4/16/2024

Analyzing Seattle Weather

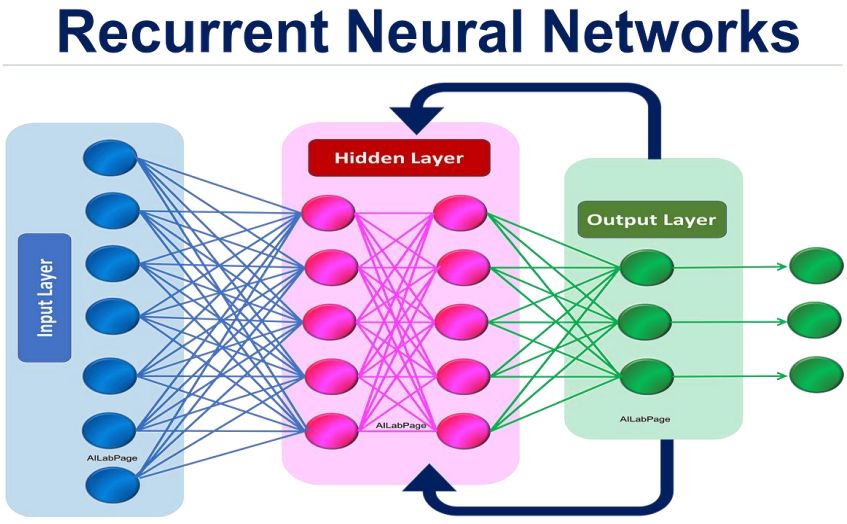
MSA 8600 Deep Learning Analytics (Spring 2024)

Naveen Seelam	**:**	**002794970**
Rahul Kumar	**:**	**002779804**
Dipak Bhattarai	**:**	**002776875**
An Nguyen	**:**	**900935293**
Vaishnavi Mada	**:**	**002775010**

J. Mack Robinson College of Business
Georgia State University

Contents

Analyzing Seattle Weather Patterns Using Recurrent Neural Networks



Introduction

In the realm of data science, sequential data sets, which include series of data points indexed in time order, are a fundamental component for analyzing temporal patterns and forecasting future events. This is particularly pertinent in meteorological studies, where sequential data plays a crucial role in understanding climate behaviors and predicting weather conditions. The Seattle weather dataset, comprising daily observations of various atmospheric variables, offers a rich source for such temporal analysis.

This project leverages Recurrent Neural Networks (RNNs), a class of artificial neural networks designed to recognize patterns in sequences of data such as time series or text. RNNs are uniquely capable of processing data inputs with sequential dependence, making them ideal for tasks such as weather prediction where current conditions are influenced by past events.

The project initiated with the meticulous preparation of Seattle weather data, which includes daily metrics such as precipitation, temperatures, and wind speeds. The dataset undergoes several cleaning steps to ensure high-quality inputs, such as removal of missing values, normalization, and careful feature selection, followed by splitting into training and validation sets. Scaling is applied to optimize the training process of the neural network.

An RNN model is then developed to handle the sequential nature of weather data. The model is trained using these sequences, with the training progress monitored through loss metrics visualized over epochs. The effectiveness of the model is assessed by comparing its predictions against actual weather observations on both training and validation datasets. The project concludes with detailed visual comparisons for key weather parameters and provides insights and recommendations for future applications or studies involving RNNs in complex, time-sensitive datasets like weather patterns.

Dataset

The seattle-weather.csv dataset, Weather Prediction using RNN (kaggle.com) consists of 1,461 entries covering daily weather observations from January 1, 2012, to December 31, 2015. Each entry in the dataset corresponds to a single day and includes several meteorological variables critical for various analyses. The dataset features columns such as:

Date: which provides the specific day of the observation,

Precipitation: indicating the amount of rainfall in millimeters,

temp_max: which records the day's highest temperatures in degrees Celsius.

temp_min: which records the day's lowest temperatures in degrees Celsius.

wind: detailing the wind speed in meters per second.

weather: describing the general weather condition (e.g., rain, sunny, cloudy).

This structured dataset offers a comprehensive overview of the weather patterns in Seattle over the specified four-year period, making it a valuable resource for climatic studies, trend analysis, or training predictive models.

Data Splitting

Data splitting is a crucial process in data analysis, particularly in the context of building predictive models such as machine learning algorithms. The basic idea is to divide the original dataset into distinct subsets to train and evaluate the performance of these models effectively and to ensure that they generalize well to new, unseen data.

Training Set: The training set is a subset of the dataset used to train a model, allowing it to learn and recognize patterns and relationships within the data.

In our project, we allocated 70% of the data, equivalent to 1023 records, as the training set

Validation Set: The validation set is a subset of data used to evaluate the model's performance during training, allowing for the tuning of hyperparameters and the prevention of overfitting.

