Understanding GRU

Minor in AI - IIT ROPAR

13th March, 2025

The Memory Dilemma: Building a Chatbot That Remembers

Imagine you are developing your own AI-powered chatbot, much like the famous ChatGPT. Your goal is to make it as intelligent and responsive as possible. One of the biggest challenges in building this chatbot is ensuring that it can predict the next word in a conversation accurately.

Let's say a user types:

> *"The cat jumped over the ____"*

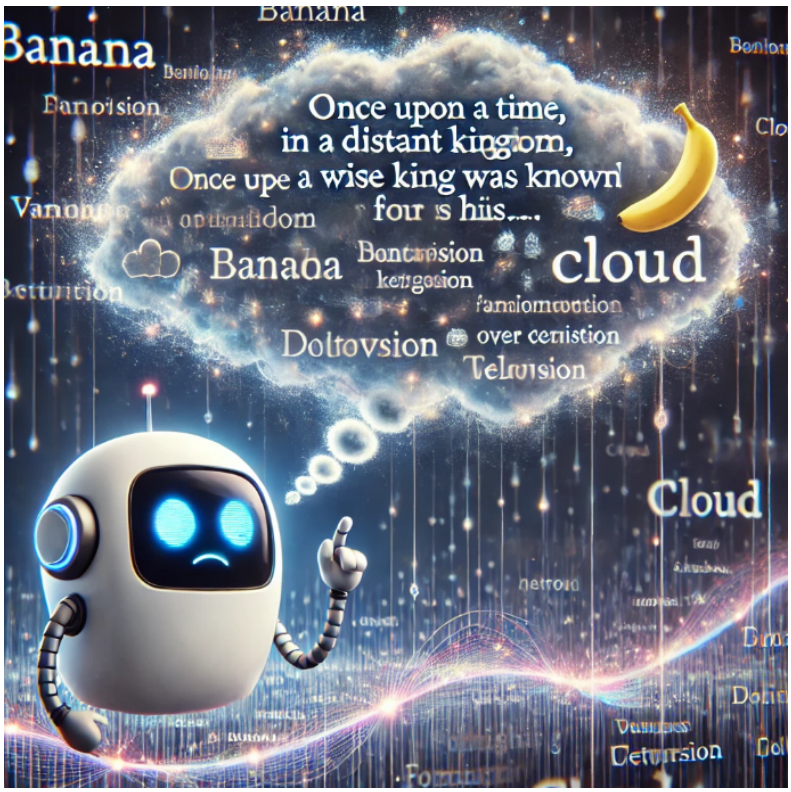
Your chatbot needs to analyze the sentence structure, retain context, and predict the most appropriate word—perhaps "fence" or "wall." But how does it know which word fits best? How does it remember the context from previous words?

You decide to start by using a basic Recurrent Neural Network (RNN). The idea seems promising—your chatbot will process each word sequentially and update its understanding at every step. However, as you test longer sentences, something goes wrong. Your chatbot starts forgetting crucial words that appeared earlier in the sentence.

For example, consider the sentence:

> *"Once upon a time, in a distant kingdom, a wise king ruled over the land. The king was known for his ____."*

You expect the model to predict "wisdom" or "justice," but instead, it outputs something unrelated. You dig deeper and realize that your model is suffering from a common problem known as the **vanishing gradient problem**.



The Challenge: Memory Loss in RNNs

RNNs process words one by one, updating their hidden state at each step. Theoretically, this hidden state should store everything necessary for making a correct prediction. However, in practice, when sentences become long, older words start fading from memory. The gradients used to train the model shrink exponentially, preventing it from learning meaningful long-term dependencies. This phenomenon is known as the **vanishing gradient problem**.

You realize that if your chatbot is going to be useful, it must remember important details from earlier in the conversation—just like humans do.

Vanishing Gradient Problem

When training a deep network using backpropagation, gradients become progressively smaller as they move backward through time. This causes earlier layers to stop learning, making it difficult for RNNs to retain information from the distant past.

Exploding Gradient Problem

On the other hand, gradients can also become excessively large, leading to unstable updates and making training difficult.

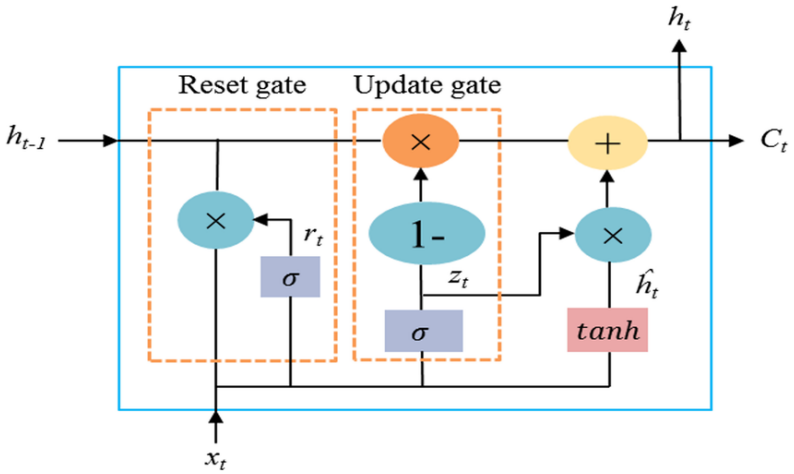
Due to these limitations, RNNs struggle to remember dependencies that span long sequences. This significantly impacts their performance in applications like language modeling, speech recognition, and translation.

