Lab Activity 3 – Recurrent Neural Network
Apply RNN on a Custom NLP Case

Graded assignment (Continuous assessment)

You are required to submit your solution of this assignment on BOOSTCAMP, respecting the deadline given in class. Submit your own work. Cheating will not be tolerated and will be penalized.

You may work in groups of up to 4 members.

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a special class of neural networks designed to handle sequential data. Unlike traditional feedforward neural networks, RNNs can retain information from previous inputs through their internal state (memory). This makes them particularly effective for tasks where context and order matter, such as natural language processing, speech recognition, and time series forecasting.

Quick Recap on Time Series Analysis
A time series is a sequence of observations recorded at successive points in time (e.g., stock prices, temperature, or website traffic).
Analyzing time series data allows us to extract patterns, trends, and seasonality, which can be used for prediction and forecasting.

- Univariate time series: based on a single variable (e.g., daily sales).
- Multivariate time series: based on multiple variables (e.g., temperature, humidity, and pressure to predict weather).

RNNs, and especially their improved variants such as LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit), are widely applied in both univariate and multivariate time series forecasting.

Learning Objectives:

- Understand and apply **text preprocessing** for machine learning.
- Design and train a **Recurrent Neural Network (RNN / LSTM)** on NLP case such as Text classification.
- Experiment with different **hyperparameters** and sequential model architectures.
- Evaluate the RNN using **relevant metrics.**

Use Case 3.1: Predict Google Stock Prices Using RNN

This lab activity predicts Google's Stock Prices using **LSTM (Long Short-Term Memory) Networks**.

Imagine living in a world where you know the exact prices of the stocks you invested in, but ahead of time. When you know exactly when they're heading up. When you know exactly when they're going down. You'll be on your path to becoming a millionaire.

*Quick Recap on **univariate or multivariate time series forecasting**: When you consider a univariate time series analysis, this means that you forecast a future variable based on the previous values of that only one variable, such as yearly sales forecast which is the process of predicting your future sales revenue for a specific timeframe (yearly), based on the historical sales revenue values.*
While multivariate time series forecasting means you forecast the future values of a specific target variable depending on not only the previous values of that variable but also on other variables' values.

In this lab activity, we are going to implement an Recurrent Neural Network model of type LSTM Long Short-Term Memory model to predict Open Stock Price of 2017 based on a data structure with 60 timesteps, which means forecasting Open Stock Price at time t based on previous daily 60 values at time [t-60, t-1]. It consists of univariate time series forecasting: one target variable forecast depending on historical previous values of the same variable which is the Open Stock Price.

This same case study can be implemented using multivariate time series forecasting. Please refer to the additional support materials that are provided to you to understand the implementation of this kind of model.

Two datasets are given: Google_Stock_Price_Train.csv and Google_Stock_Price_Test.csv

The solution of implementing the LSTM network to predict Google stock price is shared with you for pedagogical purpose.

Use Case 3.2: Fake News Detection Using RNN (Guided, not graded)

This activity aims to classify fake news from real news using a recurrent neural network. To simplify the text preprocessing procedure, the built in functions from TensorFlow will be used instead of the more established libraries like NLTK.

It is a binary text classification.

The dataset is separated into two files:
Fake.csv (23502 fake news article)
True.csv (21417 true news article)

The dataset columns are:

- Title: title of news article
- Text: body text of news article
- Subject: subject of news article
- Date: publish date of news article

Credits to Xingyu Bian / Git user: therealcyberlord
Source :
https://github.com/therealcyberlord/Fake-News-Detection-Using-RNN?tab=readme-ov-file
https://www.kaggle.com/code/therealcyberlord/fake-news-detection-using-rnn/notebook

Steps followed to implement the RNN classifier:

1. Data Loading
    - Loaded two CSV files: one with fake news (Fake.csv) and one with true news (True.csv).
2. Data Cleaning & Preprocessing
    - Checked for missing values.
    - Dropped irrelevant columns (date, subject) to avoid bias.
    - Combined fake and real news into a single dataset with labels.
3. Text Preprocessing
    - Removed unwanted characters and symbols using regex.
    - Converted text to lowercase.
    - Tokenized and padded sequences with TensorFlow's Tokenizer and pad_sequences.
4. Dataset Splitting
    - Split the data into training and testing sets using train_test_split.
5. Model Design (RNN)
    - Used an Embedding layer to convert words into dense vectors.
    - Added a SimpleRNN/LSTM layer to capture sequential dependencies.
    - Added Dense layers for classification.
6. Model Training
    - Compiled the model with adam optimizer and binary_crossentropy loss.
    - Trained on the training set, validated on the testing set.
7. Evaluation
    - Measured accuracy, precision, recall.
    - Built a confusion matrix (expressed in percentages).
8. Results
    - Model achieved near-perfect classification of fake vs. real news.
      Strong performance:
        - Accuracy: ~99%
        - Precision: ~98.8%
        - Recall: ~99.2%

Use Case 3.3: Apply RNN on a Custom NLP Case (Graded part)

The main objective of this activity is to reinforce your understanding of sequence models and text preprocessing.

