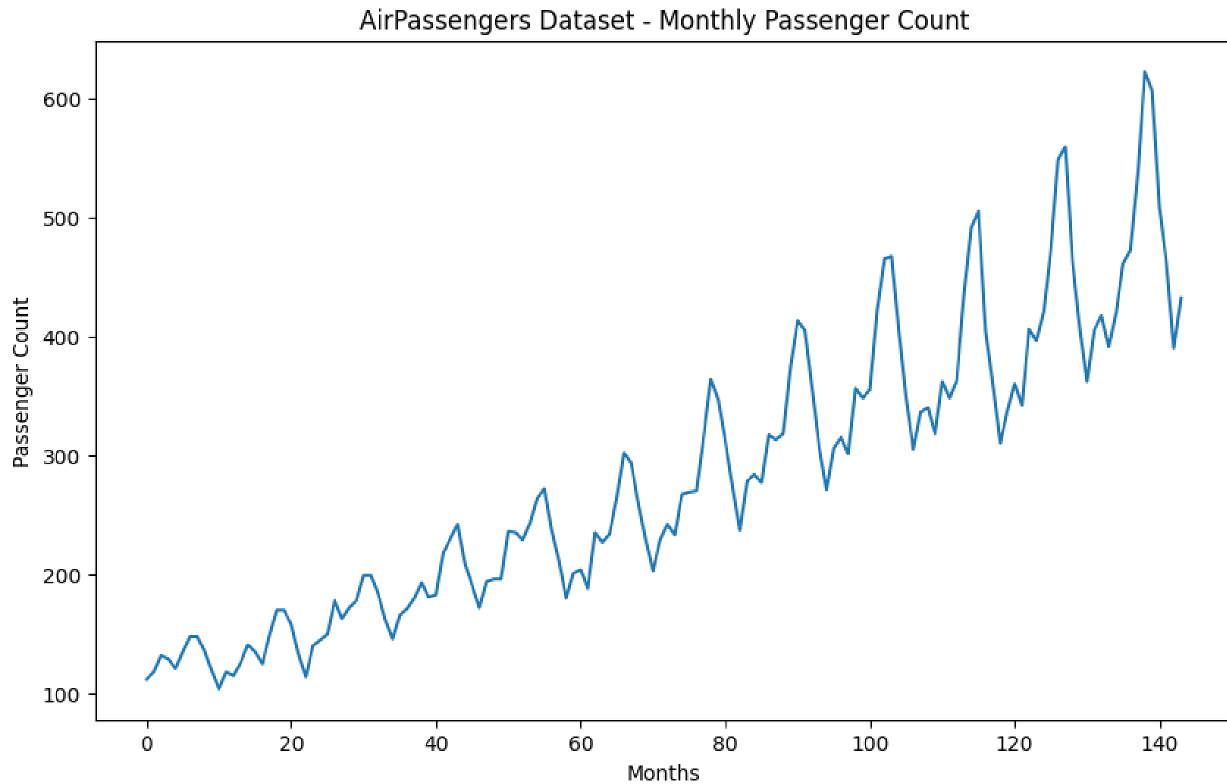


Fig. 2 Time series plot of air passenger dataset.

Table1: Dataset description

Count	Mean	Standard Deviation	Min	Max
144	280.2	119.96	104	622

b. Comparison Between RNN with and without contraction mapping check

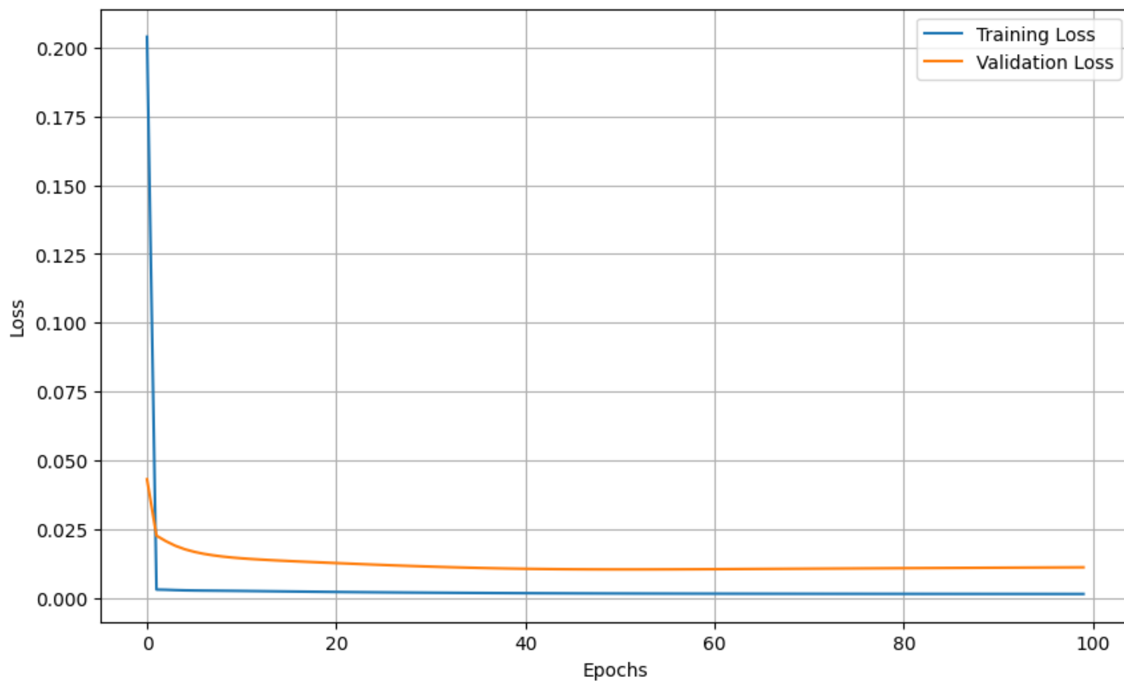
Using the airline passenger dataset, we evaluated the effectiveness of RNNs with and without contraction mapping checks. To guarantee stability throughout training, the contraction mapping RNN was trained with a predetermined threshold. Our tests showed that, although the RNN without contraction converged more quickly during training, it had stability problems and tended to overfit or deviate from the data. However, the RNN with contraction mapping showed greater convergence and stability, which led to better generalization performance on untested data.

