Deep-Learning-Module-4-Important-Topics-PYQs

For more notes visit

https://rtpnotes.vercel.app

1. How does a recursive neural network work?

A **Recursive Neural Network (RvNN)** is a type of neural network that processes data with a **hierarchical structure**, such as **trees or graphs**, instead of a simple sequence like RNNs. It is different from **Recurrent Neural Networks (RNNs)**, which process sequential data (like sentences or time-series).

🔹 **Key Idea**: Instead of processing data step by step (like RNNs), RvNN **builds a tree-like structure** where information from smaller parts is combined to form a larger understanding.

🔹 **Example**: Understanding a sentence based on **grammatical structure** rather than just word order.

How Does a Recursive Neural Network Work?

🔹 **Tree-structured processing**: RvNN processes data in a hierarchical manner, merging smaller units into larger representations.

🔹 **Example (Sentence Parsing)**:
Consider the sentence:
"(The cat) (sits (on the mat))"

Here, RvNN would first merge:

1. `"The"` + `"cat"` → **Phrase: "The cat"**
2. `"on"` + `"the mat"` → **Phrase: "on the mat"**
3. `"sits"` + `"on the mat"` → **Phrase: "sits on the mat"**
4. `"The cat"` + `"sits on the mat"` → **Final representation of the sentence**

Structure of Recursive Neural Network

```
         Sentence
        /        \
   Phrase 1   Phrase 2
  /      \      /      \
Word1  Word2  Word3   Word4
```

This hierarchical process allows the network to **understand relationships** between words beyond just their order.

Applications of Recursive Neural Networks

1. Natural Language Processing (NLP)

- **Sentence Parsing**: Understanding grammar structures.
- **Sentiment Analysis**: Understanding sentiment based on **hierarchical phrases** instead of just word order.

2. Computer Vision

- **Scene Understanding**: Understanding images in a **part-whole hierarchy**.
- Example: Recognizing a **face** from eyes, nose, and mouth.

Limitations of Recursive Neural Networks

1. Requires Predefined Structure

- Unlike RNNs, RvNNs need a **tree structure** before training.
- Example: In NLP, it needs a **predefined sentence structure**.

2. Computational Complexity

- Since it **builds trees**, it is more computationally expensive than RNNs.

3. Not Suitable for All Sequential Data

- If the data does **not have a clear hierarchy**, RvNNs may not be effective.

---

2. Explain the concept of ‘Unrolling through time’ in Recurrent Neural Networks.

RNNs are a type of neural network used when the input is a **sequence**, like:

- Text (sentence = sequence of words)
- Audio (sequence of sounds)
- Stock prices over time (sequence of numbers)
  The cool thing about RNNs is that they **remember** what happened before in a sequence.

The Problem: Normal neural networks can't remember the past.

In regular neural networks (like in image recognition), they just look at **one input at a time** and make a decision.
But in language or time series, you need to **remember what came before**. Like:

`“The cat sat on the ___.”`

To fill in the blank correctly, the model needs to remember **"The cat sat on the"** before it predicts "mat".

How RNNs Work (Quick idea):

RNNs have loops, so they can **pass information from one step to the next**.
Like

- At time step 1: input is “The” → RNN processes it and remembers something.
- At time step 2: input is “cat” → it uses the memory from “The” and updates it.
- And so on...

What is “Unrolling Through Time”?

RNNs are **reused at every time step**—but in the background, the model looks like it's **copying itself** for each word/time step.

Imagine this sentence:

> "I love AI"

You give one word at a time to the RNN.

Unrolling through time looks like this:

```
RNN1 → RNN2 → RNN3
 I      love     AI
```

Even though it’s the **same RNN model**, we **unroll** it across time steps to understand how the input flows.

Each copy:

- Takes the current word
- Takes memory from the previous step
- Produces an output
- Passes memory forward

So “unrolling through time” just means **showing how the RNN processes each part of the sequence, one step at a time**, using the same logic, while passing memory forward.

---

3. Explain applications of Recurrent Neural Network.

1. Autocomplete (Text Prediction)

> 📍 **Example:** You type "I love" and your phone suggests "cars", "coding", etc.

RNN looks at the previous words you typed and **predicts the next word** based on the sequence so far.

2. Translation (Machine Translation)

> 📍 **Example:** English → French
> "I am happy" → "Je suis content"

It reads the input sentence word by word, **remembers the meaning**, and then produces the translated sentence **in order**.

3. Named Entity Recognition (NER)

> 📍 **Example:**
> “Roger Federer” → Person
> “Honda City” → Car
> “Samsung Galaxy S10” → Product

RNNs understand the **context** of the sentence and figure out which words are **special names** (people, places, products, etc.)