In our project, we allocated 30% of the data, equivalent to 438 records, as the validation set.

Scaling the data

Scaling the data is a crucial preprocessing step in many machine learning algorithms, particularly when the input features have varying scales or units. This process involves transforming the features into a specific range, typically 0 to 1, to ensure that no single feature dominates the model due to its scale. This normalization allows for a more stable and faster convergence during training, as it ensures that all features contribute equally to the model's learning process. By maintaining consistent scales across all features, the algorithm can perform more effectively and is less sensitive to the scale of the input data.

Steps:

- **Initialization of Scaler:** This creates an instance of the MinMaxScaler, which will be used to scale the data to a range of 0 to 1.
- **Selecting Features:** features = ['temp_max', 'temp_min', 'precipitation', 'wind'] - A list of feature names is defined, representing the columns in the dataset that will be scaled.
- **Scaling Training Data:** The fit_transform method first fits the scaler to the training data (calculating the minimum and maximum values of each feature) and then transforms the training data by scaling it based on these values.
- **Scaling Validation Data:** The scaler, which was fitted to the training data, is now used to transform the validation data. This ensures that the same scaling transformation is applied to both training and validation datasets, making them directly comparable during model evaluation. The transform method uses the same scale set from the training data to ensure consistency across datasets.

Creating sequences

In time series analysis, especially when using Recurrent Neural Networks (RNNs), it's crucial to convert the data into sequences. Each sequence represents a specific window of consecutive data points. These sequences, which include both inputs (X) and targets (y), allow the model to learn from the past information to predict future data points. The sequence creation process involves selecting a fixed number of consecutive data points (defined by time_step) as input and the subsequent data point as the target output. This technique helps the model in capturing the temporal dependencies present in the data.

Steps:

- **Define the function** create_sequences

- iterate from time_step to the length of the dataset:
- Call create_sequences with the scaled training dataset and a specified time_step to generate X_train and y_train.
- call create_sequences with the scaled validation dataset and the same time_step to generate X_val and y_val. These are used for validating the model performance during or after training.

Model

The RNN model was designed with a sequential layout comprising three RNN layers interspersed with Dropout layers to prevent overfitting:

- **Input Layer:** The first SimpleRNN layer with 50 units, using tanh as the activation function, and return_sequences=True to pass sequences to the next layer. The input shape was determined by the time_step and the number of features.
- **Hidden Layers:** Two additional SimpleRNN layers, each with 50 units and tanh activation, followed by a Dropout layer with a dropout rate of 0.2 to reduce overfitting.
- **Output Layer:** A Dense layer with units equal to the number of features in the dataset, activated by the relu function to predict continuous output values.

Model Compilation and Training: The model was compiled using the Adam optimizer and mean squared error (MSE) as the loss function. It was trained over 50 epochs with a batch size of 32. The training process included both training and validation datasets to monitor and minimize overfitting.

```
Epoch 1/50
32/32 [==============================] - 6s 40ms/step - loss: 0.1581 - val_loss: 0.0819
Epoch 2/50
32/32 [==============================] - 1s 17ms/step - loss: 0.1075 - val_loss: 0.0498
Epoch 3/50
32/32 [==============================] - 1s 19ms/step - loss: 0.0975 - val_loss: 0.0529
Epoch 4/50
32/32 [==============================] - 0s 14ms/step - loss: 0.0781 - val_loss: 0.0444
Epoch 5/50
32/32 [==============================] - 0s 14ms/step - loss: 0.0656 - val_loss: 0.0317
Epoch 6/50
32/32 [==============================] - 1s 16ms/step - loss: 0.0517 - val_loss: 0.0352
Epoch 7/50
32/32 [==============================] - 0s 14ms/step - loss: 0.0433 - val_loss: 0.0209
Epoch 8/50
32/32 [==============================] - 0s 13ms/step - loss: 0.0371 - val_loss: 0.0177
Epoch 9/50
32/32 [==============================] - 1s 30ms/step - loss: 0.0325 - val_loss: 0.0156
Epoch 10/50
32/32 [==============================] - 0s 13ms/step - loss: 0.0277 - val_loss: 0.0165
Epoch 11/50
32/32 [==============================] - 1s 20ms/step - loss: 0.0267 - val_loss: 0.0154
Epoch 12/50
32/32 [==============================] - 1s 24ms/step - loss: 0.0253 - val_loss: 0.0149
```

Plot - Loss over Epochs

The objective of this analysis was to evaluate the training process of a Recurrent Neural Network (RNN) designed to forecast weather conditions in Seattle, focusing specifically on the model's loss progression during training and validation phases. This assessment helps identify overfitting, underfitting, and the optimal point for stopping training.