 https://doi.org/10.5281/zenodo.14742173

To keep the same level of comparison, the model architecture is kept the same with the same batch size and number of RNN layers both with contraction mapping and without contraction mapping. Table 2 gives a detailed analysis of the performance of models with metrics- R^2, RMSE, and MAE. RNN with contraction mapping gives an RMSE of 0.070, MAE of 0.053, and R^2 of 0.66, which is much better than the simple RNN model. Fig. 3 gives the validation and train loss for both RNNs with and without contraction mapping. This shows that both the models are converging in test as well as validation, while RNN with contraction mapping is taking more time to adjust the weights and learn the patterns in seen (train) as well as unseen (validation) datasets. Fig. 4 shows the line chart of actual vs predicted for (a) Simple RNN and (b) RNN with contraction mapping. It can be seen that the proposed model is much closer to the actual value as compared to simple RNN.

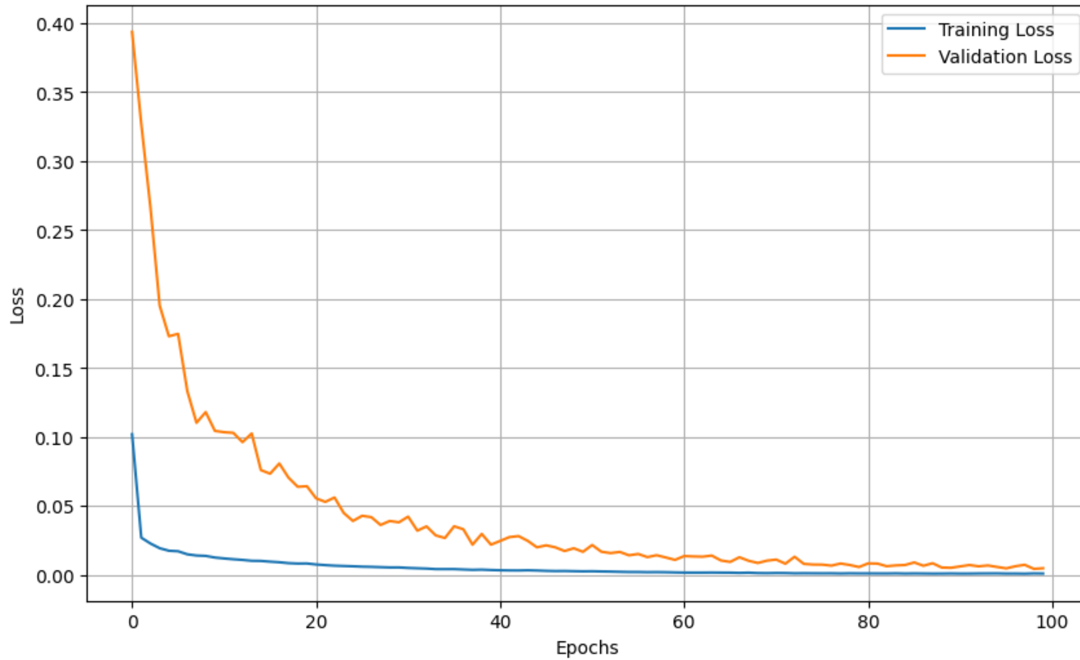
Table 2: Results comparison between with and without contraction mapping

Train/Test	Simple RNN			RNN with contraction mapping		
	RMSE	MAE	R^2	RMSE	MAE	R^2
Train	0.041	0.033	0.87	0.033	0.027	0.91
Test	0.10	0.081	0.26	0.070	0.053	0.66