4. Sentiment Analysis

> 📍 **Example:**
> A product review: “This phone is terrible.” → 1 star
> “This is amazing!” → 5 stars

RNN reads the sentence **word by word**, understands the emotion or opinion behind it, and gives a sentiment score (positive/negative).

---

4. Explain any three applications of LSTM

1. Handwriting Recognition

LSTMs can understand the sequence of strokes and letters in **unconstrained handwriting** (freehand writing, not printed), making it easier to recognize what is written.

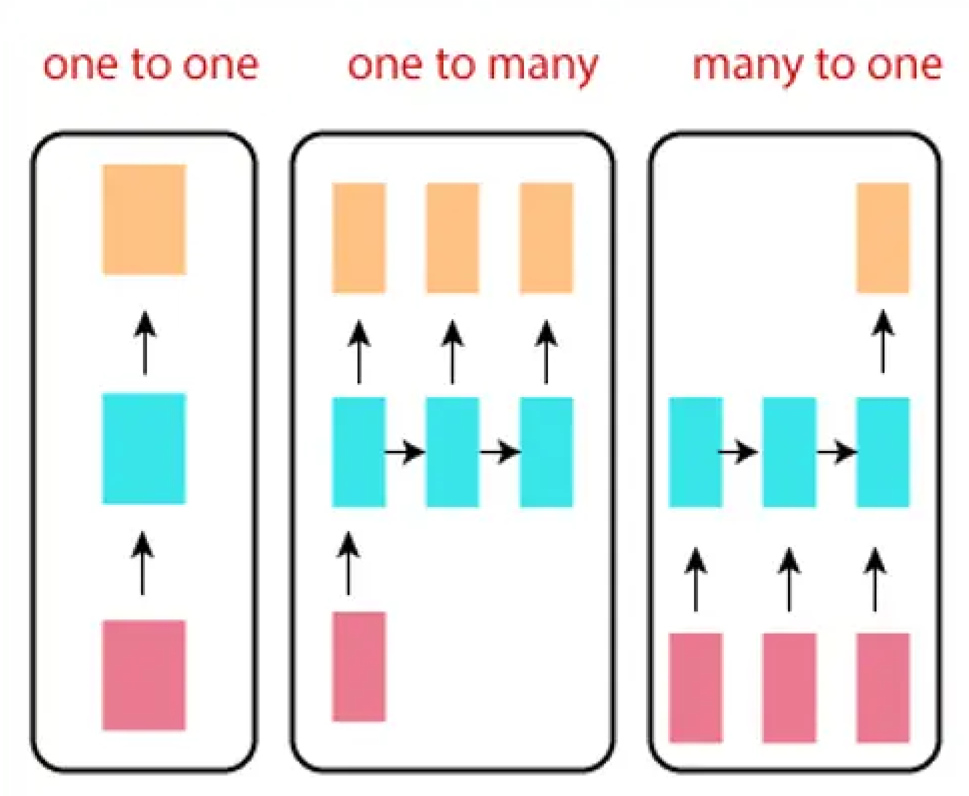
2. Speech Recognition

LSTMs are good at processing audio over time. They can listen to speech, remember important patterns, and convert it into text accurately.

3. Machine Translation

LSTMs help in translating a sentence from one language to another by remembering long-term dependencies between words.

5. Distinguish between One to Many and Many to One RNN.Also write one example application for each.



Feature	One-to-Many RNN	Many-to-One RNN
Input	Single input	Sequence of inputs
Output	Sequence of outputs	Single output
Data Flow	One input → multiple time steps of output	Multiple time steps of input → one final output
Use Case	When one input leads to a sequence of outputs	When a sequence leads to one final decision or label

Feature	One-to-Many RNN	Many-to-One RNN
Example Application	Image captioning (1 image → sentence of words)	Sentiment analysis (text → positive/negative label)

6. Describe how an LSTM takes care of the vanishing gradient problem.

What is the vanishing gradient problem?

- When training regular RNNs (Recurrent Neural Networks), we pass information from one step to the next. But as the network goes through many steps (like many words in a sentence), the learning signal — called the **gradient** — becomes very, very small.
- This is called the **vanishing gradient problem**, and it means the RNN “forgets” information from earlier steps.

What is an LSTM,

- An **LSTM** is a special kind of RNN that was **designed to remember things for longer periods**.
- It solves the vanishing gradient problem using a **clever design with gates**.

LSTM uses Gates to control the flow of information:

Think of it like a machine with switches (gates) that decide:

- **What to remember**
- **What to forget**
- **What to pass on**

Gate Type	Function	Example (Layman)
Forget Gate	Decides what to throw away from memory	“Forget what I had for breakfast”
Input Gate	Decides what new info to add to memory	“Remember this new concept”

Gate Type	Function	Example (Layman)
Output Gate	Decides what to send out as output	"Use this memory to answer the question"

How does this fix vanishing gradients?