The RNN was trained on a dataset comprising several weather-related features, with the goal of minimizing the difference between the predicted and actual values, quantified as loss. The model was trained for 50 epochs, and both training and validation losses were recorded at each epoch. The performance of the model was then visualized through a line plot to compare these losses across epochs.

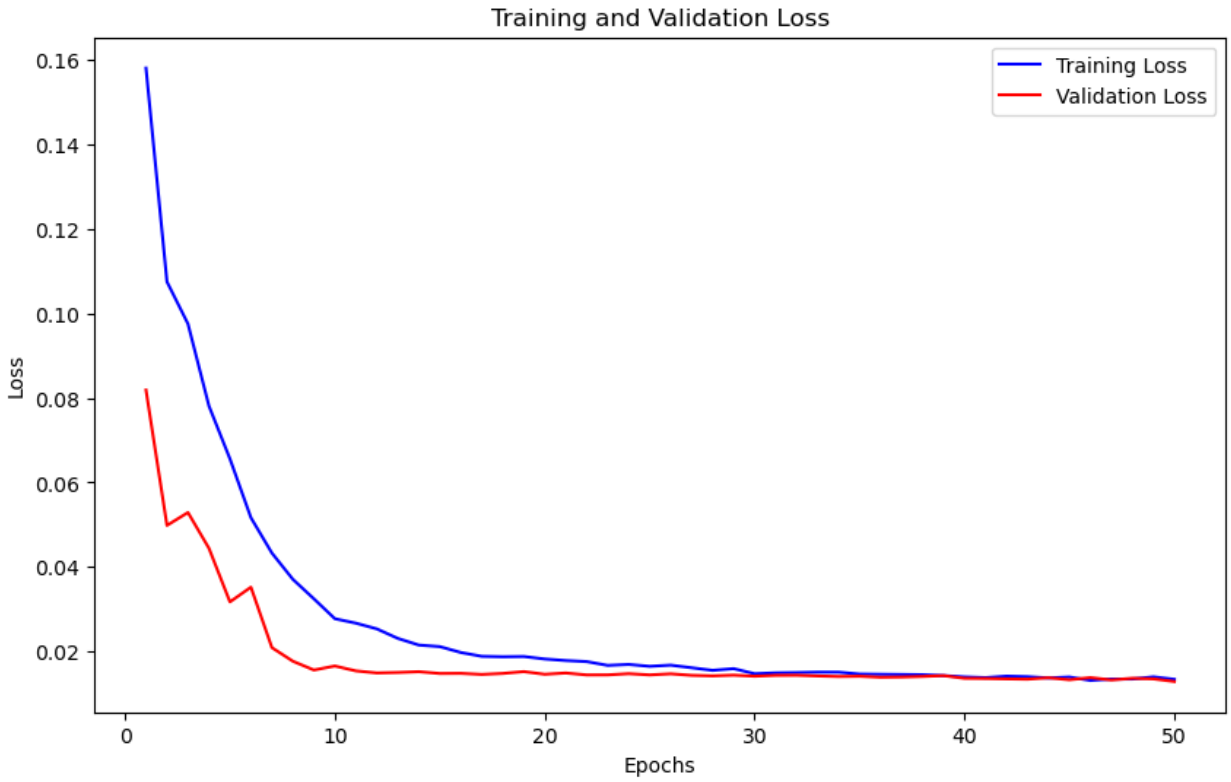
Loss Plot Analysis:

Plot Configuration: A line plot was created using Matplotlib, displaying training loss (in blue) and validation loss (in red) against the number of epochs.

Loss Trends: The training loss consistently decreased, indicating that the model was effectively learning from the dataset. The validation loss initially decreased alongside the training loss, suggesting that the model was generalizing well to new data.

Overfitting Indicators: A key aspect of the analysis was monitoring for divergence between training and validation loss, which could indicate overfitting. If the validation loss begins to increase while training loss continues to decrease, it suggests that the model is starting to memorize the training data rather than learning to generalize from it.

The plotted losses provided a clear visual representation of the model's learning process over the epochs. Consistently decreasing loss values confirmed that the model was learning effectively. However, careful attention was required to detect any signs of overfitting, as indicated by a divergence between training and validation loss. Based on this plot, decisions could be made about potential adjustments in the model's training duration, complexity, or regularization techniques to optimize performance and prevent overfitting.



Prediction

Comparative Analysis of Predicted vs. Actual Weather Features

This segment of the project focuses on the comparative analysis of predicted versus actual values for key weather features - maximum temperature, minimum temperature, precipitation, and wind speed. The primary goal was to visually assess the performance of the developed Recurrent Neural Network (RNN) model in forecasting these features across both training and validation datasets.