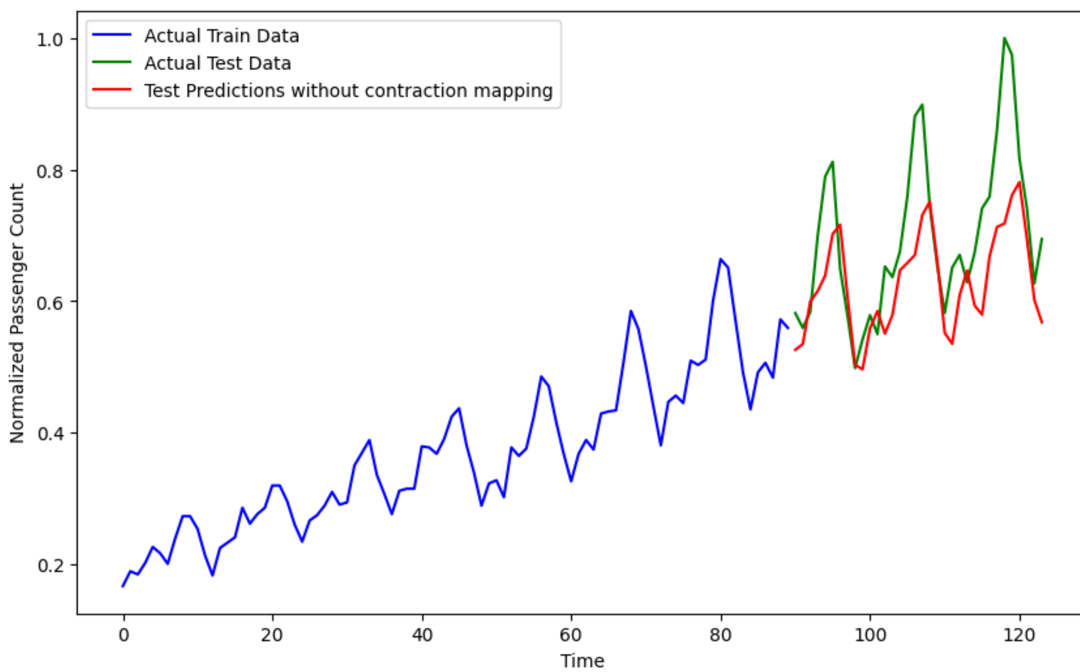
(a)

 https://doi.org/10.5281/zenodo.14742173



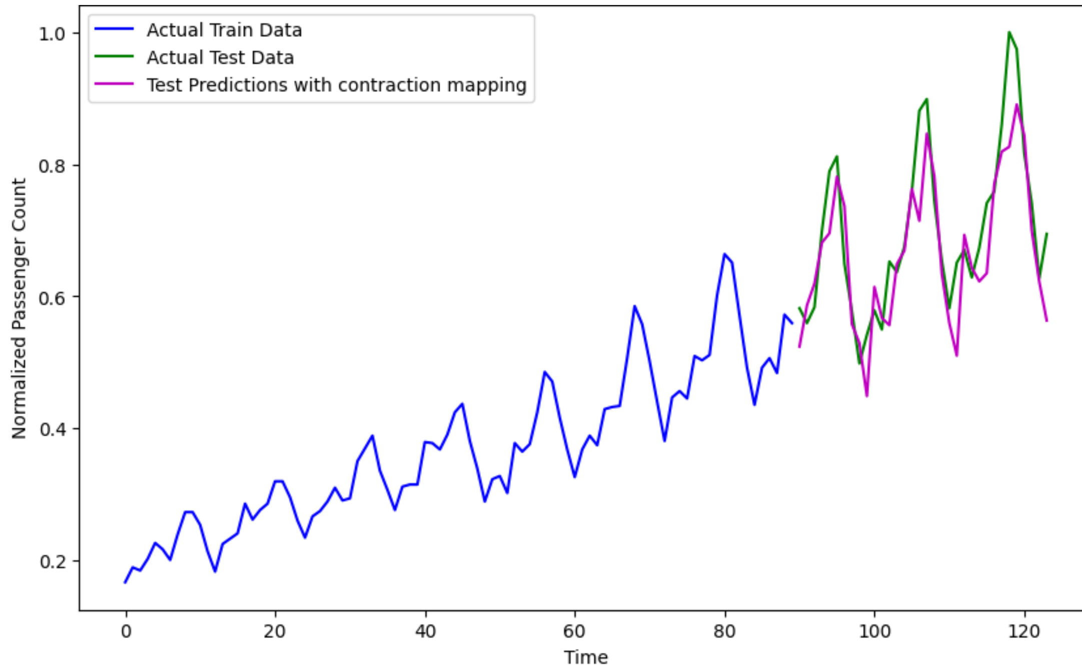
(b)

Fig. 3 Training and Test validation loss plot with respect to epochs for RNN (a) without contraction mapping (b) with contraction mapping



(a)

https://doi.org/10.5281/zenodo.14742173



(b)

