Experiment No.: 7

Title: Recurrent Neural Network

Batch: HO–DL 1 **Roll No.: 16010422234** **Experiment No. 07**

Aim: To implement Recurrent Neural Network

Resources needed: Python/Matlab

Theory:

Recurrent neural networks (RNNs) are a class of neural networks that are naturally suited to processing time-series data and other sequential data. Recurrent neural networks are an extension to feedforward networks, in order to allow the processing of variable-length (or even infinite-length) sequences, and some of the most popular recurrent architectures in use, including long short-term memory (LSTM) and gated recurrent units (GRUs). RNN is a neural network designed for analyzing streams of data by means of hidden units. In some of the applications like text processing, speech recognition and DNA sequences, the output depends on the previous computations. Since RNNs deal with sequential data, they are well suited for the health informatics domain where enormous amounts of sequential data are available to process.

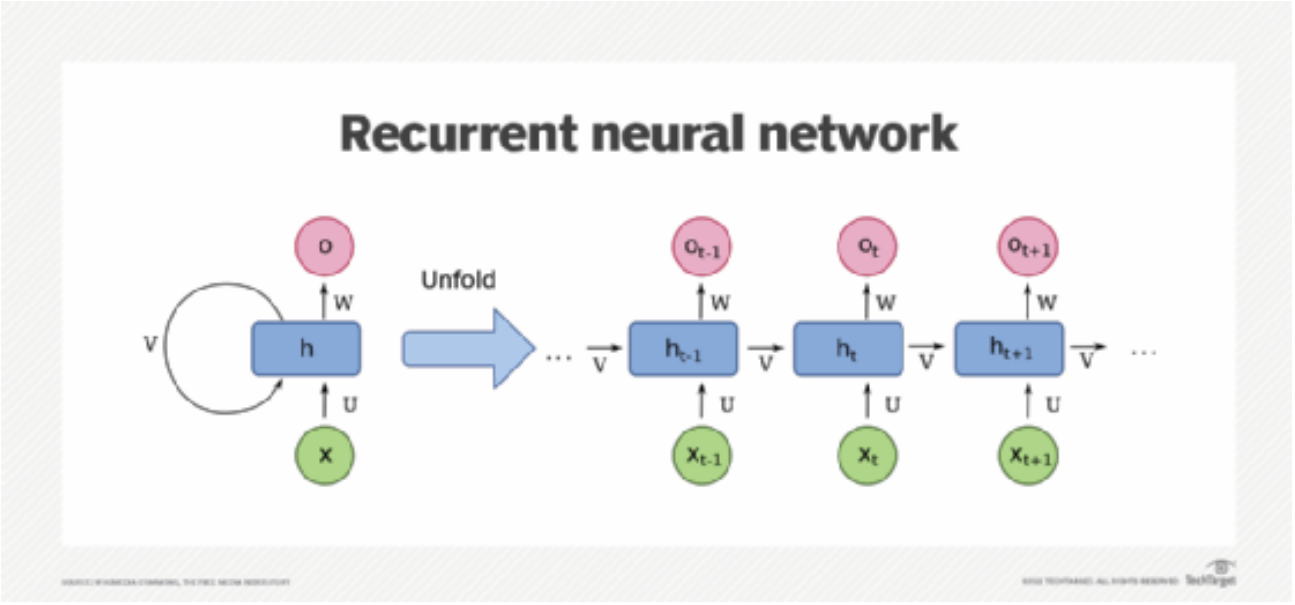


Figure 1: Courtesy of Wikimedia Commons – Depicting a one-unit RNN.

The bottom is the input state; middle, the hidden state; top, the output state. U, V, W are the weights of the network. Compressed version of the diagram on the left, unfold version on the right.

Looking at the visual below, the “rolled” visual of the RNN represents the whole neural network, or rather the entire predicted phrase, like “feeling under the weather.” The “unrolled” visual represents the individual layers, or time steps, of the neural network. Each layer maps to a single word in that phrase, such as “weather”. Prior inputs, such as “feeling” and “under”, would be represented as a hidden state in the third timestep to predict the output in the sequence, “the”.

