LSTM (Long Short Term Memory)

🔷 Summary (Quick Orientation)

- LSTM is a special kind of **Recurrent Neural Network (RNN)**.
- It's designed to **remember long-term dependencies** and **solve the vanishing gradient problem**.
- It does this using **gates** that control what to **keep**, **forget**, and **output**.

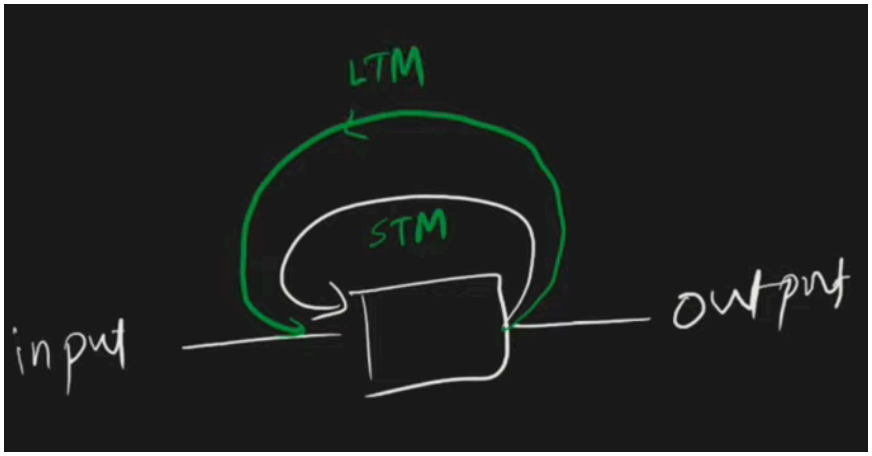
> **Think of it as a memory cell with smart gates that decide:**
>
> - **What information to keep in memory**
> - **What to throw away**
> - **What to send to the output**

Invented by Hochreiter & Schmidhuber in 1997.

Here, you maintain 2 Paths:

1. Short term
2. Long term

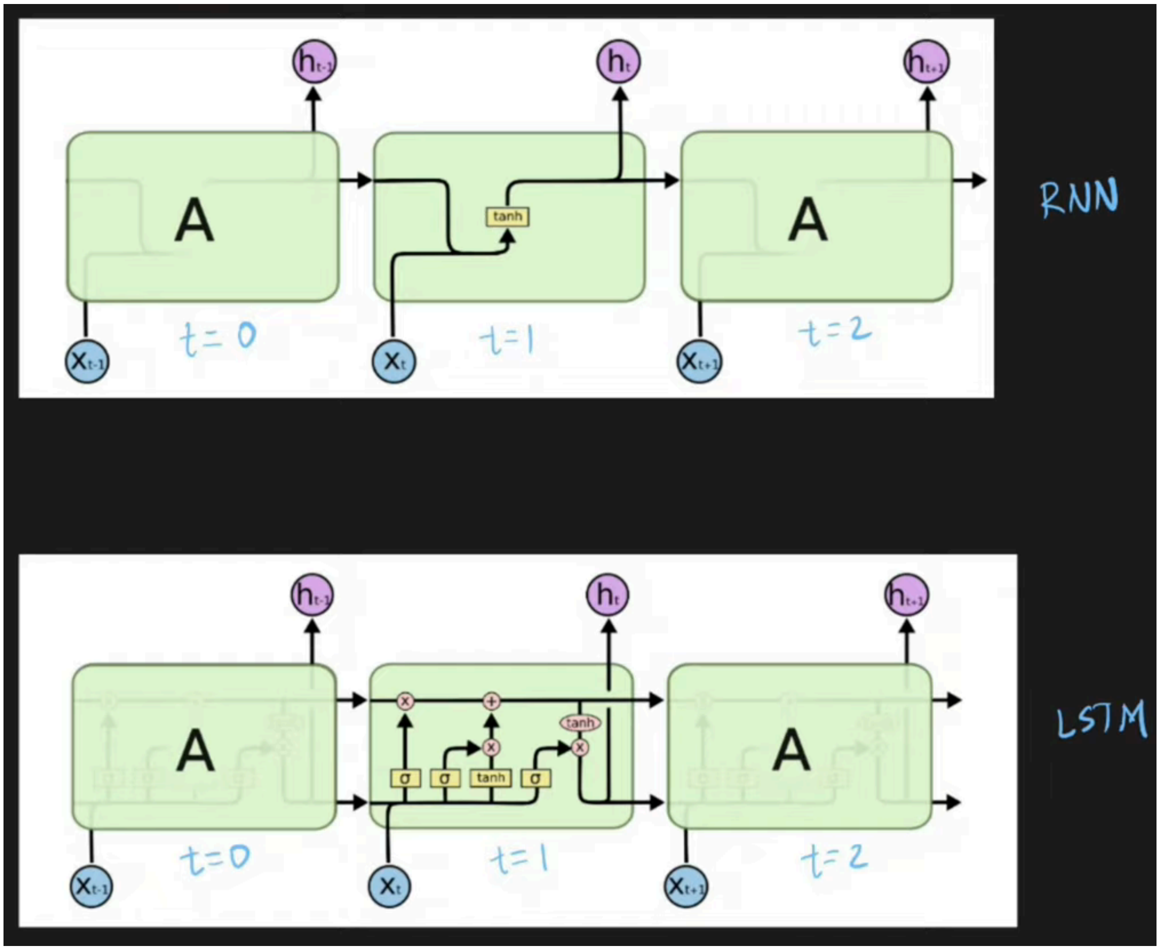
- **If you find a thing important, you put it in long term path.**
    - If you don't remove it, it will be there in the long term path till the end.