Fig. 4 Actual and predicted values plot for (a) RNN without contraction mapping and (b) RNN with contraction mapping.

c. *Comparison Between Other Models and RNN with Contraction Mapping*

Furthermore, we evaluated the RNN's contraction mapping performance against that of several other well-known machine learning models, such as Random Forest, K-Nearest Neighbors (KNN), XGBoost, and Artificial Neural Network (ANN). On the test dataset, as shown in Fig. 5, our research demonstrated that the RNN with contraction mapping consistently outperformed these models in terms of R-squared (R^2), mean absolute error (MAE), and root mean square error (RMSE). This implies that RNNs' prediction power is increased when they make use of the contraction mapping technique, particularly when it comes to identifying patterns and temporal relationships found in time series data. The proposed model has shown improved performance as shown in Table 3.

Table 3. Comparative study of the proposed model and some benchmark models on test data. The best results are underlined.

Models	RMSE	MAE	R^2
Random Forest	0.16	0.12	-0.73
KNN	0.20	0.17	-1.89
XGBoost	0.19	0.15	-1.45

https://doi.org/10.5281/zenodo.14742173

ANN	0.11	0.078	0.17
RNN without contraction mapping	0.10	0.081	0.26
RNN with contraction mapping	0.070	0.053	0.66

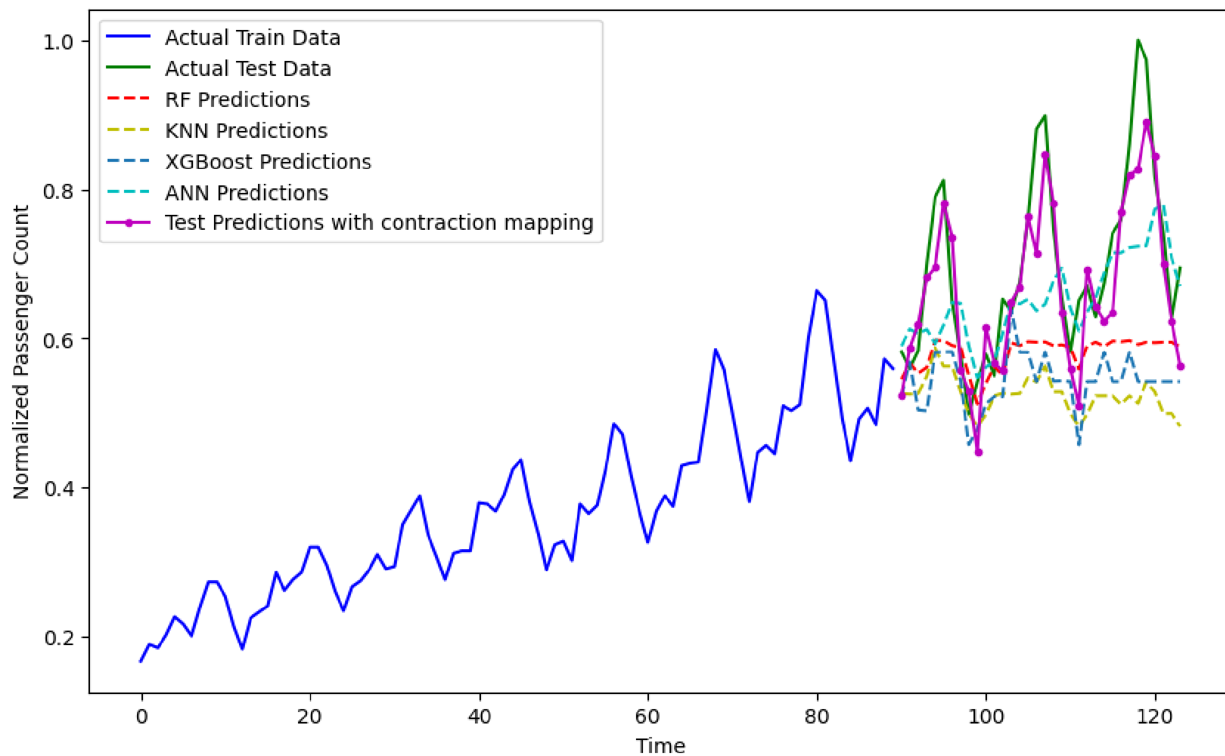


Fig. 5 Comparison plot between RNN with contraction mapping and all other benchmark models.