The analysis utilized custom plotting functions to display the actual and predicted values of each weather feature on the original scale. These plots provide a clear visual representation of the model's predictive accuracy and its ability to replicate the temporal patterns observed in the actual data.

To ensure the comparisons were meaningful and interpretable, predictions and actual values were descaled back to their original units using the inverse transformation of the previously applied scaling method. This step was crucial for accurate visual comparison.

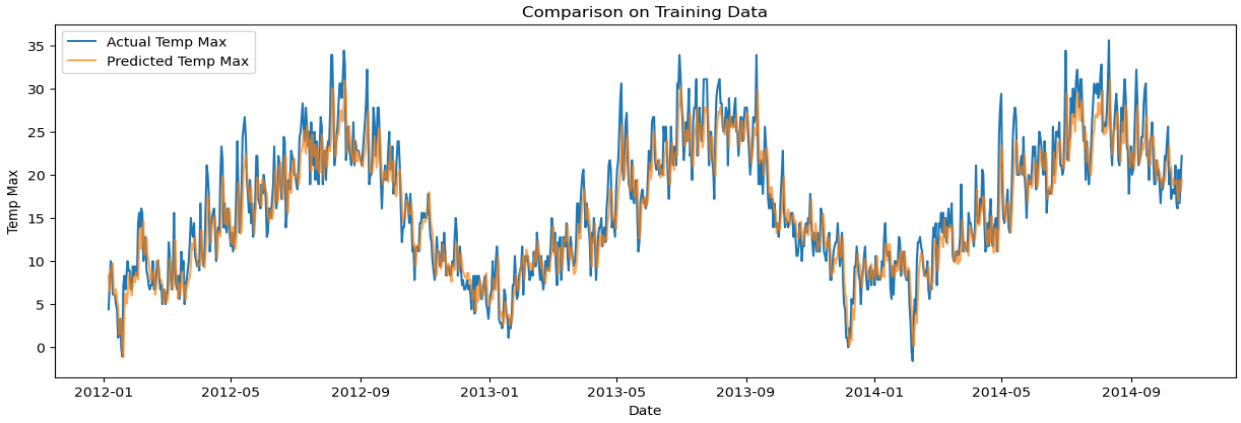
Visualization Process:

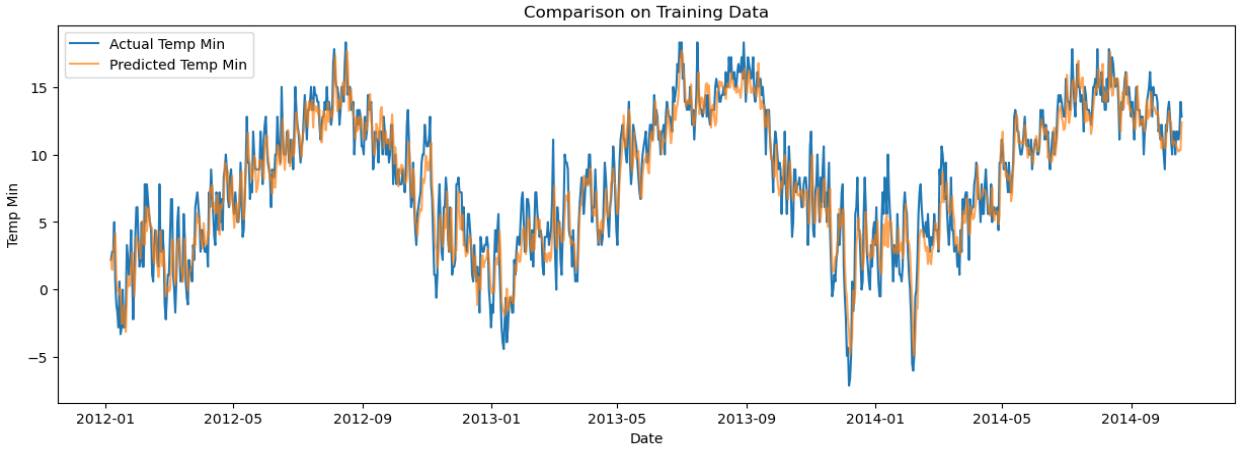
Training Set Visualization: For each feature (maximum temperature, minimum temperature, precipitation, wind speed), a plot was generated comparing the actual data with the RNN predictions within the training dataset. These plots were titled "Comparison on Training Data" and included data from the relevant time steps onward to match the predictions with corresponding actuals.

