**What is LSTM?**

**LSTM** is a type of Recurrent Neural Network (RNN) that is capable of **learning long-term dependencies**. It was specifically designed to solve the **vanishing gradient problem** that affects traditional RNNs when trying to learn from long sequences.

**Why Not Plain RNN?**

* RNNs have a feedback loop that allows them to "remember" past information.
* However, **RNNs struggle to learn long-term dependencies** due to the **vanishing/exploding gradient** problem during backpropagation through time (BPTT).
* They forget old information quickly, especially over long sequences.

**How Does LSTM Fix That?**

LSTM introduces a **cell state** and a set of **gates** that control the flow of information, allowing it to **remember or forget** information over long periods.

**LSTM Architecture -** An LSTM cell contains:

**1. Cell State (Ct)**

* Acts like a **conveyor belt** carrying long-term memory across the sequence.
* Data can be added or removed via gates.

**2. Hidden State (ht)**

* Carries short-term memory or output at each time step.

**3. Gates in LSTM**

These gates are **neural network layers with sigmoid activations**. They control what information should be **kept**, **added**, or **removed**.

**Forget Gate (ft)**

* **Purpose**: Decides what information to **forget** from the cell state.
* **Equation**:  
  ft = sigmoid(Wf · [ht-1, xt] + bf)
* Output is 0 (forget) or 1 (keep), element-wise.

**Input Gate (it + Ct~)**

* **Purpose**: Decides what new information to **store** in the cell.
* **Equations**:  
  it = sigmoid(Wi · [ht-1, xt] + bi)  
  Ct~ = tanh(Wc · [ht-1, xt] + bc)
* These are combined to update the memory:  
  Ct = ft \* Ct-1 + it \* Ct~

**Output Gate (ot)**

* **Purpose**: Decides what to **output**.
* **Equations**:  
  ot = sigmoid(Wo · [ht-1, xt] + bo)  
  ht = ot \* tanh(Ct)

**LSTM Flow at Time Step t**

Given input xt, previous hidden state ht-1, and previous cell state Ct-1:

1. **Forget** old info from Ct-1 → ft
2. **Calculate** new info to be added → it and Ct~
3. **Update** cell state → Ct = ft \* Ct-1 + it \* Ct~
4. **Output** → hidden state ht = ot \* tanh(Ct)

**Summary Table**

| **Component** | **Function** | **Activation** |
| --- | --- | --- |
| Forget Gate | What to remove from memory | Sigmoid |
| Input Gate | What to add to memory | Sigmoid |
| Candidate | Proposed new memory content | Tanh |
| Output Gate | What to output | Sigmoid |
| Final Output | Controlled output from current memory | Tanh |

**Why Use LSTM:** Text sentiment analysis, Speech recognition, Time series forecasting, Machine translation, Video analysis

**Problem: Predict the next word in a sentence**

A classic sequence modeling task with Input: "I like to eat ice" and Expected Output: "cream"

Why Use LSTM?

This task requires understanding *context* over several words:

* “ice” → makes most sense with “cream”
* LSTM remembers long-term context and handles the vanishing gradient problem better than plain RNNs.

ALGORITHM: LSTM Step-by-Step. We’ll break this into parts:

1. Prepare the Data: Let’s use a very simple toy corpus:

corpus = ["I like to eat ice cream", "I like to eat pizza", "I like to play football", "Do you want some ice cream", "We love to eat ice cream too"]

2. Tokenize & Convert to Integer Sequences

Step 1: Tokenization and vocab\_size

tokenizer = Tokenizer()

tokenizer.fit\_on\_texts(corpus)

sequences = tokenizer.texts\_to\_sequences(corpus)

vocab\_size = len(tokenizer.word\_index) + 1

Step2: Split into Input and Output (Supervised Learning): From the sentence: "I like to eat ice cream”, We extract: X = ["I"], ["I like"], ["I like to"], ... and y = ["like"], ["to"], ["eat"], ..., Using sliding windows of sequences.