- LSTM **preserves important information** by allowing gradients (learning signals) to **flow unchanged through time** using something called the **cell state**.
- This **cell state** is like a **conveyor belt** that carries information forward.
- If the model decides something is important, it **lets the information stay on the belt**, so the gradient **doesn't vanish**.

In simple terms: LSTM has **a memory** and **smart switches**

- It **remembers what's important** and **forgets what's not**
- So the network doesn't lose old info as it learns new things

---

7. Draw and explain the architecture of Recurrent Neural Networks.

What is an RNN?

A **Recurrent Neural Network (RNN)** is a type of neural network designed to handle **sequential data**, such as:

- Text (e.g., a sentence)
- Time series (e.g., stock prices)
- Audio or speech (e.g., spoken words)

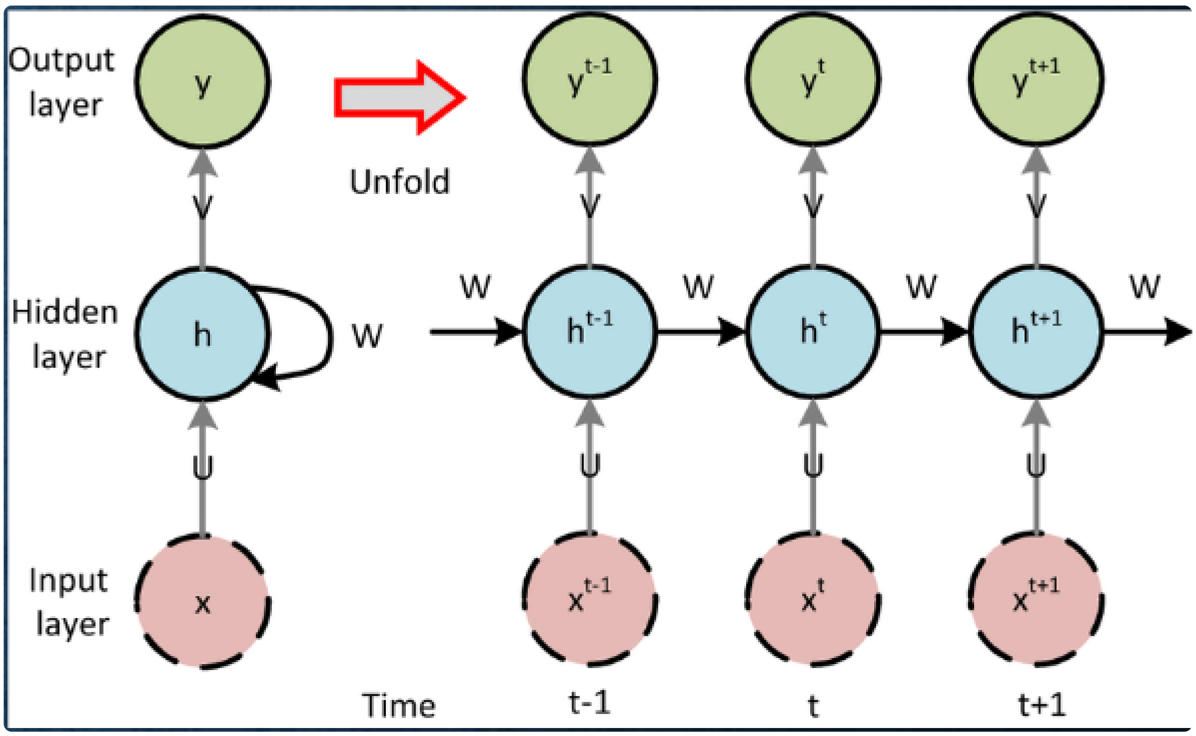
Unlike traditional neural networks that process all inputs independently, an RNN **remembers** what it has seen before by keeping a **hidden state** (memory). This makes it great for tasks where **order and context** matter.

How Does an RNN Work?

Imagine you're reading a sentence, one word at a time.
RNN works in a similar way:

- It reads one word (or input) at a time.
- It remembers what it has read before using a memory (hidden state).
- It uses that memory to help make better predictions or decisions.

Architecture



- The figure illustrates how a Recurrent Neural Network (RNN) operates by showing its architecture at a single time step and its process when unfolded over time.
- On the left, the diagram shows a single time step of an RNN.
  - Here: The input at the current time step x_t feeds into the hidden layer h_t.
  - The hidden state from the previous time step h_{t-1} influences h_t, enabling the network to retain past information.
  - The output y_t is generated based on the current hidden state h_t.
- On the right, the diagram depicts the RNN unfolded over three time steps (t-1, t, t+1).
  - The flow of information from one time step to the next is highlighted by arrows, showing how h_{t-1} influences h_t and subsequently h_{t+1}.
  - The weights W for the hidden states and U for the input connections remain shared across all time steps, making RNNs efficient for sequential tasks.