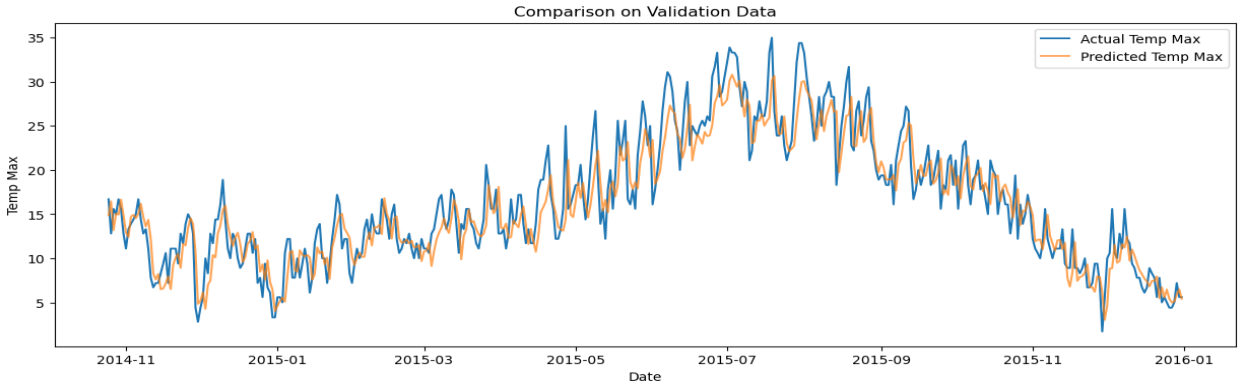
Validation Set Visualization: Similarly, for the validation dataset, plots were generated for each feature. These plots were titled "Comparison on Validation Data" and also started from the appropriate time step to align predictions with their actual counterparts.

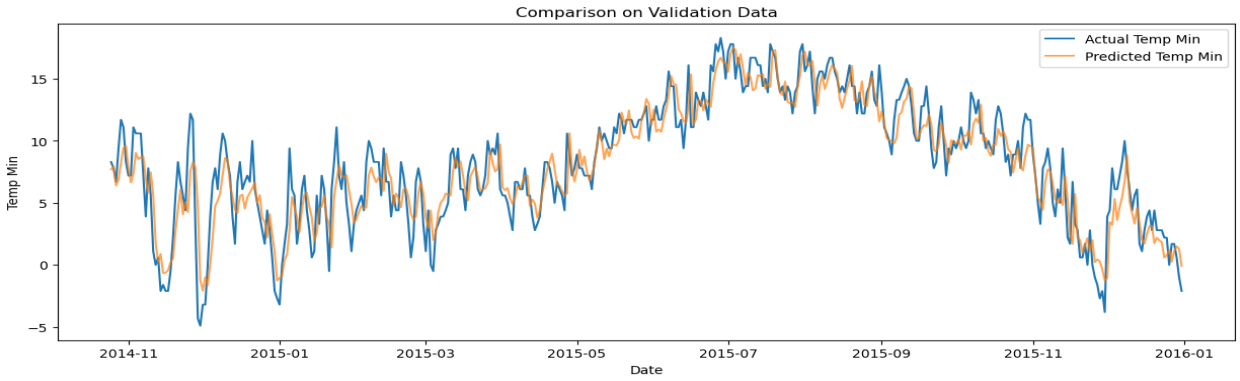
Results and Discussion:

Temperature Predictions (Max and Min): The plots displayed a close match between the predicted and actual temperature values, with the model capturing daily fluctuations effectively. However, certain peaks and troughs presented challenges, particularly on extremely cold or hot days.