2. LSTM Model (Concept)

An LSTM cell has:Three Gates:

1. Forget gate – what to throw away from memory
2. Input gate – what new information to store
3. Output gate – what to send out as the hidden state

At each word (time step), LSTM updates its cell state (Ct) and hidden state (ht) using these gates. For our task: Input: Word embeddings (one per word) and Output: Probabilities for next word

3. Architecture

model = Sequential()

model.add(Embedding(vocab\_size, 10, input\_length=max\_sequence\_len-1)) # Turn word indices into dense vectors

model.add(LSTM(100)) # Core LSTM layer

model.add(Dense(vocab\_size, activation='softmax')) # Predict one word from vocab

4. Training

Train the model to predict the next word, given a sequence

5. Prediction

After training: For Input = "I like to eat", to predict the next word, the LSTM processes the entire sentence (with its memory), then outputs a probability distribution over the vocabulary. It may output: {'pizza': 0.01, 'ice': 0.03, 'cream': 0.93, ...}. Basedon it, the Predicted next word: "cream"

Summary: LSTM Workflow

1. Tokenize and prepare input-output pairs
2. Use embedding to vectorize input
3. Feed sequence into LSTM (which remembers context)
4. Predict next word using softmax over vocabulary
5. Repeat this for sequence generation if needed

In summary,

* LSTM keeps memory (cell state) across steps
* Learns patterns like "eat ice → cream" through backpropagation over sequences
* Unlike RNN, LSTM doesn’t forget distant context easily

**GRU: Gated Recurrent Unit**

Recurrent Neural Networks (RNNs) struggle with **vanishing gradients** when learning long-term dependencies. LSTMs addressed this with a complex memory mechanism. **GRU** was introduced as a **simpler and computationally cheaper alternative** to LSTM — with **comparable performance**.

Introduced by **Cho et al. in 2014**, GRUs aim to retain the strengths of LSTMs while reducing complexity.

**GRU Architecture (Internals):** Instead of 3 gates and a separate memory cell (like LSTM), GRU combines everything using just **two gates**:

| **Component** | **Function** |
| --- | --- |
| **Update Gate zₜ** | Decides how much past information to keep |
| **Reset Gate rₜ** | Decides how much past information to forget |

No separate memory cell Cₜ in GRU — the hidden state hₜ **serves as both** the output and memory.

**GRU Equations**

Let’s say: xₜ: input at time step t, hₜ₋₁: hidden state at previous time, hₜ: current hidden state (also the output), W, U, b: learnable weights and biases

**1. Update gate zₜ** Controls the degree to which the unit updates its **activation (memory)**

**2. Reset gate rₜ** Controls how much of the **previous state to forget**

**3. Candidate activation ˜ht** Generates the **new candidate memory**, influenced by rₜ:

**4. Final hidden state hₜ:** Weighted mix of previous state and new candidate​

**What Do These Equations Do in Intuition:**

**Reset Gate (rₜ)**: When close to 0, it forces the model to ignore the past (forget).

**Update Gate (zₜ)**: When close to 1, the model updates to new content; when 0, it retains past memory.

Together, they give the GRU the ability to **adaptively remember and forget**, just like LSTM, but with fewer components.

**GRU vs LSTM — Comparison**

| **Feature** | **GRU** | **LSTM** |
| --- | --- | --- |
| Gates | 2 (Update, Reset) | 3 (Input, Forget, Output) |
| Memory cell (Cₜ) | No separate cell | Yes |
| Parameters | Fewer | More |
| Computation | Faster | Slower |
| Tuning | Simpler | Complex |
| Performance | Comparable in many tasks | Slightly better in very long sequences |
| Output | hₜ = memory & output | hₜ is output; Cₜ is internal memory |

**Real-world Analogy**

Imagine writing a report:

**LSTM** is like using **a notebook (memory)** and **a clipboard (output)** — three knobs (gates) control how you: Write, Erase and Read out

**GRU** is like just using a clipboard — one knob decides **what to keep**, the other **what to erase** — and that’s it!

**When to Use What?**

| **Scenario** | **Use** |
| --- | --- |
| Less training data or low compute | **GRU** — it's simpler and faster |
| Tasks needing precise long-term memory (e.g. long sequences like translation, speech) | **LSTM** often better |
| Empirical testing says both work | Start with GRU, then try LSTM |

**Summary**

GRU is a **simpler cousin** of LSTM, Uses **2 gates** instead of 3. There is **No separate cell state**; hidden state does both jobs. It is Faster to train with fewer parameters and Performs comparably to LSTM on many tasks