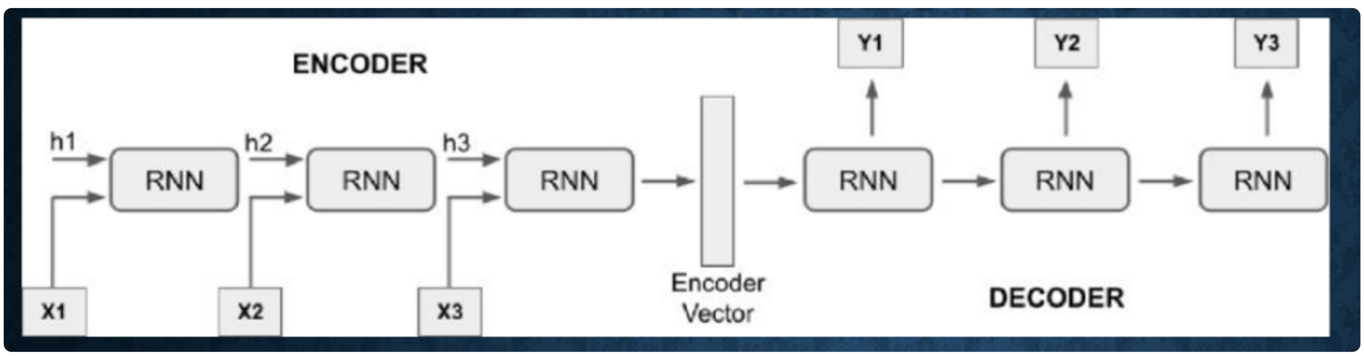
    - The outputs at each time step (y_{t-1}, y_t, y_{t+1}) reflect how the network processes and utilizes both current and past information.
- The red arrow labeled "Unfold" demonstrates how the RNN structure expands to handle sequences, making it useful for tasks like time-series prediction and sequence modeling

---

8. How does encoder-decoder RNN work?

- The **Encoder–Decoder architecture** is a neural network design used to convert **one sequence into another**, like:
    - Translating a sentence from **English** → **French**
    - Summarizing a paragraph
    - Converting a question into an answer
- Imagine someone listens to a sentence in English (encoder), remembers its meaning (context vector), and then speaks it in French (decoder).

1. Components of the Architecture



Encoder

- **Purpose**: Understand the input sequence and convert it into a summary called the **context vector**.
- **How it works**:
    - Takes input sequence like `["I", "am", "happy"]`
    - Passes each word through RNN cells → generates hidden states: `h₁`, `h₂`, `h₃`
    - Final hidden state (`h₃`) becomes the **context vector** — it contains the summary of the whole input

Decoder

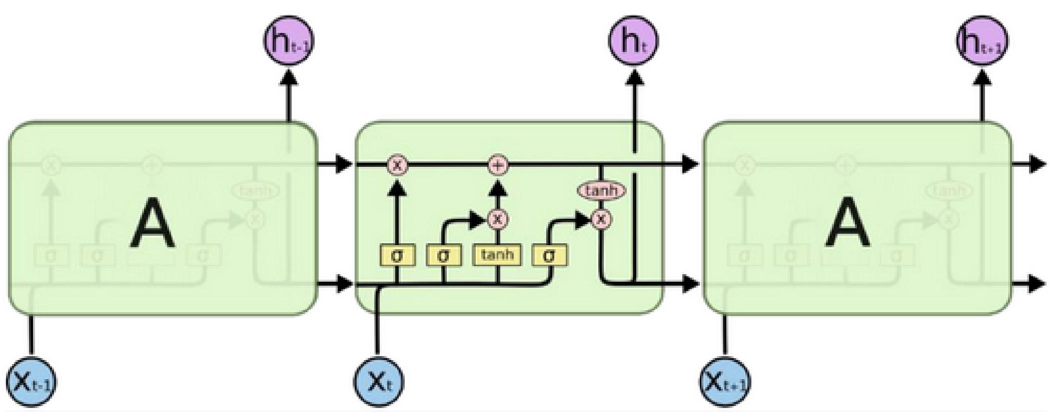
- **Purpose**: Use the context vector to generate the output sequence step-by-step.
- **How it works**:
  - Starts with the context vector from the encoder
  - Predicts the first output word (like `"Je"`)
  - Then, using `"Je"` and the context, predicts the next word (like `"suis"`)
  - Continues until the full translated sentence is produced

9. Draw and explain the architecture of LSTM.

What makes LSTM different from RNN?

