Department of Information Engineering Technology

The Superior University, Lahore

Artificial Intelligence Lab

Experiment No.13

Implementation of RNN on Different Real-World Problems

Prepared for

_____

By:

Name: _____

ID: _____

Section: _____

Semester: _____

Total Marks: _____

Obtained Marks: _____

Signature: _____

Date: _____

Experiment No. 13

Implementation of RNN on Different Real-World Problems (Rubrics)

Name: _____ **Roll No. : _____**

A. PSYCHOMOTOR

Sr. No.	Criteria	Allocated Marks	Unacceptable 0%	Poor 25%	Fair 50%	Good 75%	Excellent 100%	Total Obtained
1	Follow Procedures	1	0	0.25	0.5	0.75	1	
2	Software and Simulations	2	0	0.5	1	1.5	2	
3	Accuracy in Output Results	3	0	0.75	1.5	2.25	3	
Sub Total		**6**	**Sub Total marks Obtained in Psychomotor(P)**					

B. AFFECTIVE

Sr. No.	Criteria	Allocated Marks	Unacceptable 0%	Poor 25%	Fair 50%	Good 75%	Excellent 100%	Total Obtained
1	Respond to Questions	1	0	0.25	0.5	0.75	1	
2	Lab Report	1	0	0.25	0.5	0.75	1	
3	Assigned Task	2	0	0.5	1	1.5	2	
Sub Total		**4**	**Sub Total marks Obtained in Affective (A)**					

Instructor Name:_____ **Total Marks (P+A): _____**

Instructor Signature:_____ **Obtained Marks (P+A): _____**

Introduction to Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) are a class of neural networks designed for sequence modeling and processing. They are particularly powerful in handling sequential data, making them well-suited for tasks in Natural Language Processing (NLP). In NLP, RNNs can be applied to tasks such as language modeling, machine translation, sentiment analysis, and more.

1. Sequence Modeling:

NLP often deals with sequential data, such as sentences, paragraphs, or entire documents. RNNs are well-suited for modeling sequential dependencies, as they maintain hidden states that capture information about past inputs in the sequence. This makes them effective for tasks where context and order matter.

2. Basic Architecture:

The fundamental building block of an RNN is the recurrent neuron, which maintains a hidden state. In each time step, the network takes an input, updates its hidden state, and produces an output. The hidden state serves as a memory that retains information from previous time steps.

3. Vanishing Gradient Problem:

While RNNs are powerful for capturing short-term dependencies, they struggle with long-term dependencies due to the vanishing gradient problem. Gradients diminish exponentially as they are backpropagated through time, making it challenging for the network to learn relationships across distant time steps.

4. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU):

To address the vanishing gradient problem, more advanced RNN architectures like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been introduced. These architectures include specialized mechanisms (gates) that regulate the flow of information, allowing the network to better capture long-term dependencies.

5. Applications in NLP:

RNNs find application in various NLP tasks, such as:

- **Language Modeling:** Predicting the next word in a sequence.
- **Machine Translation**: Translating text from one language to another.
- **Sentiment Analysis:** Analyzing and classifying the sentiment expressed in a piece of text.
- **Named Entity Recognition (NER):** Identifying and classifying entities (e.g., names, locations) in text.

6. Challenges and Advances:

While RNNs have been widely used, more recent architectures like Transformer models have gained popularity in NLP. Transformers overcome some limitations of RNNs and are particularly effective in capturing long-range dependencies.

In conclusion, RNNs play a crucial role in NLP by enabling the modeling of sequential dependencies. However, ongoing research and developments in neural network architectures continue to shape the landscape of NLP, with newer models providing improved performance on various tasks.