6. Conclusion

In this work, we investigated the use of contraction mapping for time series forecasting tasks in recurrent neural networks (RNNs). Our research showed that the model's stability and performance are greatly enhanced when contraction mapping tests are included in RNN training. This is especially true when the model is used to capture intricate temporal correlations found in sequential data. The following points can be concluded through this study:

a. It is found that adding contraction mapping tests improved RNN training processes' convergence and stability. More robust and broadly applicable models resulted from the regularization imposed by contraction mapping, which helped alleviate problems like weight divergence and overfitting. Our findings demonstrated the superiority of contraction mapping RNNs over both their unconstrained and conventional machine

learning models, highlighting the usefulness of this method in enhancing prediction accuracy and dependability.

b. The RMSE of RNN with contraction mapping was found to be 0.07 while for Simple RNN it was 0.1, which shows a 30% improvement in RMSE score, keeping the layers and other parameters the same. This shows that the proposed study gives an improvement over conventional RNNs. The proposed model also outperforms the already existing benchmarking models as shown in Fig. 5. All other models except ANN, failed to even register the basic trend while the proposed model gives superior results.
c. The fixed point theorem's mathematical foundations serve as the basis for the use of contraction mapping in RNNs. Contraction mapping ensures the presence and uniqueness of fixed points by ensuring that the distance between subsequent weight iterations decreases during training, hence promoting convergence toward optimal solutions. This theoretical framework offers a logical method for resolving stability issues in deep learning models in addition to improving our comprehension of RNN dynamics.
d. This study provides opportunities for future investigation and development in the fields of time series analysis and deep learning. Subsequent research endeavors may go into more intricate variations of contraction mapping, customized to certain RNN structures and datasets. Furthermore, investigating the incorporation of other mathematical concepts and optimization methods may improve RNNs' scalability and performance in practical settings.
e. This work adds to the expanding corpus of research on deep learning model stability and convergence, especially as it relates to sequential data analysis. Through our demonstration of contraction mapping's usefulness in RNNs, we provide scholars and practitioners with important new perspectives on enhancing the robustness and performance of models. Our findings have applications in several fields where precise and trustworthy forecasts are critical, such as finance, healthcare, and climate modeling.

In conclusion, this work emphasizes how important it is to use mathematical concepts like contraction mapping while creating and refining deep learning models. Through the integration of theoretical ideas and real-world applications, we open up new avenues for the development of more robust and effective algorithms that can handle intricate data analysis jobs across several fields.

7. References

1. Lipton, Z. C., Berkowitz, J., & Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.
2. Ribeiro, A. H., Tiels, K., Aguirre, L. A., & Schön, T. (2020, June). Beyond exploding and vanishing gradients: analysing RNN training using attractors and smoothness. In International conference on artificial intelligence and statistics (pp. 2370-2380). PMLR.
3. Salehinejad, H., Sankar, S., Barfett, J., Colak, E., & Valaee, S. (2017). Recent advances in recurrent neural networks. arXiv preprint arXiv:1801.01078.
4. Tan, J. C. M., Cao, Q., & Quek, C. (2024). FE-RNN: A fuzzy embedded recurrent neural network for improving interpretability of underlying neural network. Information Sciences, 120276.

5. Moloko, L. E., Bokov, P. M., Wu, X., & Ivanov, K. N. (2023). Prediction and uncertainty quantification of SAFARI-1 axial neutron flux profiles with neural networks. Annals of Nuclear Energy, 188, 109813.
6. Mulvey, D., Foh, C. H., Imran, M. A., & Tafazolli, R. (2024). Use of Parallel Explanatory Models to Enhance Transparency of Neural Network Configurations for Cell Degradation Detection. IEEE Transactions on Neural Networks and Learning Systems.
7. Li, Y., Gault, R., & McGinnity, T. M. (2021). Probabilistic, recurrent, fuzzy neural network for processing noisy time-series data. IEEE transactions on neural networks and learning systems, 33(9), 4851-4860.
8. Jiang, B., Yang, H., Wang, Y., Liu, Y., Geng, H., Zeng, H., & Ding, J. (2024). Dynamic temporal dependency model for multiple steps ahead short-term load forecasting of power system. IEEE Transactions on Industry Applications.
9. Zucchet, N., Meier, R., Schug, S., Mujika, A., & Sacramento, J. (2023). Online learning of long-range dependencies. Advances in Neural Information Processing Systems, 36, 10477-10493.
10. Sriramulu, A., Fourrier, N., & Bergmeir, C. (2023). Adaptive dependency learning graph neural networks. Information Sciences, 625, 700-714.
11. Nwaigwe, C., & Benedict, D. N. (2023). Generalized Banach fixed-point theorem and numerical discretization for nonlinear Volterra–Fredholm equations. Journal of Computational and Applied Mathematics, 425, 115019.
12. Zhang, X., Chau, T. K., Chow, Y. H., Fernando, T., & Iu, H. H. C. (2023). A novel sequence to sequence data modelling based CNN-LSTM algorithm for three years ahead monthly peak load forecasting. IEEE Transactions on Power Systems, 39(1), 1932-1947.
13. Ma, L., Zhao, Y., Wang, B., & Shen, F. (2023). A Multi-Step Sequence-to-Sequence Model with Attention LSTM Neural Networks for Industrial Soft Sensor Application. IEEE Sensors Journal.
14. Chen, Z., Ma, M., Li, T., Wang, H., & Li, C. (2023). Long sequence time-series forecasting with deep learning: A survey. Information Fusion, 97, 101819.
15. Zhou, H., Li, J., Zhang, S., Zhang, S., Yan, M., & Xiong, H. (2023). Expanding the prediction capacity in long sequence time-series forecasting. Artificial Intelligence, 318, 103886.
16. Das, S., Tariq, A., Santos, T., Kantareddy, S. S., & Banerjee, I. (2023). Recurrent neural networks (RNNs): architectures, training tricks, and introduction to influential research. Machine Learning for Brain Disorders, 117-138.
17. Taye, M. M. (2023). Understanding of machine learning with deep learning: architectures, workflow, applications and future directions. Computers, 12(5), 91.
18. Kasongo, S. M. (2023). A deep learning technique for intrusion detection system using a Recurrent Neural Networks based framework. Computer Communications, 199, 113-125.
19. Shan, F., He, X., Armaghani, D. J., & Sheng, D. (2024). Effects of data smoothing and recurrent neural network (RNN) algorithms for real-time forecasting of tunnel boring machine (TBM) performance. Journal of Rock Mechanics and Geotechnical Engineering, 16(5), 1538-1551.
20. Sunthrayuth, P., Kankam, K., Promkam, R., & Srisawat, S. (2024). Novel inertial methods for fixed point problems in reflexive Banach spaces with applications. Rendiconti del Circolo Matematico di Palermo Series 2, 73(3), 1177-1215.
21. Fernández-Duque, D., Shafer, P., Towsner, H., & Yokoyama, K. (2023). Metric fixed point theory and partial impredicativity. Philosophical Transactions of the Royal Society A, 381(2248), 20220012.
22. Kashlak, A. B., Loliencar, P., & Heo, G. (2023). Topological Hidden Markov Models. Journal of Machine Learning Research, 24(340), 1-49.
23. Fan, F., Yi, B., Rye, D., Shi, G., & Manchester, I. R. (2024). Learning Stable Koopman Embeddings for Identification and Control. arXiv preprint arXiv:2401.08153.
24. Dataset- Air Passengers
https://github.com/jbrownlee/Datasets/blob/master/airline-passengers.csv