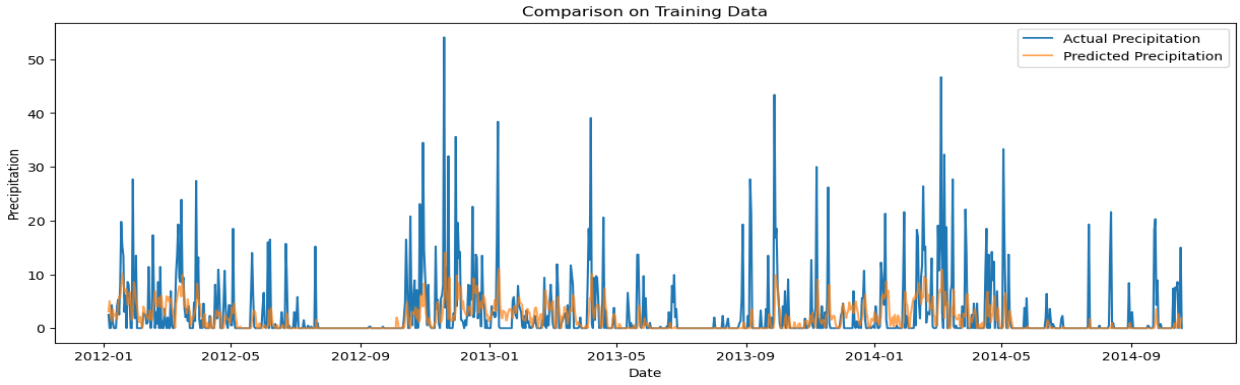


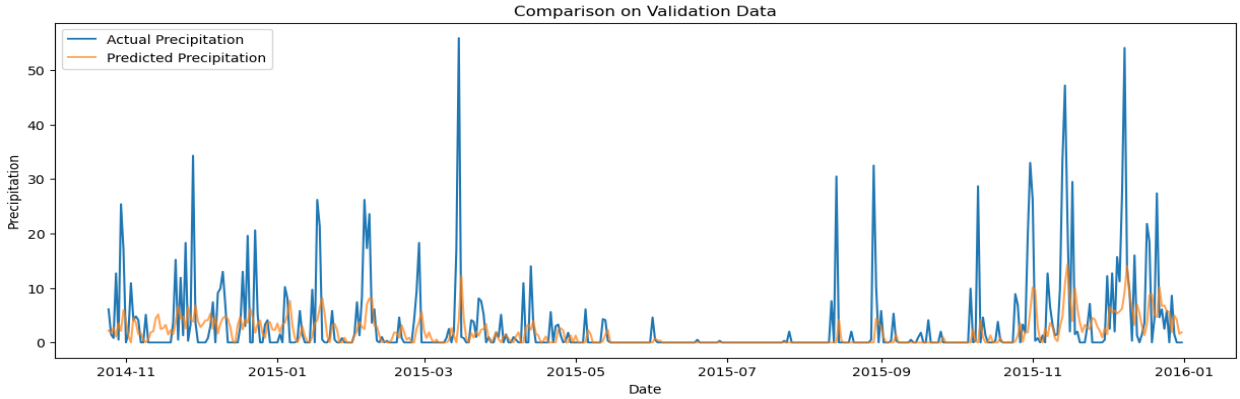




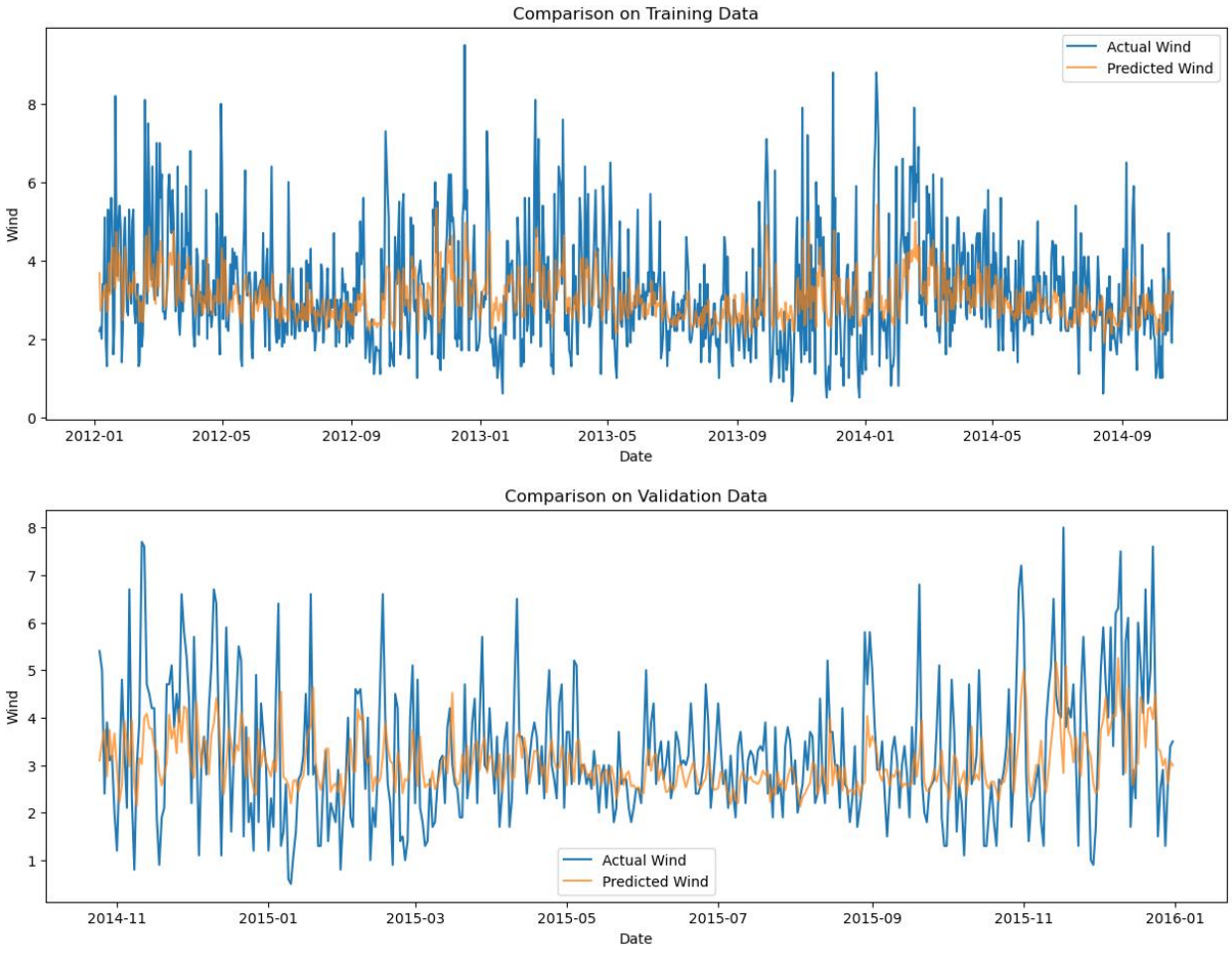


Precipitation: This feature showed variability in predictive accuracy, with the model capturing general trends but struggling with the intensity and timing of precipitation events.





Wind Speed: Predictions for wind speed closely followed the actual values, though some discrepancies were noted during periods of sudden changes in wind intensity.



The visual comparison of predicted versus actual weather data demonstrated the RNN model's capability to forecast complex weather patterns with a reasonable degree of accuracy. While the model performed well across temperature and wind predictions, it indicated room for improvement in predicting precipitation, suggesting a potential area for further model refinement.

Conclusion

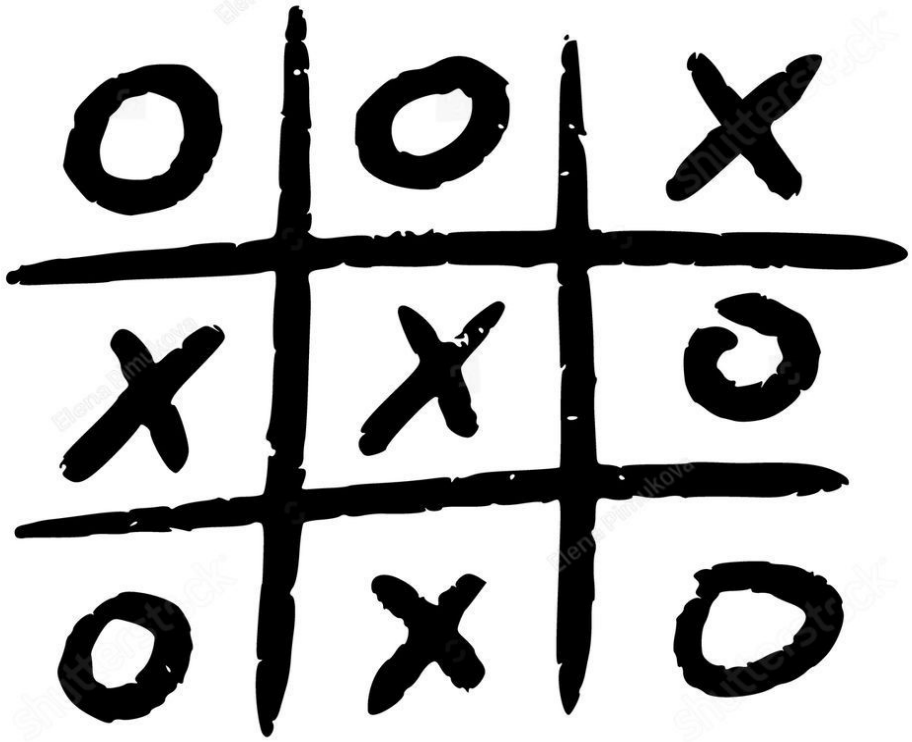
The project successfully demonstrated the capability of a Recurrent Neural Network (RNN) to forecast key weather parameters such as maximum and minimum temperatures, precipitation, and wind speed, using data from the Seattle area. The model showed promising results, particularly in predicting temperature variations and wind speeds, which closely mirrored the actual recorded data. The performance on precipitation forecasts, while reasonably good, highlighted potential areas for improvement in handling the variability and timing of rainfall events.

The visual analysis through Predicted vs. Actual plots for both training and validation sets provided a clear and intuitive means to assess model performance. These plots revealed that while the RNN can capture the general trends and many nuances of the weather data, there are opportunities to refine the model to enhance its accuracy and reliability, especially for more volatile features like precipitation.