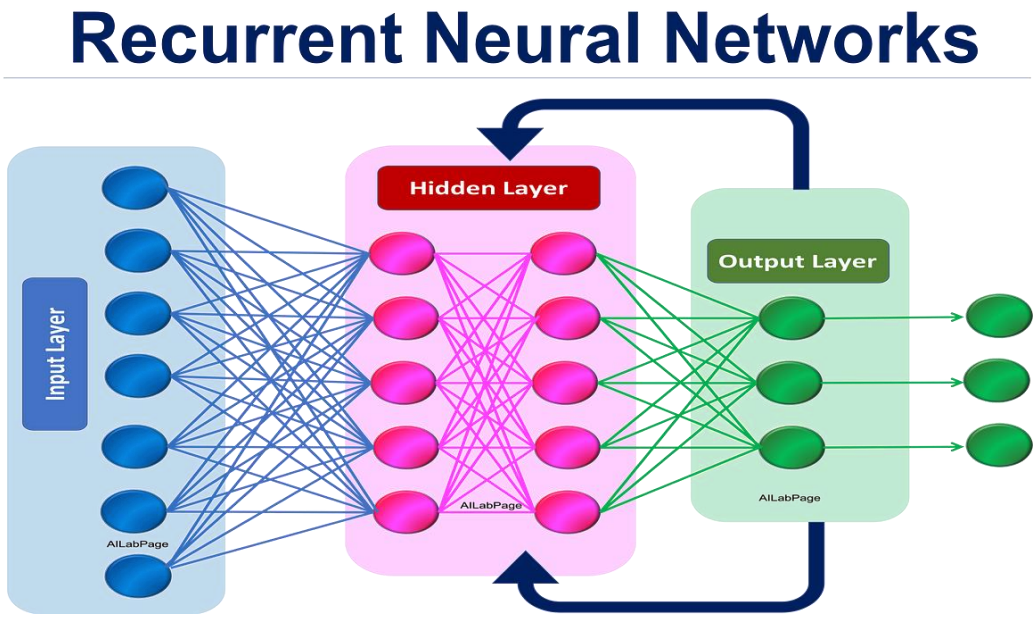

Fig 1: RNN Architecture

BUILDING A MODEL

Through Recurrent Neural Network (RNN)

Import the libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
import os
```

In this section, a CSV file named 'ABC.csv' is loaded into a Pandas DataFrame. The 'Date' column is converted to a datetime object, and additional columns for year, month, and day of the week are created. The 'Date' column is set as the index, and basic information about the dataframe is displayed.

```
file=pd.read_csv('ABC.csv')
# Load the data into a Pandas dataframe
file = pd.read_csv('ABC.csv')

# Convert the date strings to Pandas datetime objects
```

```
file['Date'] = pd.to_datetime(file['Date'], format='%d/%m/%Y')

# Extract the year, month, and day from the datetime objects
file['Year'] = file['Date'].dt.year
file['Month'] = file['Date'].dt.month
#df[' date_of_month'] = df['Dates'].dt.day_of_month
file['Day_of_week'] = file['Date'].dt.day_of_week

# Drop the original date column
#df = df.drop('Dates', axis=1)
file.set_index('Date')
file.describe()
file.info()
```

This section creates a line plot to visualize the average power consumption on different weekdays. The 'groupby' operation is used to calculate the mean power consumption for each day of the week, and the results are plotted using Matplotlib and Seaborn.

```
plt.figure(figsize=(18,6))
sns.set_style("darkgrid", {"axes.facecolor": "0.9", 'grid.color': '.6',
'grid.linestyle': '-.'})
dist1=file.groupby("Day_of_week")["Rescue"].mean()
dist1.plot(kind='line', linewidth=3, rot=0)
plt.xlabel("Day_of_week", fontsize=16, color="r")
plt.ylabel("Power in KW", fontsize=16, color="r")
plt.xticks(fontsize=14); plt.yticks(fontsize=14)
plt.title("Weekdays' Distribution", fontsize=18, color="r")
plt.legend(loc= 'best')
plt.show()
```

This part generates a line plot illustrating the total power consumption for each year. The data is grouped by year, and the sum of power consumption for each year is plotted.