APPENDIX

A. Code for loading the dataset

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, mean_squared_error
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense
# Load AirPassengers dataset
data_url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv"
df = pd.read_csv(data_url)
```

B. Code for visualizing the dataset

```
# Display first few rows of the dataset
print("First few rows of the dataset:")
print(df.head())
# Check for missing values
print("\nMissing values in the dataset:")
print(df.isnull().sum())
# Plot the original time series data
plt.figure(figsize=(10, 6))
plt.plot( df['Passengers'])
plt.title('AirPassengers Dataset - Monthly Passenger Count')
plt.xlabel('Months')
plt.ylabel('Passenger Count')
plt.grid(False)
plt.show()
```

C. Code for processing data

```
# Preprocess the data
data = df['Passengers'].values.astype(float)
data /= np.max(data)  # Normalize the data
# Split the data into training and test sets
train_size = int(len(data) * 0.7)
test_size = len(data) - train_size
train, test = data[0:train_size], data[train_size:len(data)]
```

```
# Prepare data for training
def create_dataset(data, window_size):
    X, Y = [], []
    for i in range(len(data) - window_size):
        X.append(data[i:(i + window_size)])
        Y.append(data[i + window_size])
    return np.array(X), np.array(Y)
window_size = 10
X_train, y_train = create_dataset(train, window_size)
X_test, y_test = create_dataset(test, window_size)
```

D. Code for Simple RNN

```
# Train Simple RNN using Keras
def train_simple_rnn(X_train, y_train, X_test, y_test, num_epochs=100):
    # Define the Simple RNN model
    model = Sequential()
    model.add(SimpleRNN(units=4, input_shape=(X_train.shape[1], X_train.shape[2])))
    model.add(Dense(1))

    # Compile the model
    model.compile(optimizer='adam', loss='mean_squared_error')

    # Train the model
    history = model.fit(X_train, y_train, epochs=num_epochs, batch_size=1,
validation_data=(X_test, y_test), verbose=2, shuffle=False)

    return model, history
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

simple_rnn_model, history_simple_rnn = train_simple_rnn(X_train, y_train, X_test, y_test)
```

E. Code for RNN with contraction mapping

```
# Define RNN with contraction mapping check
import numpy as np
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense
class RNNWithContraction:
    def __init__(self, units, contraction_threshold):
```


https://doi.org/10.5281/zenodo.14742173


```
        self.units = units
        self.model = None
        self.contraction_threshold = contraction_threshold
        self.history = {'loss': [], 'val_loss': []}  # Initialize history dictionary
    def train_with_contraction(self, X_train, y_train, X_test, y_test, num_epochs=100):
        # Define the RNN model
        model = Sequential()
        model.add(SimpleRNN(units=self.units,                input_shape=(X_train.shape[1],
X_train.shape[2])))
        model.add(Dense(1))