💡 **RNN has 2 paths.**

> **Communication between these 2 happen with the architect.**

⚙️ LSTM Cell Structure



Each LSTM unit has:

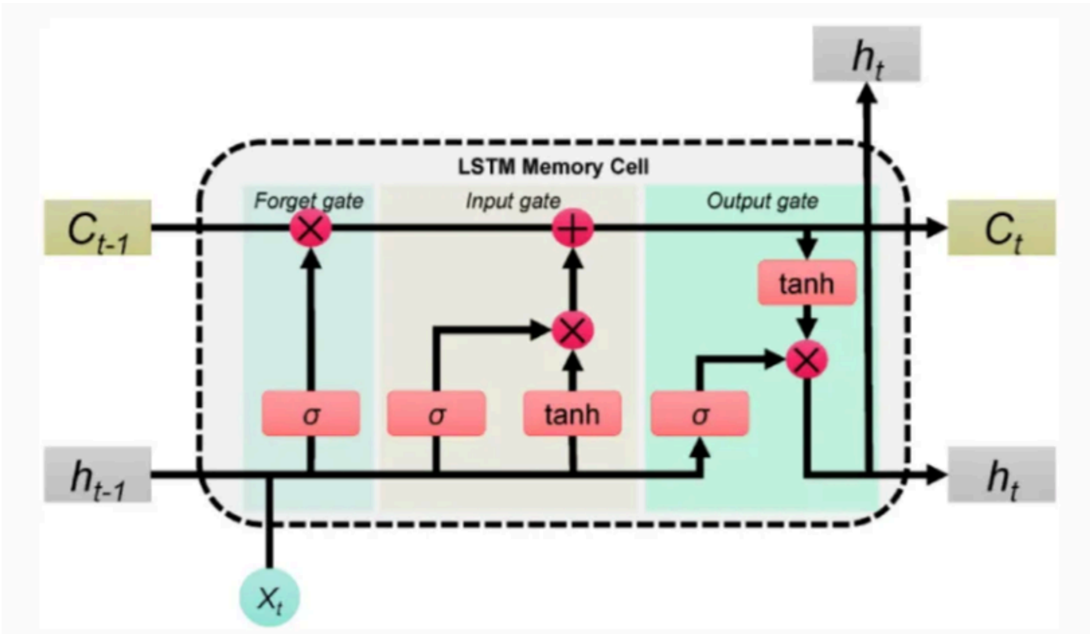
1. **Cell state** C_t:
    - **Long-term memory** (like a conveyor belt of info).
    - Main memory that flows through time

2. **Hidden state** h_t:
    - **Short-term memory** (output at each step).
    - Output passed to next time step

3. **Three gates**:
    a. **Forget Gate**: Decides **what information to discard** from the cell state

b. **Input Gate**: Decides **what new information to store** in the cell state

c. **Output Gate**: Decides **what to output** based on the cell state



- **Input:**
  - C_{t-1} → Long Term Memory of previous Time step (Cell State)
  - h_{t-1} → Short Term Memory of previous Time step
  - X_t→Current word
- **Output:**
  - Long Term Memory for next step
  - Short Term Memory for next step
- **Inside:**
  - Update Long Term Memory
  - Create Short Term Memory

🔷 1. Forget Gate (f_t)

- **Purpose**: Decide what **past memory** to forget.