```
plt.figure(figsize=(18,6))
year_sum=file.groupby("Year")[['Rescue']].sum()
plt.plot(year_sum, linewidth=3, color="r")
plt.xlabel("Year", fontsize=16, color="r")
plt.ylabel("Power in KW", fontsize=16, color="r")
plt.xticks(range(2002,2022,1), fontsize=14); plt.yticks(fontsize=14)
plt.title("Load Consumption 2002-2022", fontsize=18, color="r")
plt.show()
```

This code uses the groupby function to calculate the minimum (year_min), maximum (year_max), and average (year_mean) values of the 'Rescue' column for each year. The plot then visualizes these statistics over the years, with different colors for maximum, average, and minimum values. The shaded area between the minimum and maximum lines is filled with a yellow color, representing the range of load consumption. The x-axis represents the years, and the y-axis represents the load consumption values.

```
plt.figure(figsize=(18,6))
year_min=file.groupby("Year")[['Rescue']].min();
year_max=file.groupby("Year")[['Rescue']].max();
year_mean=file.groupby("Year")[['Rescue']].mean()
x=year_mean.index
plt.plot(year_max, linewidth=3, label="Maximum", color="b")
plt.plot(year_mean, linewidth=3, label="Average", color="g")
plt.plot(year_min, linewidth=3, label="Minimum", color="r")
plt.fill_between(x,year_min["Rescue"], year_max["Rescue"], alpha=0.3,
facecolor='yellow')
plt.xticks(range(2002,2022,1), fontsize=14); plt.yticks(fontsize=14)
plt.xlabel("Year", fontsize=16, color="r"); plt.ylabel("Range of Max,
Min & Avg Values", fontsize=16, color="r")
plt.title("Range of Load Consumption Data", fontsize=18, color="r")
plt.legend(loc='best')
plt.draw()
```

In this code, a new column named 'date' is created in the 'file' DataFrame by combining the 'Year' and 'Month' columns.

```
file['date'] = file["Year"].map(str)+ "-" + file["Month"].map(str)
file.head()
```

In this code, we create a new DataFrame ts by copying the data from the existing DataFrame file and selecting specific columns

```
ts=file.copy()
ts=pd.DataFrame(ts, columns=['date', 'Rescue'])
ts.tail()
```

In this code, we are converting the 'date' column to a datetime object, setting it as the index, and then dropping the original 'date' column.

```
ts['Date']=pd.to_datetime(ts['date'])
ts=ts.set_index(['Date'])
ts.drop("date", axis=1, inplace=True)
ts.head()
```

In this code, we are creating a line plot to visualize the daily and weekly trends of the 'Rescue' column in the DataFrame ts.

```
plt.figure(figsize=(18,6))
plt.plot(ts, label="Daily", color="steelblue")
plt.plot(ts.rolling(window=7).mean(), label="Weekly", color="darkred")
# plt.plot(ts.resample("w").mean(), label="weekly", color="darkred")
# print(ts.resample("w").mean())
plt.xticks(fontsize=14); plt.yticks(fontsize=14)
plt.title("Daily vs Weekly Trend", fontsize=18, color="r")
plt.xlabel("Years", fontsize=16, color="r");
plt.ylabel("Power in KW", fontsize=16, color="r")
plt.legend(loc="best", fontsize = 'x-large')
plt.draw()
```

We are printing out the shapes of two DataFrames, ts and month_mean.

```
print("The shape of the original data: ",ts.shape)
print("The shape of the sampled data: ", month_mean.shape)
```

In this code, we are splitting the month_mean DataFrame into training and testing sets.

```
train=month_mean[:200]
test=month_mean[200:]
print("The shape of training set: ", train.shape)
print("The shape of testing set: ", test.shape)
print("The splitting timestamp is after ", train.index[-1])
```

This code creates a plot to visualize the training and testing sets, with a vertical line indicating a specific date.

```
train=month_mean[:200]
test=month_mean[200:]
print("The shape of training set: ", train.shape)
print("The shape of testing set: ", test.shape)
print("The splitting timestamp is after ", train.index[-1])
```

In this code, we are scaling the training data using Min-Max scaling and preparing it for training a neural network, likely an LSTM (Long Short-Term Memory) network.