How GRUs Solve This Problem

GRUs introduce two key mechanisms to manage information flow effectively:

- **Update Gate** (γ_u): Determines how much past information should be carried forward.
- **Reset Gate** (γ_r): Determines how much past information should be forgotten.

These gates allow GRUs to selectively retain relevant information from earlier time steps while discarding unnecessary details, thereby solving the vanishing gradient problem.



Detailed Breakdown of GRU Architecture

The **Gated Recurrent Unit (GRU)** is a type of **recurrent neural network (RNN)** designed to address the vanishing gradient problem in standard RNNs. It uses **gates** to control the flow of information, making it more efficient in handling long-term dependencies. The GRU has **two main gates**: - **Reset Gate** (γ_r) - **Update Gate** (γ_u)

At each **time step** t, the GRU **updates its hidden state** h_t using a series of operations that determine how much past information should be retained and how much new information should be incorporated.

Step 1: Compute the Reset Gate (γ_r)

The **reset gate** determines how much of the previous hidden state h_{t-1} should be forgotten before computing the candidate hidden state. If the reset gate is **close to 0**, it means that the previous state is mostly ignored. If it is **close to 1**, it means that the past state is largely retained.

Mathematical Formula:

$$\gamma_r = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{1}$$

Explanation of Variables:

- γ_r: **Reset gate activation** (a vector of values between 0 and 1).
- σ: **Sigmoid activation function**, which squashes values into the range $(0, 1)$.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- W_r: **Weight matrix** for the reset gate.
- h_{t-1}: **Previous hidden state** (capturing past information).
- x_t: **Current input** at time step t.
- b_r: **Bias term** for the reset gate.
- $[h_{t-1}, x_t]$: **Concatenation** of the previous hidden state and the current input.

Step 2: Compute the Update Gate (γ_u)

The **update gate** controls how much of the **previous hidden state** should be **carried forward** versus how much should be updated with new information.

Mathematical Formula:

$$\gamma_u = \sigma(W_u[h_{t-1}, x_t] + b_u) \tag{2}$$

Explanation of Variables:

- γ_u: **Update gate activation** (a vector of values between 0 and 1).
- W_u: **Weight matrix** for the update gate.
- b_u: **Bias term** for the update gate.
- σ: **Sigmoid activation function**.

Step 3: Compute the Candidate Hidden State ($\tilde{h}_t$)

The **candidate hidden state** is the **new potential hidden state** that replaces the old hidden state. It is computed using a combination of: - The current input x_t - The previous hidden state h_{t-1}, **scaled by the reset gate** γ_r

Mathematical Formula:

$$\tilde{h}_t = \tanh(W_h[\gamma_r * h_{t-1}, x_t] + b_h) \tag{3}$$

Explanation of Variables:

- $\tilde{h}_t$: **Candidate hidden state** (a possible new state to replace the previous one).
- γ_r: **Reset gate** (determines how much of the past should be used).
- W_h: **Weight matrix** for computing the candidate state.
- b_h: **Bias term** for the candidate state.
- tanh: **Hyperbolic tangent activation function**, which squashes values between -1 and 1.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Step 4: Compute the Final Hidden State (h_t)

The final hidden state is a **weighted combination** of the previous hidden state h_{t-1} and the new candidate state $\tilde{h}_t$. This is controlled by the **update gate** γ_u.

Mathematical Formula:

$$h_t = \gamma_u * h_{t-1} + (1 - \gamma_u) * \tilde{h}_t \tag{4}$$

Explanation of Variables:

- h_t: **Final hidden state at time step** t.
- γ_u: **Update gate** (determines how much old state vs. new state to keep).
- h_{t-1}: **Previous hidden state**.
- $\tilde{h}_t$: **Candidate hidden state**.

Summary of the GRU Mechanism

Step	Gate/State	Role
Step 1	Reset Gate (γ_r)	Decides how much of the past should be forgotten.
Step 2	Update Gate (γ_u)	Controls how much of the previous state is kept.
Step 3	Candidate Hidden State ($\tilde{h}_t$)	Computes a new potential hidden state.
Step 4	Final Hidden State (h_t)	Combines old and new states based on the update gate.

Table 1: Summary of GRU Mechanism

Example: How GRUs Work in Practice

Consider the sentence: "The cat is sleeping."

- The word "The" is passed through the GRU, generating an initial hidden state.
- The word "cat" is processed, and the reset gate decides whether "The" is still relevant.
- The word "is" is processed, and the update gate determines how much past context to retain.
- The word "sleeping" is processed, and the final prediction is made using the retained context.

If the sentence were longer, GRUs would ensure that essential words (like "cat") remain in memory until needed.

Why GRUs Are Better than Standard RNNs

- They solve the vanishing gradient problem by selectively remembering information.
- They handle long-term dependencies better than standard RNNs.
- They have fewer parameters compared to LSTMs, making them more computationally efficient.

Key Takeaways

- **GRUs effectively handle long-term dependencies** by using gating mechanisms that decide what information should be retained and what should be forgotten.
- **The vanishing gradient problem in standard RNNs** makes it difficult for them to retain useful context from earlier words in long sequences.
- **The update gate** in a GRU helps determine how much of the previous memory should be carried forward.
- **The reset gate** decides how much of the past state should be forgotten when computing the new hidden state.
- **GRUs are computationally efficient** compared to LSTMs, as they use fewer gates while still improving performance over standard RNNs.
- **GRUs are widely used in NLP applications**, including chatbots, machine translation, and speech recognition, due to their ability to manage sequential data effectively.
- **Choosing between GRU and LSTM** depends on the specific use case—GRUs are faster and simpler, while LSTMs provide finer control over long-term dependencies.