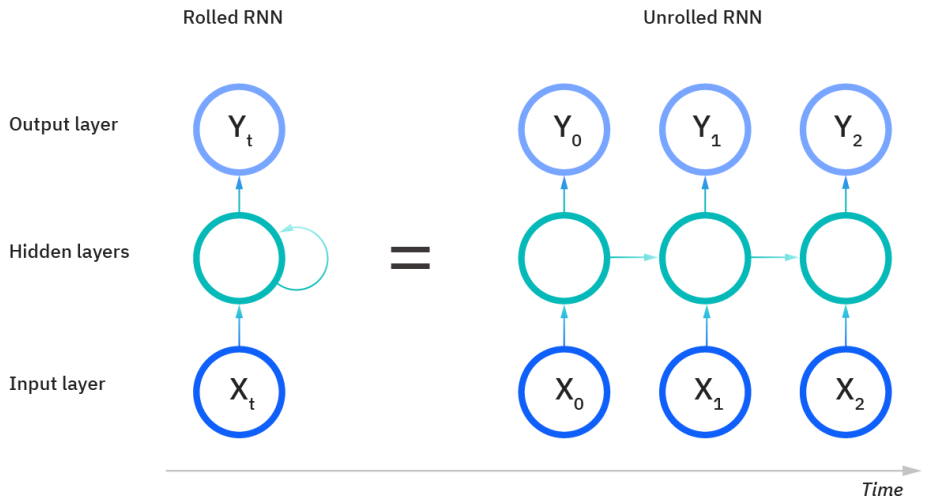


Figure 2: Rolled and Unrolled RNN

Another distinguishing characteristic of recurrent networks is that they share parameters across each layer of the network. While feedforward networks have different weights across each node, recurrent neural networks share the same weight parameter within each layer of the network. That said, these weights are still adjusted through the processes of backpropagation and gradient descent to facilitate reinforcement learning. Recurrent neural networks leverage backpropagation through time (BPTT) algorithm to determine the gradients, which is slightly different from traditional backpropagation as it is specific to sequence data. The principles of BPTT are the same as traditional backpropagation, where the model trains itself by calculating errors from its output layer to its input layer. These calculations allow us to adjust and fit the parameters of the model appropriately. BPTT differs from the traditional approach in that BPTT sums errors at each time step whereas

feedforward networks do not need to sum errors as they do not share parameters across each layer.

Training through RNN:

1. A single time step of the input is provided to the network.
2. Then calculate its current state using the set of current input and the previous state.
3. The current ht becomes ht-1 for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained.

Activity:

1. Import Requisite Libraries
2. Load any time series dataset.
3. Pre-process and visualize the dataset.
4. Form the Training and Testing Data.
5. Develop and train the model.
6. Plot the predictions for training and testing data.

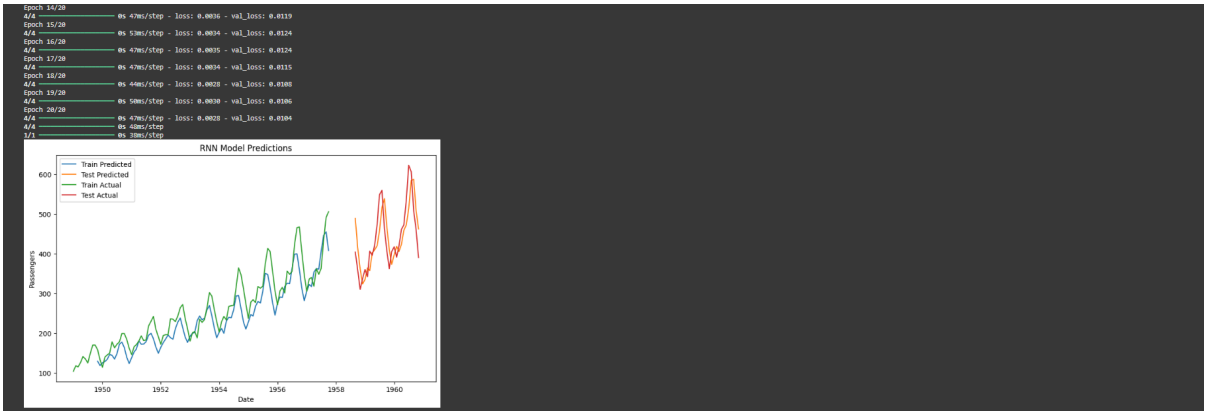
Program:

```
# 1. Import Requisite Libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.optimizers import Adam

# 2. Load any time series dataset (For this example, using a sample
time-series dataset)
# You can replace this with your own dataset (e.g., stock prices,
weather data, etc.)
url =
'https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-pa
ssengers.csv'
data = pd.read_csv(url, header=0, parse_dates=[0], index_col=0,
date_parser=pd.to_datetime)

# 3. Pre-process and visualize the dataset
data.plot()
plt.title('Airline Passengers Time Series')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.show()

# MinMax scaling to normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data.values.reshape(-1, 1))

# 4. Form the Training and Testing Data
# Create a function to prepare the data for RNN
def create_dataset(dataset, time_step=1):
    X, y = [], []
    for i in range(len(dataset)-time_step-1):
        X.append(dataset[i:(i+time_step), 0])
        y.append(dataset[i+time_step, 0])
    return np.array(X), np.array(y)

# Define the time step (number of previous time steps to consider)
time_step = 10
X, y = create_dataset(data_scaled, time_step)
```

```
# Reshape X to be in the format [samples, time steps, features]
X = X.reshape(X.shape[0], X.shape[1], 1)

# Split the data into training and testing sets (80% for training, 20%
for testing)
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# 5. Develop and train the model
model = Sequential()
model.add(SimpleRNN(units=50, activation='relu',
input_shape=(time_step, 1)))
model.add(Dense(1))  # Output layer with 1 unit
model.compile(optimizer=Adam(), loss='mean_squared_error')

# Train the model
history = model.fit(X_train, y_train, epochs=20, batch_size=32,
validation_data=(X_test, y_test), verbose=1)

# 6. Plot the predictions for training and testing data
# Predict the values for the training and testing data
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)

# Inverse transform the predictions back to original scale
train_predict = scaler.inverse_transform(train_predict)
y_train_actual = scaler.inverse_transform(y_train.reshape(-1, 1))
test_predict = scaler.inverse_transform(test_predict)
y_test_actual = scaler.inverse_transform(y_test.reshape(-1, 1))

# Plotting the predictions
plt.figure(figsize=(10, 6))

# Plot Train data (adjust the index to match the length of predictions)
plt.plot(data.index[time_step:train_size],
train_predict[:len(data.index[time_step:train_size])], label='Train
Predicted')

# Plot Test data (adjust the index and predictions to match the length)
plt.plot(data.index[train_size+time_step:train_size+time_step+len(test_
predict)], test_predict, label='Test Predicted')
```

```
# Plot Actual data for Train and Test (adjust the index range for Test Actual)
plt.plot(data.index[:train_size], y_train_actual, label='Train Actual')
plt.plot(data.index[train_size+time_step:train_size+time_step+len(y_test_actual)], y_test_actual, label='Test Actual')

plt.title('RNN Model Predictions')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.legend()
plt.show()
```

