.rolling(100)

This creates a rolling window of 100 rows. It means:

"Take every 100-day chunk as you move day by day." So for each day

t, it considers the closing prices from t−99 to 𝑡

MinMaxScaler is a data normalization technique from sklearn.preprocessing that scales features to a fixed range, usually [0, 1].So:

The minimum value becomes 0

The maximum value becomes 1

Everything else is proportionally scaled between them

model = Sequential()

model.add(LSTM(units=50, activation='relu', return\_sequences=True,

input\_shape=(x.shape[1], 1)))

model.Dropout(0.2)

Creates a **Sequential model**, meaning layers will be added **one after another** in order.

| **Argument** | **Meaning** |
| --- | --- |
| units=50 | Number of memory cells (neurons) in this LSTM layer |
| activation='relu' | Activation function applied to output of each cell. *(Normally tanh is default in LSTM, but ReLU is being used here)* |
| return\_sequences=True | Output the **entire sequence**, not just the final output — needed if you're adding another LSTM layer later |
| input\_shape=(x.shape[1], 1) | Tells the model: each sample has **x.shape[1] time steps** and **1 feature** (usually the price) |

| **Dropout** | **Purpose** |
| --- | --- |
| Dropout(0.2) | Randomly "drops" (i.e., disables) 20% of neurons during training to prevent **overfitting** |

**⚡ What is an Activation Function? — *In Simple Terms***

An **activation function** is a mathematical function used in neural networks to **decide whether a neuron should "fire" (activate) or not** — basically, it **introduces non-linearity** to the model.

Without it, your entire neural network would behave like just a big linear equation (which can't solve complex problems like stock prediction, image recognition, etc.).

**Dropout** is a **regularization technique** used in neural networks — including LSTM models — to prevent **overfitting**.

**Overfitting** happens when your model learns the **training data too well**, including its **noise and minor fluctuations**, but **fails to perform well on new (unseen) data**.

| **Feature** | **Benefit** |
| --- | --- |
| **Stacked LSTM layers** | Capture complex temporal patterns in stock data |
| **Increasing units (50 → 60 → 80 → 120)** | Learn progressively higher-level features |
| **Dropout (0.2 → 0.5)** | Prevent overfitting by randomly disabling neurons |

**Time steps** refer to the number of sequential data points (or observations) in a single sample — i.e., how far back in time the model can "look" at once. 100 in this case and with 1 feature whis is the closing price

**🔹 Dense layer:**

This adds a **fully connected (dense) layer** at the **end** of your LSTM model.

Each neuron in this layer is connected to **all outputs** of the previous layer.

**🔍 Explanation of units=1**

* units=1 means this layer has **1 neuron**.
* This neuron will output **1 value** → your **predicted stock price** for the next day.

model.compile(optimizer='adam', loss='mean\_squared\_error')

This **prepares the model for training** by specifying how it will **learn** and how it will **measure error**.

**🔍 optimizer='adam'**

* **Adam** = Adaptive Moment Estimation.
* It's a powerful and widely used optimization algorithm that:
  + Combines **momentum** and **RMSProp**
  + Automatically adjusts the **learning rate** during training
* Very effective for **time-series** tasks like stock price prediction.

📌 In short: Adam helps your model **quickly and efficiently find the best weights**.

**🔍 loss='mean\_squared\_error' (MSE)**

* MSE = Mean Squared Error = average of the squares of the prediction errors:
* 
* Ideal for **regression problems**, where you’re predicting **continuous values** like stock prices.

📌 In short: MSE tells the model **how wrong its predictions are**, and the model tries to minimize this error during training.

model.fit(x, y, epochs=50, batch\_size=32, verbose = 1)

* x: Input data (e.g., sequences of 100 past stock prices)
* y: Target labels (e.g., next day’s stock price)

**epochs=50**

* The model will go through the **entire training dataset 50 times**.
* More epochs = more learning, but too many can cause **overfitting**.

**batch\_size=32**

* The training data is split into **batches of 32 samples**.
* The model updates weights after each batch (not after the whole dataset).
* This helps make training faster and more memory-efficient.

**verbose=1**

* Controls the training output in your notebook/console.
* verbose=1 = prints **progress bar and metrics** after each epoch.

past\_100\_days = data\_train.tail(100)

data\_test = pd.concat([past\_100\_days, data\_test], ignore\_index=True)

Your model was trained to **predict the next day's price** using the **previous 100 days** of data.

**🧾 Let’s say:**

* You split your dataset:
  + data\_train → first 80% of rows (say rows 0 to 799)
  + data\_test → last 20% (say rows 800 to 999)

Now, when predicting the **first value in data\_test**, the model needs:

The **100 days before that** → i.e., rows 700 to 799

But these 100 days are **not in data\_test** — they are in data\_train.

**Streamlit:**

**Streamlit** is an open-source Python library used to **quickly build and share web applications** for data science and machine learning projects — all with just **pure Python code**.

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature\_range=(0, 1))

This creates a scaler object that will scale data to the range **[0, 1]**.