```
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0, 1))
train_scaled = sc.fit_transform(train)
X_train = []
y_train = []
for i in range(60, 200):
    X_train.append(train_scaled[i-60:i, 0])
    y_train.append(train_scaled[i, 0])
X_train, y_train = np.array(X_train), np.array(y_train)

X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
```

In this code, we are defining and training an LSTM (Long Short-Term Memory) neural network using Keras.

- **Sequential():** Initializes a sequential model.
- **LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)): Adds** the first LSTM layer with 50 units, return sequences set to True (because you have subsequent LSTM layers), and input shape specified.
- **Dropout(rate=0.2):** Adds a Dropout layer to help prevent overfitting by randomly setting a fraction of input units to 0 at each update during training.
- Subsequent LSTM and Dropout layers are added similarly to the model.
- **Dense(units=1):** Adds a Dense output layer with 1 unit for regression.
- **compile(optimizer='adam', loss='mean_squared_error')**: Compiles the model with the Adam optimizer and mean squared error loss, which is suitable for regression problems.
- **fit(X_train, y_train, epochs=1000, batch_size=10, verbose=1):** Trains the model on the training data (X_train, y_train) for 1000 epochs with a batch size of 10. The verbose=1 argument displays the training progress.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
regressor = Sequential()
```

```
regressor.add(LSTM(units = 50, return_sequences = True, input_shape =
(X_train.shape[1], 1)))
regressor.add(Dropout(rate=0.2))

regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(rate=0.2))

regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(rate=0.2))

regressor.add(LSTM(units = 50))
regressor.add(Dropout(rate=0.2))

regressor.add(Dense(units = 1))

regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')
np.random.seed(0)
regressor.fit(X_train, y_train, epochs = 1000, batch_size = 10,
verbose=1)
```

In this code, we are preparing the test data and making predictions using the trained LSTM model.

- **dataset_total = pd.concat((train['Rescue'], test['Rescue']), axis=0):** Concatenates the 'Rescue' values of the training and testing sets along the rows to create a combined dataset.
- **inputs = dataset_total[len(dataset_total) - len(test) - 60:** values: Selects the inputs for the test set, including the 60 time steps preceding the test set.
- **inputs = inputs.reshape(-1, 1):** Reshapes the inputs to be a 2D array with one column.
- **inputs = sc.transform(inputs):** Scales the inputs using the MinMaxScaler that was fitted on the training data.

The following code creates sequences of 60-time steps for the input features of the test set (X_test) and converts the list to a NumPy array.

- **X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1)):** Reshapes the input features array to be compatible with the expected input shape of the LSTM model.
- **prediction = regressor.predict(X_test):** Makes predictions using the trained LSTM model.
- **prediction = sc.inverse_transform(prediction):** Inverts the scaling to obtain the actual values.
- **prediction = pd.DataFrame(prediction, index=test.index):** Creates a DataFrame with predictions and uses the test index.

```
dataset_total = pd.concat((train['Rescue'], test['Rescue']), axis = 0)
inputs = dataset_total[len(dataset_total) - len(test) - 60:].values
inputs = inputs.reshape(-1,1)
inputs = sc.transform(inputs)
X_test = []
for i in range(60, 112):
    X_test.append(inputs[i-60:i, 0])
```

```
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
prediction = regressor.predict(X_test)
prediction = sc.inverse_transform(prediction)
prediction = pd.DataFrame(prediction, index=test.index)
```

In this code, we are creating a plot to visualize the training data, testing data, and the LSTM model's predictions

```
plt.figure(figsize=(18,6))
plt.plot(month_mean[:200], color = 'darkred', label = 'Train')
plt.plot(month_mean[200:], color = 'darkblue', label = 'Test')
plt.plot(prediction, color = 'darkgreen', label = 'Prediction')
plt.xticks(fontsize=14); plt.yticks(fontsize=14)
plt.title("LSTM PREDICTION", fontsize=18, color="r")
plt.xlabel("Years", fontsize=16, color="r"); plt.ylabel("Power in KW",
fontsize=16, color="r")
plt.legend()
plt.show()
```