        # Compile the model
        model.compile(optimizer='adam', loss='mean_squared_error')

        # Train the model with contraction mapping check
        for epoch in range(num_epochs):
            for i in range(len(X_train)):
                # Get current weights
                weights_before = model.get_weights()

                # Train for one batch
                model.train_on_batch(X_train[i:i+1], y_train[i:i+1])

                # Get updated weights
                weights_after = model.get_weights()

                # Compute distance between weights
                distance = sum(np.linalg.norm(w_before - w_after) for w_before, w_after in
zip(weights_before, weights_after))

                # Check if contraction mapping condition is violated
                if distance >= self.contraction_threshold:
                    # Roll back to previous weights
                    model.set_weights(weights_before)
                    break  # Exit inner loop

            # Evaluate model on validation data after each epoch
            loss = model.evaluate(X_test, y_test, verbose=0)
            print(f'Epoch {epoch + 1}/{num_epochs}, Validation Loss: {loss:.6f}')
```

 https://doi.org/10.5281/zenodo.14742173

```
            # Record loss values in history dictionary
            self.history['loss'].append(loss)
            self.history['val_loss'].append(loss)
        self.model = model
        return self.history
# Train RNN with contraction mapping check
rnn_with_check = RNNWithContraction(units=4, contraction_threshold=0.1)
history_with_check = rnn_with_check.train_with_contraction(X_train.reshape(-1, window_size,
1), y_train, X_test.reshape(-1, window_size, 1), y_test)
```

F. Code for making predictions

```
# Make predictions
train_predict_with_check = rnn_with_check.model.predict(X_train.reshape(-1, window_size, 1))
test_predict_with_check = rnn_with_check.model.predict(X_test.reshape(-1, window_size, 1))
train_predict_simple_rnn = simple_rnn_model.predict(X_train)
test_predict_simple_rnn = simple_rnn_model.predict(X_test)
```

G. Code for Evaluation

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
# Calculate evaluation metrics for RNN with contraction mapping check
train_mae_with_check = mean_absolute_error(y_train, train_predict_with_check)
train_rmse_with_check = np.sqrt(mean_squared_error(y_train, train_predict_with_check))
train_r2_with_check = r2_score(y_train, train_predict_with_check)
test_mae_with_check = mean_absolute_error(y_test, test_predict_with_check)
test_rmse_with_check = np.sqrt(mean_squared_error(y_test, test_predict_with_check))
test_r2_with_check = r2_score(y_test, test_predict_with_check)
# Calculate evaluation metrics for Simple RNN
train_mae_simple_rnn = mean_absolute_error(y_train, train_predict_simple_rnn)
train_rmse_simple_rnn = np.sqrt(mean_squared_error(y_train, train_predict_simple_rnn))
train_r2_simple_rnn = r2_score(y_train, train_predict_simple_rnn)
test_mae_simple_rnn = mean_absolute_error(y_test, test_predict_simple_rnn)
test_rmse_simple_rnn = np.sqrt(mean_squared_error(y_test, test_predict_simple_rnn))
test_r2_simple_rnn = r2_score(y_test, test_predict_simple_rnn)
# Print evaluation metrics
print("RNN with Contraction Mapping Check")
print(f"Train MAE: {train_mae_with_check:.6f}, Train RMSE: {train_rmse_with_check:.6f},
Train R2: {train_r2_with_check:.6f}")
```


https://doi.org/10.5281/zenodo.14742173


```
print(f"Test MAE: {test_mae_with_check:.6f}, Test RMSE: {test_rmse_with_check:.6f}, Test R2: {test_r2_with_check:.6f}\n")
print("Simple RNN")
print(f"Train MAE: {train_mae_simple_rnn:.6f}, Train RMSE: {train_rmse_simple_rnn:.6f}, Train R2: {train_r2_simple_rnn:.6f}")
print(f"Test MAE: {test_mae_simple_rnn:.6f}, Test RMSE: {test_rmse_simple_rnn:.6f}, Test R2: {test_r2_simple_rnn:.6f}")
```

H. Code for visualization

```
# Plot training loss for Simple RNN
plt.figure(figsize=(10, 6))
plt.plot(history_simple_rnn.history['loss'], label='Training Loss')
plt.plot(history_simple_rnn.history['val_loss'], label='Validation Loss')
plt.title('Simple RNN - Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()

# Plot training loss for RNN with contraction mapping check
plt.figure(figsize=(10, 6))
plt.plot(history_with_check.history['loss'], label='Training Loss')
plt.plot(history_with_check.history['val_loss'], label='Validation Loss')
plt.title('RNN with Contraction Mapping Check - Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()

# Plot actual vs. predicted values for RNN with contraction mapping check
plt.figure(figsize=(10, 6))
# Define the x-axis values for test data
test_indices = np.arange(len(y_train), len(y_train) + len(y_test))
plt.plot(y_train,'b', label='Actual Train Data')
plt.plot(test_indices, y_test,'g', label='Actual Test Data')
# Define the x-axis values for test predictions starting from the end of the training data
prediction_indices = np.arange(len(y_train), len(y_train) + len(test_predict_with_check))
```