- RNNs have a simple hidden layer (only `tanh`).
- LSTMs have a special **cell** with **gates** to control the flow of information.
- These gates help the network **remember important things** and **forget unimportant things** over time.

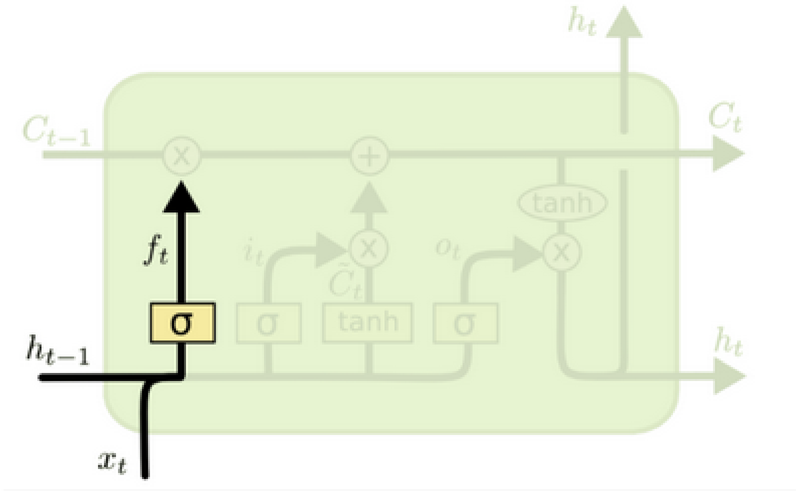
Main Idea of LSTM:



- LSTM has a **cell state** (long-term memory).
- It **adds or removes information** from the cell state using **gates**.
- Gates use values between `0` and `1` to control how much to let through (like a filter).

Components of LSTM Cell:

1. Forget Gate



Imagine:

You’re reviewing yesterday’s notes (**old memory**) and today’s topic (**new input**). But you can’t keep everything — you need to **decide what to forget**.

Equation terms

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

- **[h_{t-1}, x_t]** – Combine:
  - ht−1: last time step’s hidden state (summary)
  - xt: input at the current time step
- **Wf** – weights of the forget gate (learned during training)
- **bf** – bias of the forget gate
- **σ** – sigmoid function (squishes values between 0 and 1)
- **ft** – forget factor (how much of the past to erase)

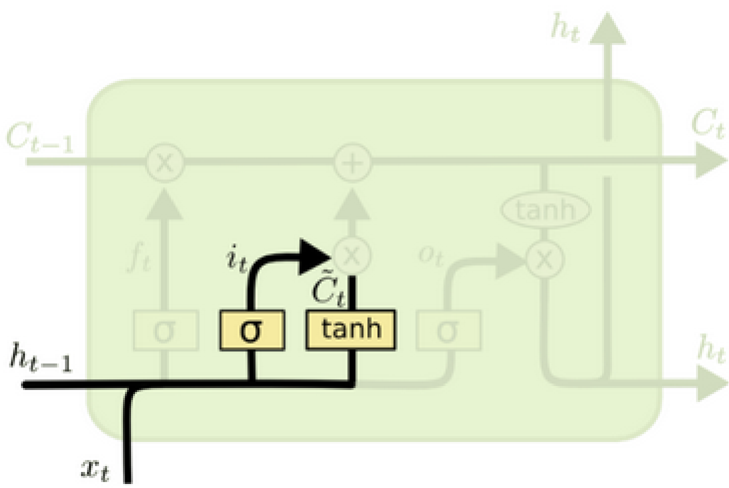
What Forget Gate does does:

- Combines yesterday's summary (`h_{t-1}`) and today’s input (`x_t`)
- Passes it through a forget filter (`W_f` , `b_f`) and the sigmoid function (which gives a number between 0 and 1)

- That number (`f_t`) tells the LSTM **how much of the old memory to erase**

Purpose: Decides what information to remove from the memory.

2. Input Gate



Step 1

Imagine:

You now **decide what new info to learn today**.

Equation terms

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

- **[h_{t-1}, x_t]** – same combo as before
- **Wi** – weights for the input gate
- **bi** – bias for the input gate
- **σ** – squashes output to [0, 1]
- **it** – input filter (how much new info to accept)

What Input Gate does:

- Same combo: yesterday’s brain + today’s input
- Runs it through another filter (input gate)

- The result (`i_t`) is a number between 0 and 1, telling how much **new info to allow into memory**

Step 2

Imagine:

Before writing new notes, your brain suggests **what it thinks should be added to memory**.