Future work should focus on exploring more sophisticated neural network architectures such as Long Short-Term Memory (LSTM) or Gated Recurrent Units (GRU) which might better capture the dependencies in weather data over longer time periods. Additionally, incorporating more diverse datasets and external factors that influence weather could further improve the predictive capabilities of the model. The ongoing refinement and testing of the model will contribute to its potential application in practical settings, such as dynamic weather prediction systems that could benefit sectors like agriculture, transportation, and event planning.

Overall, the project underscores the efficacy of machine learning techniques in meteorological applications and sets the stage for further innovations in environmental data analysis and forecasting.

Reinforcement Learning for Tic -Tac -Toe



Introduction

Tic-Tac-Toe is a turn-based board game where two players (“X” and “O”) take turns marking spaces in a 3x3 grid. The player who succeeds in placing three of their marks (“x” or “o”) in a horizontal, vertical, or diagonal row wins the game.

In this project, our objective is to design an optimal strategy for a computer to play Tic-Tac-Toe using reinforcement learning. We aimed to develop an AI agent capable of learning the optimal policy through interactions with the game environment.

Game Description

State:

- The state of the game is represented by the current configuration of the 3x3 grid, where each cell can be empty, contain an 'X', or contain an 'O'.

Actions:

- Each action corresponds to placing either an 'X' or an 'O' in an empty space on the grid and the number of actions will vary depending on the number of empty spaces available during each turn.

Rewards/Penalties:

- The agent receives a reward of +1 for winning, -1 for losing, and 0 for a draw.

Hypothesis for Action Rule and Reward/State Distribution

Action Rule

We hypothesized that a Deep Q-Network (DQN) architecture would be suitable for learning the optimal action rule. The network consists of multiple layers of densely connected neurons, with ReLU activation functions to introduce non-linearity. The final layer uses a linear activation function for Q-value approximation.

Reward Distribution

The reward distribution follows a binary scheme, where the reward is +1 for a win, -1 for a loss, and 0 for a draw. This distribution simplifies the learning process and provides clear feedback to the agent.

State Distribution

States are represented as one-hot encoded vectors, capturing the current configuration of the grid. The distribution of states is multinomial, with each state having a probability proportional to its occurrence during gameplay.

Mathematical Notation:

- **State:** S represents the set of all possible Tic-Tac-Toe board configurations.
- **Action:** A denotes the set of all possible actions (empty cells where the AI can place its mark).
- **Reward:** R consists of rewards (+1 for win, -1 for lose, 0 for draw).
- **Transition Probability:** P(Rt+1 = r, St+1 = s' | St = s; At = a) represents the probability of receiving reward R and transitioning to state S' after taking action A in state S.

Goal:

- Find the optimal action rule π(a|s), which represents the probability of choosing action a in state s, to maximize the expected future reward.

Performance Measure:
The agent receives a reward of:

- +1: Wins the game.
- -1: Loses the game.
- 0: Draw.

Discount Factor (Y)
The discount factor (between 0 and 1) determines how much the agent values future rewards compared to immediate ones. Higher γ prioritizes long-term strategies.

Q-Value Function (Q(s, a)):
This function estimates the expected future discounted reward of taking action a in state s, following a specific policy (π). It's updated using the Bellman equation.

Bellman Equation:
$Q(s, a) = R(s, a) + \gamma * \Sigma P(s', r' | s, a) * Q(s', \pi(s'))$

- R(s, a): Reward received for taking action a in state s.
- γ: Discount factor.
- P(s', r' | s, a): Probability of transitioning to state s' and receiving reward r' after taking action a in state s.
- Q(s', π(s')): Q-value of the next state s' following the current policy π(s').

Policy (π(s))
This function defines the probability of taking action a in state s based on the current Q-values.

Algorithm/ Procedure

1. Initialize Q-values: Set Q(s, a) to 0 for all states s and actions a.

2. Episodes: Repeat for a number of episodes:
    - Initial State: Start with an initial state s0.

- Action Selection: Choose an action a from state s0 based on the policy π(s).
- Exploration vs. Exploitation: Use an ε-greedy strategy where with a small probability ε choose a random action, otherwise choose the action with the highest Q-value in s0.
- Take Action: Take action a in state s0 and observe the reward R1 and the next state S1.
- Update Q-value: Update the Q-value for the previous state-action pair (s0, a) using the Bellman equation:
- Q(s0, a) = Q(s0, a) + α * [R1 + γ * max(Q(S1, a')) for all a' in A(S1)]
- α: Learning rate (between 0 and 1) determines how much the new information updates the existing Q-value.
- max(Q(S1, a')): Maximum Q-value of all possible actions in the next state S1.

3. Policy Update (Optional): After a number of episodes, consider updating the policy based on the learned Q-values to prioritize actions with higher Q-values.