In this code, we are evaluating the performance of the LSTM model by calculating and printing several metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the loss on the training set

```
from sklearn.metrics import mean_squared_error, mean_absolute_error

print("RMSE is ", np.round(np.sqrt(mean_squared_error(test,
prediction))))
print("MAE is ", np.round(mean_absolute_error(test, prediction)))
print("Loss  is ", np.round(regressor.evaluate(X_train ,y_train
,batch_size =10),4))
```

In this code, we are defining a function moving_test_window_preds to make predictions using a moving test window

- **preds_moving:** An empty list to store the predictions.
- **moving_test_window:** Initializes the moving test window with the first row of the test set.
- The function iterates through the desired number of feature predictions (n_feature_preds).
- **regressor.predict(moving_test_window):** Makes a one-step prediction using the trained LSTM model.
- Appends the predicted value to the preds_moving list.
- Reshapes the prediction for concatenation.
- Updates the moving test window by removing the first element and adding the prediction.

- Converts the list of predictions to a NumPy array.
- Reshapes the array.
- Inverts the scaling to obtain the actual values.
- Returns the array of predictions.

```
#PREDICTION
def moving_test_window_preds(n_feature_preds):
    preds_moving=[]
    moving_test_window=[X_test[0,:].tolist()]
    moving_test_window=np.array(moving_test_window)

    for i in range(n_feature_preds):
        preds_one_step=regressor.predict(moving_test_window)
        preds_moving.append(preds_one_step[0,0])
        preds_one_step=preds_one_step.reshape(1,1,1)
        moving_test_window=np.concatenate((moving_test_window[:,1:,:],pr
eds_one_step), axis= 1)
    preds_moving=np.array(preds_moving)
    preds_moving.reshape(1,-1)
    preds_moving=sc.inverse_transform(preds_moving.reshape(-1,1))
    return preds_moving
```

In this code, we are using the moving_test_window_preds function to generate future predictions and then plotting the original time series along with the predicted future values.

```
plt.figure(figsize=(18,6))
original=ts.resample("M").mean()
plt.plot(original, color = 'darkred', label = 'Original')
future=pd.DataFrame(moving_test_window_preds(121),
index=pd.date_range(start="20221231", end="20321231", freq="M"))
plt.plot(future, color="r", linestyle="-.")
plt.title('LSTM PREDICTION: Future Values', fontsize=18, color="r")
plt.xticks(fontsize=14); plt.yticks(fontsize=14)
plt.xlabel("Years", fontsize=16, color="r"); plt.ylabel("POWER in KW",
fontsize=16, color="r")
plt.draw()
```

In this code, we are creating a subplot with two panels to compare the original values of the time series with the LSTM model's predictions of future values.

```
plt.figure(figsize=(18,6))
```

```
plt.subplot(1, 2, 1)
original=ts.resample("M").mean()
plt.plot(original, color = 'darkred', label = 'Original')
plt.title('Original Values', fontsize=18, color="r")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.xlabel("Years", fontsize=16, color="r")
plt.ylabel("Power in KW", fontsize=16, color="r")

plt.subplot(1, 2, 2)
future=pd.DataFrame(moving_test_window_preds(132),index=pd.date_range(st
art="20220101", end="20321231", freq="M"))
plt.plot(future, color="r", linestyle="-.")
plt.title('LSTM PREDICTION: Future Values', fontsize=18, color="r")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.xlabel("Years", fontsize=16, color="r")
plt.ylabel("Power in KW", fontsize=16, color="r")

plt.draw()
```

Task:

Task: Time Series Forecasting with RNN

Objective:

Build an RNN model for time series forecasting using a dataset (https://www.kaggle.com/datasets/shenba/time-series-datasets) with sequential data. The goal is to predict future values in the time series based on historical patterns.

Include these Basic Steps:

- Dataset Selection
- Data Preprocessing
- Train-Test Split
- RNN Model Architecture
- Model Training
- Validation
- Hyperparameter Tuning
- Performance Evaluation
- Visualization
- Future Predictions