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

- Uses a **sigmoid (σ)** function → output between 0 and 1
- If f_t = 0 → forget that info completely
- If f_t = 1 → keep it fully

🔷 2. Input Gate (i_t) + Candidate Memory ($\tilde{C}_t$)

Purpose: Decide what **new information** to add to memory.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)\tilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

- i_t: input gate (what to add)
- $\tilde{C}_t$: candidate content (new data to be stored)

🔷 3. Update Cell State (C_t)

Purpose: Update the main memory cell.

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

- Old memory is **partially forgotten**
- New memory is **partially added**
- This controlled flow keeps memory stable over time

🔷 4. Output Gate (o_t)

Purpose: Decide what to output at this time step.

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

- Final hidden state h_t is what's **sent to the next step**
- Also used as the **output** if needed

$\bar{C}_t \rightarrow$ candidate cell state

$$\underbrace{f_t}_{\text{forget gate}} \qquad \underbrace{i_t \qquad \bar{C}_t}_{\text{input gate}} \qquad \underbrace{o_t}_{\text{output gate}}$$

- The dimensions of above vectors are same.

🧠 Summary Flow of LSTM Cell at Time t :

1. Decide what to forget → f_t
2. Decide what to remember → i_t , $\tilde{C}_t$
3. Update memory → C_t
4. Decide what to output → h_t

Applications of LSTMs

1. Sequence prediction
2. Machine translation
3. Speech recognition
4. Time series forecasting
5. Text generation
6. Video analysis

Hyperparameter Considerations

1. **Number of units**: Typically between 64-1024
2. **Number of layers**: Usually 1-4
3. **Dropout**: Recurrent dropout is often helpful
4. **Initialization**: Orthogonal initialization for recurrent weights

Python Code for LSTM:

Import libraries:

```
# Import necessary tools
from tensorflow.keras.models import Sequential  # Basic neural network container
from tensorflow.keras.layers import Embedding, LSTM, Dense  # Layers we'll use
from tensorflow.keras.preprocessing.text import Tokenizer  # For text handling
from tensorflow.keras.utils import pad_sequences  # To make sequences same length
import numpy as np
```

Data:

```
texts = ["I loved this movie", "Hated the film", "Best movie ever", "Worst experi
ence"]
labels = np.array([1, 0, 1, 0])  # 1=positive, 0=negative
```

Step 1: Prepare the text data:

```
# Step 1: Prepare the text data
tokenizer = Tokenizer(num_words=10000)  # Keep top 10,000 words
tokenizer.fit_on_texts(texts)  # Learn all words in our texts
sequences = tokenizer.texts_to_sequences(texts)  # Convert words to number
s
data = pad_sequences(sequences, maxlen=100)  # Make all reviews 100 word
s long
```

Step 2: Build the model

```
# Step 2: Build the model
model = Sequential()  # Create empty model

# Add layers one by one:
# 1. Embedding: Turns word numbers into meaningful vectors
model.add(Embedding(input_dim=10000,  # How many unique words we have
                    output_dim=128))    # Size of each word vector

# 2. LSTM layer: Understands sequences in the text
model.add(LSTM(units=64))  # 64 memory units

# 3. Dense layer: Final decision maker (positive/negative)
model.add(Dense(1, activation='sigmoid'))  # 1 output: 0-1 probability
```

`64` →The number of neurons (or memory cells) inside the LSTM layer.

Compile:

```
model.compile(optimizer='adam',      # Smart learning algorithm
             loss='binary_crossentropy',  # How to measure errors
             metrics=['accuracy'])   # Track correct guesses
```

Fit:

```
# Step 4: Train the model
model.fit(data, labels,  # Our prepared data and answers
         epochs=5,      # How many times to see all data
         batch_size=32) # Process 32 reviews at once
```

Predict:

```
# Now the model can predict sentiments!
test_text = ["This film was okay"]
test_seq = tokenizer.texts_to_sequences(test_text)
test_data = pad_sequences(test_seq, maxlen=100)
prediction = model.predict(test_data)  # Returns something like [[0.65]] (65% positive)
prediction
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step

array([[0.48196715]], dtype=float32)
```