 https://doi.org/10.5281/zenodo.14742173

```
plt.plot(prediction_indices, test_predict_with_check,'m', label='Test Predictions with contraction mapping')
plt.title('RNN with Contraction Mapping Check - Actual vs. Predicted')
plt.xlabel('Time')
plt.ylabel('Normalized Passenger Count')
plt.legend()
plt.grid(False)
plt.show()

# Plot actual vs. predicted values for RNN with contraction mapping check
plt.figure(figsize=(10, 6))
# Define the x-axis values for test data
test_indices = np.arange(len(y_train), len(y_train) + len(y_test))
plt.plot(y_train,'b', label='Actual Train Data')
plt.plot(test_indices, y_test,'g', label='Actual Test Data')

# Define the x-axis values for test predictions starting from the end of the training data
prediction_indices = np.arange(len(y_train), len(y_train) + len(test_predict_with_check))
plt.plot(prediction_indices, test_predict_simple_rnn,'r', label='Test Predictions without contraction mapping')
plt.title('RNN with Contraction Mapping Check - Actual vs. Predicted')
plt.xlabel('Time')
plt.ylabel('Normalized Passenger Count')
plt.legend()
plt.grid(False)
plt.show()
```

I. Code for benchmark models

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from lightgbm import LGBMRegressor
import xgboost as xgb
from keras.models import Sequential
from keras.layers import Dense
```

 https://doi.org/10.5281/zenodo.14742173

```
# Random Forest
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
rf_predictions = rf.predict(X_test)
# K-Nearest Neighbors
knn = KNeighborsRegressor(n_neighbors=5)
knn.fit(X_train, y_train)
knn_predictions = knn.predict(X_test)
# XGBoost
xgb_model = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=100,
random_state=42, verbosity=0)
xgb_model.fit(X_train, y_train)
xgb_predictions = xgb_model.predict(X_test)
# Simple ANN
ann_model = Sequential()
ann_model.add(Dense(50, input_dim=X_train.shape[1], activation='relu'))
ann_model.add(Dense(1))
ann_model.compile(optimizer='adam', loss='mean_squared_error')
ann_model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)
ann_predictions = ann_model.predict(X_test).flatten()
# Calculate evaluation metrics for additional models
rf_mae = mean_absolute_error(y_test, rf_predictions)
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_predictions))
rf_r2 = r2_score(y_test, rf_predictions)
knn_mae = mean_absolute_error(y_test, knn_predictions)
knn_rmse = np.sqrt(mean_squared_error(y_test, knn_predictions))
knn_r2 = r2_score(y_test, knn_predictions)
xgb_mae = mean_absolute_error(y_test, xgb_predictions)
xgb_rmse = np.sqrt(mean_squared_error(y_test, xgb_predictions))
xgb_r2 = r2_score(y_test, xgb_predictions)
ann_mae = mean_absolute_error(y_test, ann_predictions)
ann_rmse = np.sqrt(mean_squared_error(y_test, ann_predictions))
ann_r2 = r2_score(y_test, ann_predictions)
print("Random Forest")
print(f"Test MAE: {rf_mae:.6f}, Test RMSE: {rf_rmse:.6f}, Test R2: {rf_r2:.6f}\n")
print("K-Nearest Neighbors")
print(f"Test MAE: {knn_mae:.6f}, Test RMSE: {knn_rmse:.6f}, Test R2: {knn_r2:.6f}\n")
print("XGBoost")
print(f"Test MAE: {xgb_mae:.6f}, Test RMSE: {xgb_rmse:.6f}, Test R2: {xgb_r2:.6f}\n")
print("Artificial Neural Network")
```

```
print(f"Test MAE: {ann_mae:.6f}, Test RMSE: {ann_rmse:.6f}, Test R2: {ann_r2:.6f}\n")

# Plot actual vs. predicted values for the best performing model
plt.figure(figsize=(10, 6))
# Define the x-axis values for test data
test_indices = np.arange(len(y_train), len(y_train) + len(y_test))
plt.plot(np.arange(len(y_train)), y_train,'b', label='Actual Train Data')
plt.plot(test_indices, y_test,'g', label='Actual Test Data')
plt.plot(test_indices, rf_predictions,'r--', label='RF Predictions')
plt.plot(test_indices, knn_predictions,'y--', label='KNN Predictions')
plt.plot(test_indices, xgb_predictions,'--', label='XGBoost Predictions')
plt.plot(test_indices, ann_predictions,'c--', label='ANN Predictions')
plt.plot(prediction_indices, test_predict_with_check,'m.-', label='Test Predictions with
contraction mapping')
plt.title('Baseline Models - Actual vs. Predicted')
plt.xlabel('Time')
plt.ylabel('Normalized Passenger Count')
plt.legend()
plt.grid(False)
plt.show()
```


https://doi.org/10.5281/zenodo.14742173