Instructions:

1. Select a text dataset (e.g., sentiment analysis, movie reviews, topic classification).
2. Preprocess the data (tokenization, padding, embedding).
3. Design and implement an RNN model to solve the task.
4. Evaluate performance and discuss results.
5. Experiment with LSTM variants or hyperparameter tuning: trying different combinations of hyperparameters to improve performance (accuracy, loss, F1-score, etc.).

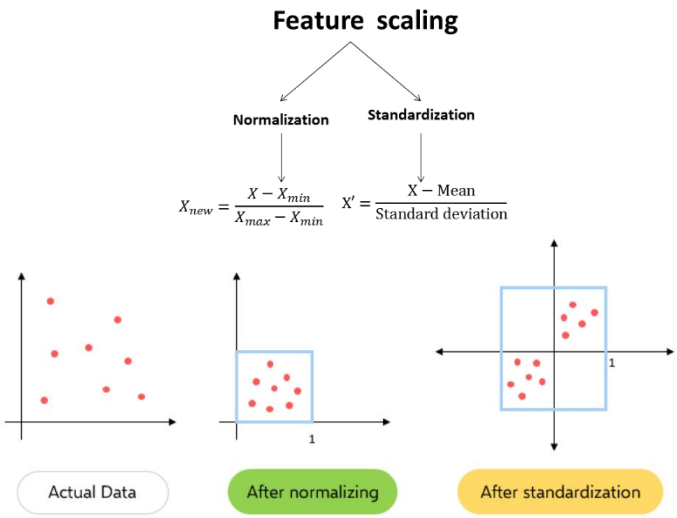
Hyperparameter	**What it does**
Number of units (hidden size)	How many neurons in each LSTM layer
Number of layers	How deep the network is (single vs stacked LSTM)
Learning rate	How fast the model updates weights during training
Batch size	Number of samples per training step
Dropout rate	Fraction of neurons to randomly ignore during training
Sequence length / input window	How many time steps the LSTM sees at once

Do not forget to add internal comments for interpretation and explanation in each step.

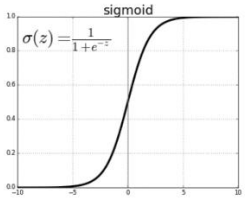
GOOD LUCK!

To help you, here are some recommendations for implementing an RNN/LSTM. Note: these are not absolute rules, but general guidelines that can be applied in most cases.

Scale your data with the appropriate scaling function:



It is recommended to use normalization when implementing RNN (because Sigmoid is a main activation function used in RNN).



<u>Attention: When using RNN / LSTM for NLP tasks, you don't typically scale text data the way you scale numeric features in regression or time series forecasting. Instead, you apply embedding representations.</u>

Scaling is important when your RNN/LSTM works on **numeric time series data** (e.g., stock prices, weather data, IoT sensor signals).

- In that case, you typically use **Min-Max Scaling** or **Standardization** so values are within a reasonable range (e.g., [0,1] or mean=0, std=1).
- This improves convergence because activation functions like **sigmoid** or **tanh** are sensitive to large input values.

Example (time series preprocessing):

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0,1))
scaled_data = scaler.fit_transform(time_series_data.reshape(-1,1))
```

For NLP (text data):

- The raw text is **tokenized** (converted into integer sequences).
- These integers are then passed into an **Embedding layer** (learned embeddings, or pretrained like GloVe/Word2Vec/BERT).
- Since embeddings already produce vectors with consistent ranges (often small values like -1 to 1 or near 0), **no extra scaling is required**.

Example:

```
model = tf.keras.Sequential([
  tf.keras.layers.Embedding(input_dim=10000, output_dim=128, input_length=200),
  tf.keras.layers.LSTM(64),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Here, the **Embedding layer** automatically maps words into dense vectors suitable for the LSTM.

<u>Building and Training the LSTM / RNN includes the following steps:</u>

1. Start building your LSTM recurrent neural network
2. Import necessary libraries
3. First, you will create and initialize your NN model as a sequential model.
4. Next, you will create the layers…
5. Add the first LSTM layer that will have a fixed number of units. Set reurn_sequences to True, in order to return the full sequence in the output. In more technical terminology,

the return_sequence will also output the hidden sequence as well. By default, the return_sequences is set to False in Keras RNN layers.

6. Add a Dropout layer which is used to prevent the model from overfitting, which essentially means that the model is limited only to the certain dataset, which makes the model not very flexible for other random inputs. To prevent this, the dropout layer (implemented for each LSTM layer), ignores units (neurons) during training randomly. It essentially randomly sets the outgoing edges of the hidden units to 0 in each layer. However, they are not completely ignored. Each value dropped specifies the probability at which outputs of the layer are dropped out at.
7. Add another second, third and fourth LSTM and Dropout layers, etc.
8. The last layer to add is the Dense layer. By adding this dense layer, you are simply stating that the neurons from that output layer are fully connected to the neurons in the previous layer.
9. Compile the model. To do so, use adam, which is the best optimizer for training deep learning models.
10. Use the loss function calculating the mean squared error comparing the observed values, to the predicted values.
11. Fit the model on your data. Iterate through multiple epochs and set a convenient batch size.
12. Prepare you test set.
13. Test the model with predictions.
14. Evaluate the performance of the model.
15. Visualize the data to compare the predicted values to the actual values using a graph.