Project Title	**Apple Stock Price Prediction**
Skill Set	**Basic Python, Data Visualization, Data Cleaning, EDA, Deep Learning (SimpleRNN and LSTM)**
Domain	**Financial Services**

Project Objective

To create a predictive Deep Learning (DL) model to predict the stock price of Apple that can predict 1 day, 5 days and 10 days behavior of stock's adjusted closing price. And analyze whether the model effectively captures stock price trends.

Approach for Apple Stock Price Prediction

The goal is to predict Apple stock prices using historical data. A deep learning-based approach has been used to model stock price trends.
The dataset consists of features: Date, Open, High, Low, close, Adj adj close, and Volume.

The Deep Learning Models used are Recurrent Neural Network (RNN):

- Simple Recurrent Neural Network (SimpleRNN)
- Long Short Term Memory(LSTM)

RNN

A Recurrent Neural Network (RNN) is a type of neural network designed for sequential data. Unlike traditional neural networks that assume inputs are independent, RNNs remember past information to make decisions about the current input.

Uses of RNN:

RNNs are ideal for tasks where **order and context matter**, such as:

- Text (sentences, paragraphs)
- Time series (stock prices, weather)
- Audio and speech
- Video frames

Variants of RNN

1. **Simple RNN or Vanilla RNN** — simple but limited (short-term memory)
2. **LSTM (Long Short-Term Memory)** — solves long-term dependency issues
3. **GRU (Gated Recurrent Unit)** — similar to LSTM but simpler

SimpleRNN

A Simple RNN is also called as Vanilla RNN, and it is the simplest form of RNN. It processes sequences one step at a time**,** and each output depends on the current input and the previous hidden state**.**

It's called "vanilla" because it lacks the gates found in more advanced RNNs like LSTM and GRU**.**

Summary of the steps in a SimpleRNN:

1. A single time step of the input is supplied to the network i.e. xt is supplied to the network
2. The calculate its current state using a combination of the current input and the previous hidden state(ht0) i.e. we calculate ht1
3. The current ht1 becomes ht-1 for the next time step
4. Once all the time steps are completed the final current state is used to calculate the output yt
5. The output is then compared to the actual output and the error is generated
6. The error is then backpropagated to the network to update the weights and the network is trained

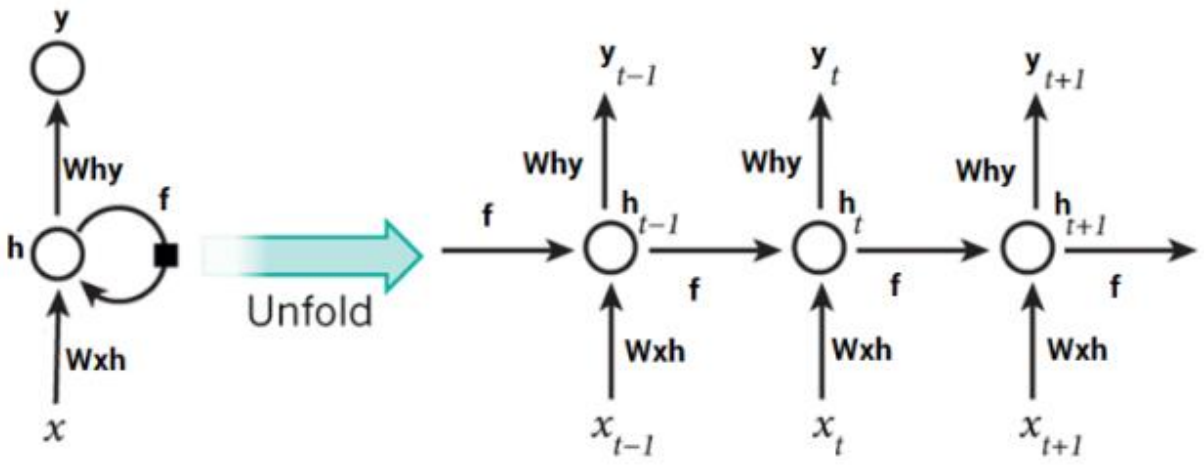
At each time step t, a Vanilla RNN does:

$$\mathbf{h}_t = \tanh(W_{xh} \cdot \mathbf{x}_t + W_{hh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{y}_t = W_{hy} \cdot \mathbf{h}_t + \mathbf{b}_y$$

Where:

- $\mathbf{x}_t$: input at time `t`
- $\mathbf{h}_t$: hidden state at time `t` (dense vector)
- $\mathbf{y}_t$: output at time `t`
- W_{xh}: weights from input to hidden
- W_{hh}: weights from previous hidden to current hidden
- W_{hy}: weights from hidden to output
- b_h, b_y: biases
- `tanh`: activation function (can also be `ReLU`, `si↓id`, etc.)