Output:



Post Lab Question — Answers (If Any):

1. **Differentiate between recurrent Neural Network and Feedforward Neural Network.**

   RNN:

   - Designed to handle sequential data.
   - Has a memory component (hidden state) that retains information from previous time steps.
   - Suitable for tasks such as time-series analysis, speech recognition, and natural language processing.
   - Parameters are shared across layers, and it processes data step-by-step over time.

   FNN:

   - Primarily used for static data where the inputs are independent of each other.
   - Lacks a memory component (hidden state) and does not retain information from previous inputs.
   - Suitable for tasks like image classification and regression.
   - Parameters are not shared across layers, and it processes the entire input at once.

2. **What are the problems associated with RNN?**

   - **Vanishing Gradient Problem:** During backpropagation, gradients tend to become very small as they are propagated backward through many time steps, making it difficult for the network to learn long-range dependencies.
   - **Exploding Gradient Problem:** Conversely, gradients can sometimes grow exponentially, causing the model's parameters to become unstable.
   - **Difficulty in Learning Long-Term Dependencies:** Standard RNNs struggle to remember information over long sequences, as they primarily focus on short-term dependencies.
   - **Slow Training:** Training RNNs can be computationally expensive and time-consuming, particularly when the network has many layers and time steps.
   - **Sensitive to Initial Conditions:** The network's performance can be highly sensitive to the initialization of weights, requiring careful tuning.

CO — 4: Underhand the essentials of Recurrent and Recursive Nets.

Conclusion:

Recurrent Neural Networks (RNNs) are a powerful class of neural networks specifically designed for sequential data processing. They are well-suited for applications like text processing, speech recognition, and time-series forecasting, where the output is dependent on previous inputs. Unlike feedforward networks, RNNs maintain a hidden state that captures information from earlier time steps, enabling them to process variable-length sequences. While training RNNs can be challenging due to issues like vanishing and exploding gradients, advancements like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) have addressed some of these problems. In this experiment, the RNN model was successfully implemented, trained on time-series data, and used to predict future values.

Grade: AA / AB / BB / BC / CC / CD / DD

Signature of faculty in-charge with date

References:

Books/ Journals/ Websites:

1. Josh Patterson and Adam Gibson, “Deep Learning A Practitioner’s Approach”, O’Reilly Media 2017.
2. https://www.ibm.com/cloud/learn/recurrent-neural-networks
3. https://searchenterpriseai.techtarget.com/definition/recurrent-neural-networks
4. https://www.sciencedirect.com/science/article/pii/B9780128161760000260