Equation terms

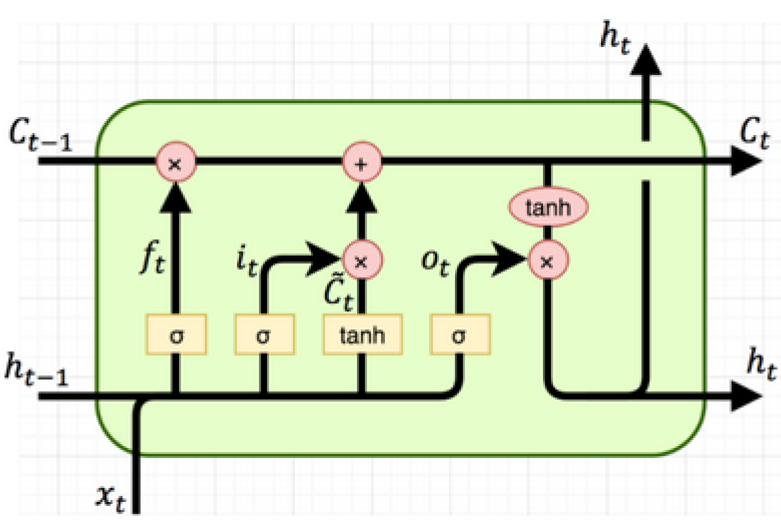
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

1. **[h_{t-1}, x_t]** – current context
2. **WC** – weights for candidate memory
3. **bC** – bias
4. **tanh** – maps values to [-1, 1]
5. **Ct** – suggested memory update (raw info)

What this does:

- Prepares the **new info** to be added
- Shapes it to fit the memory (values between -1 and 1 using `tanh`)

3. Output Gate



After updating the memory (cell state Ct), the LSTM needs to **decide what part of that memory to share** with the outside world (or the next time step). That's what the **Output Gate** does.

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

- **Filter memory using `tanh(C_t)`:**
  - Values in Ct are squashed into the range [-1, 1]
  - Big/important stuff → closer to ±1
  - Small/unimportant stuff → close to 0
- **Use the output gate ot to decide** what to actually say**:
  - ot ranges from 0 to 1
  - Acts like a **volume knob** — how much of each memory feature should be passed forward
- **Multiply both**:
  - You get a **filtered summary**: only relevant pieces of memory are shared

---

10. Discuss different ways to make a Recurrent Neural Network(RNN) deep RNN

What is an RNN (Recurrent Neural Network)?

A **Recurrent Neural Network (RNN)** is a type of neural network used for **sequential data** — like **sentences, time series, or music notes** — where the order of inputs matters.

Imagine you're reading a sentence word-by-word. An RNN remembers previous words while reading the current word — this is called **"memory"** or **"state"**.

Why Make an RNN "Deep"?

A **deep RNN** means stacking more RNN layers. This helps the network:

- Understand **more complex patterns**
- Remember information for a **longer time**

- Perform better in tasks like translation, text generation, etc.

How to Make an RNN Deep?

There are **3 common ways** to make an RNN deep. Let's look at each with simple diagrams and beginner-friendly explanation:

1. Deep in Time (Unrolling in Time)

- This is **not** truly deep, but it's how RNNs naturally work.
- You input a sequence step-by-step, and it processes each time step one by one.

```
x1 → [RNN] → h1 →
x2 → [RNN] → h2 →
x3 → [RNN] → h3 →
```

2. Deep in Space (Stacked RNN Layers)

This is the **most common way** to make RNNs deep.
We **stack multiple RNN layers** on top of each other — like multiple floors in a building. The output of one layer becomes the input to the next layer.

```
Time t
x1 ──▶ [RNN Layer 1] ──▶ [RNN Layer 2] ──▶ h1
x2 ──▶ [RNN Layer 1] ──▶ [RNN Layer 2] ──▶ h2
x3 ──▶ [RNN Layer 1] ──▶ [RNN Layer 2] ──▶ h3
```

This allows each layer to learn **different levels of patterns**:

- Lower layers: basic patterns (e.g., words)
- Higher layers: complex patterns (e.g., grammar)

3. Deep Transition RNN

In this design, **within each time step**, we have **multiple layers**.

```
x1 ──▶ [RNN Layer A] ──▶ [RNN Layer B] ──▶ h1
        (same time step)
```