In case of a forward propagation, the inputs enter and move forward at each time step. In case of a backward propagation in this case, we are figuratively going back in time to change the weights, hence we call it the Back propagation through time (BPTT).

Limitations of Vanilla RNN

- Suffers from **vanishing/exploding gradients:**
  Gradient gets very small as you go backward through time, and causes early time steps to learn almost nothing.
- **Exploding Gradients:** Gradient grows exponentially with time steps , and causes instability. Gradient clipping can be used to overcome Exploding Gradient.
- Can't learn **long-term dependencies** well
- Can't "remember" input far back in the sequence

LSTM – Long Short Term Memory

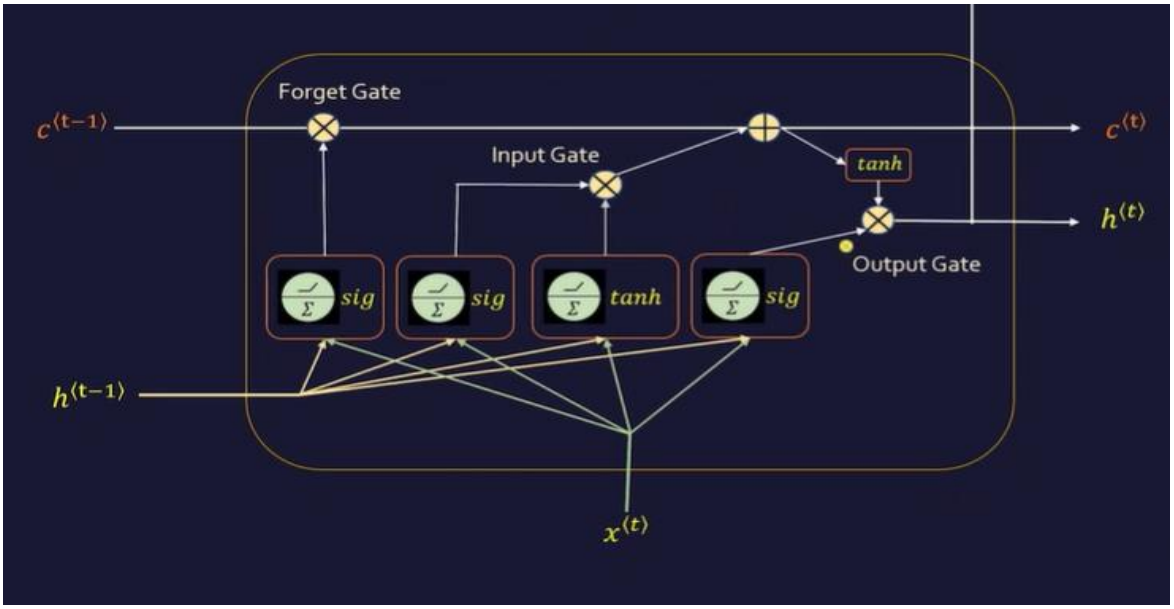
Long Short Term Memory network addresses short term memory problem in RNN by introducing long term memory cell (a.k.a cellstate)

It has both long term – cell state and short term – hidden state memory.

LSTM Architecture

An **LSTM cell** contains:

- **Cell state** Ct: long-term memory
- **Hidden state** ht: short-term output (like in RNN)
- **Three gates**:
  1. **Forget gate**: what to forget?
  2. **Input gate**: what to store?
  3. **Output gate**: what to output?



1. **Forget gate:**

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Decides how much of previous cell state to forget.

2. **Input gate:**

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Decides how much **new information** to add.

3. **Cell state update:**

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

Combines memory from past and present.

4. **Output gate:**

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \cdot \tanh(C_t)$$

Final output hidden state.

The LSTM:

- Learns patterns across time
- **Remembers long-range dependencies**
- Predicts the next value