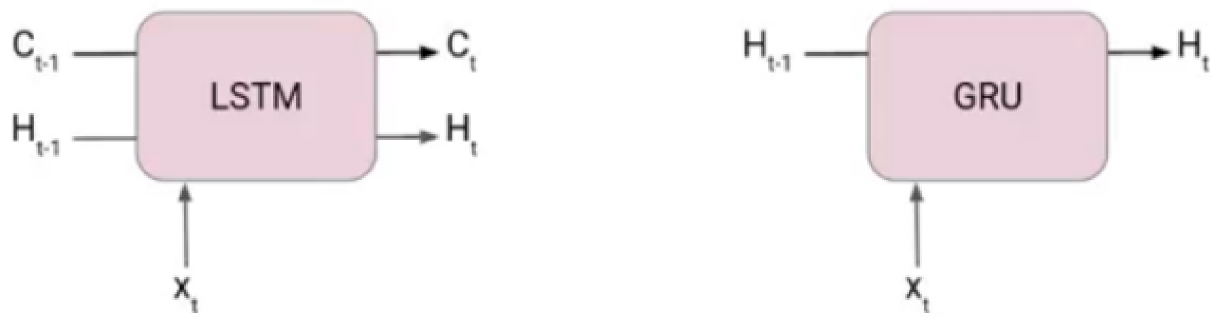
11. Discuss the architecture of Gated Recurrent Unit (GRU).

GRU is a **variant of Recurrent Neural Networks (RNNs)**, introduced in 2014 by **Kyunghyun Cho et al.** It addresses the **vanishing gradient problem** and helps retain long-term dependencies like the **LSTM**, but with a **simpler and faster architecture**.

Key Features:

- Simpler than LSTM (fewer gates).
- **No separate cell state (Ct)** — only uses **hidden state (Ht)**.
- Two gates: **Reset Gate** and **Update Gate**.
- Efficient and faster training than LSTM.

GRU Cell Architecture (at time step t)



Each GRU cell takes:

- **Input (xt)**
- **Previous hidden state (ht−1)**
  It returns:
- **New hidden state (ht)**

Component	Role
Reset gate	Discards irrelevant past information
Update gate	Chooses how much of new vs old information to retain
Alternative state	Candidate memory based on current input
Final hidden state	Smart mix of past and present for current output

12. Compare RNNs and Recursive Neural Networks

Feature	RNN (Recurrent Neural Network)	Recursive Neural Network
Structure	Linear sequence (like a chain)	Tree-like hierarchical structure
Input Type	Sequential data (e.g., time series, sentences)	Structured data with hierarchy (e.g., parse trees)
Flow of Data	Left to right (or right to left); one step after another	Bottom-up over tree structure
Used For	Language modeling, speech recognition, translation	Sentiment analysis, syntax parsing, scene understanding
Example Input	`"I am happy"` → processed word by word	Sentence parsed into a tree structure like `((I am) happy)`
Weight Sharing	Weights shared across time steps	Weights shared across tree nodes
Context Memory	Remembers previous steps via hidden states	Composes child nodes into parent nodes recursively
Architecture	Loops back to itself through time	Recursively merges child representations into a parent
Common Challenge	Vanishing gradient in long sequences	Parsing trees and defining the right structure

13. Explain the working of RNN. Also describe the different types of RNNs based on inputs and outputs. Mention one application for each.

What is an RNN?

An **RNN (Recurrent Neural Network)** is a type of neural network designed to handle **sequential data** (like sentences, time series, music, etc.).

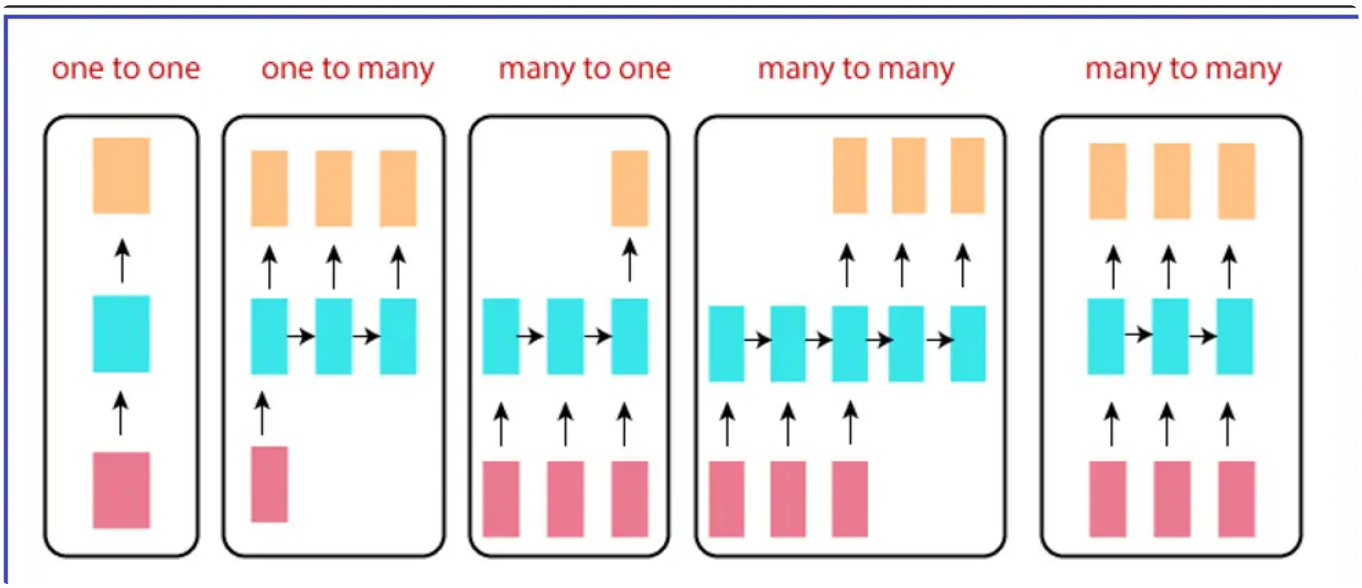
The special feature of an RNN is its **memory** — it remembers **past information** using a **hidden state**, which gets updated at every time step.

How does an RNN work?

1. **Input Sequence**: A sequence like `[x₁, x₂, x₃]` is given as input.
2. **Hidden State**: At each time step `t`, the RNN takes:
   - Current input x_t
   - Hidden state from the previous time step h_{t-1}
   - It calculates the new hidden state h_t, which stores memory
3. **Output**: Based on h_t, the output y_t is generated.

Think of it like reading a sentence word by word, and remembering the previous words to understand the next.

Types of RNNs Based on Input & Output



Type	Input	Output	Example	Application
1. One-to-One	Single input	Single output	Traditional Neural Network	Image classification
2. One-to-Many	Single input	Sequence output	Music generation from a theme	Music generation
3. Many-to-One	Sequence input	Single output	Sentiment analysis of a sentence	Sentiment analysis
4. Many-to-Many (Same length)	Sequence input	Sequence output	Part-of-speech tagging	POS tagging
5. Many-to-Many (Different length)	Sequence input	Sequence output	Machine translation	Language translation

Example Applications

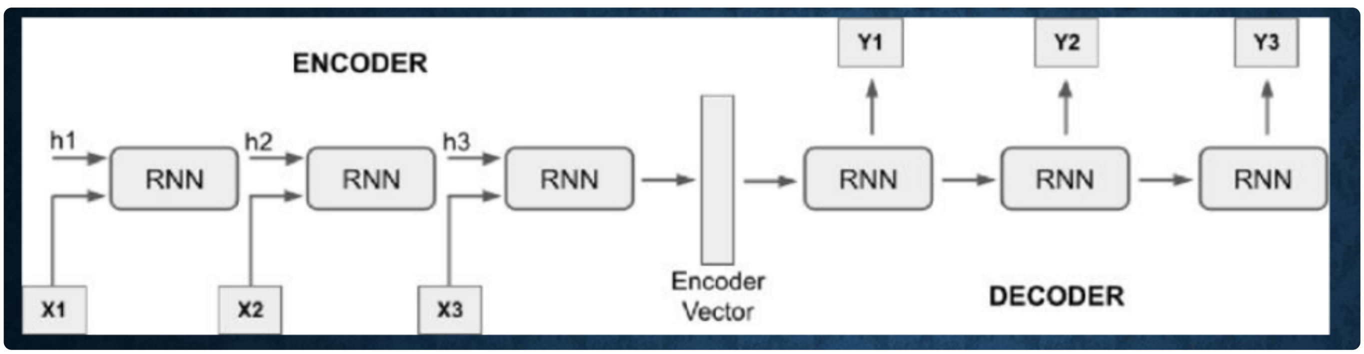
1. **One-to-Many**: Generate a melody (sequence) from a musical key
   → **Music generation**
2. **Many-to-One**: Input = sentence, Output = positive/negative
   → **Sentiment analysis**
3. **Many-to-Many**: Input = English sentence, Output = French sentence
   → **Language translation**

---

14. Name the simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence. With a neat sketch describe the working of it.

The **simplest RNN architecture** for mapping a **variable-length input sequence** to a **variable-length output sequence** is the **Encoder–Decoder (Sequence-to-Sequence) architecture**.

How it works



Encoder:

- Takes a **variable-length input sequence** like a sentence: `["I", "am", "happy"]`
- Processes each word using an RNN and produces **hidden states**
- The **last hidden state** is called the **context vector** or **encoder vector**
  - It summarizes the whole input sequence
  - Passed to the decoder

Decoder:

- Takes the **encoder vector** as input
- Starts generating the **output sequence** (e.g., translated sentence)
- It outputs one token at a time (`"Je"`, then `"suis"`, then `"heureux"`, etc.)
- The decoder RNN uses the previous output and hidden state to generate the next word