Recurrent Neural Networks and Gated Architectures

Vaibhav Chourasia

January 26, 2026

Introduction

This report provides a detailed explanation of Recurrent Neural Networks (RNNs), their limitations, and the advanced architectures developed to overcome these limitations.

1 What Are Recurrent Neural Networks?

1.1 The Problem with Standard Neural Networks

Before understanding RNNs, we need to see why regular neural networks don't work well for sequences like sentences, time series data, or any data where order matters.

Imagine trying to predict the next word in this sentence: "The cat sat on the ___". A regular neural network would look at each word independently and wouldn't understand that "cat" came before "sat" and "on". It would treat the input as separate, unrelated words rather than a connected sequence.

This is because in a standard neural network:

- Each input is processed independently
- There's no memory of previous inputs
- The network structure doesn't change based on what came before

1.2 The Basic Idea of RNNs

Recurrent Neural Networks solve this problem by having **memory**. They maintain an internal state (called the hidden state) that gets updated with each new input. This hidden state serves as a summary of everything the network has seen so far.

Think of it like reading a book: as you read each page, you remember what happened on previous pages. Your understanding of the current page depends on your memory of earlier pages. RNNs work similarly.

1.3 Mathematical Formulation

Let's define the basic equations for an RNN. At time step t:

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{1}$$

$$y_t = W_{hy}h_t + b_y \tag{2}$$

Where:

- x_t is the input at time t (could be a word, a temperature reading, etc.)
- h_t is the hidden state at time t (the "memory")
- h_{t-1} is the hidden state from the previous time step
- y_t is the output at time t
- W_{hh}, W_{xh}, W_{hy} are weight matrices that the network learns
- b_h, b_y are bias terms
- f is an activation function (usually tanh or ReLU)

The key part is $W_{hh}h_{t-1}$. This term takes the previous hidden state and transforms it to contribute to the current hidden state. This creates the recurrence - each state depends on the previous one.

1.4 Visualizing the Recurrence

We can visualize an RNN in two ways:

1. **The folded view**: Shows a single RNN cell with a recurrent connection

$$\begin{tikzcd}
\boxed{x_t} \arrow[r] & h_t \arrow[r] \arrow[loop above, "h_{t-1}"] & \boxed{y_t}
\end{tikzcd}$$

2. **The unfolded view**: Shows the RNN stretched out over time

$$\begin{tikzcd}
\boxed{x_1} \arrow[r] & h_1 \arrow[r] \arrow[d] & \boxed{y_1} \\
\boxed{x_2} \arrow[r] & h_2 \arrow[r] \arrow[dd] & \boxed{y_2} \\
 & & \cdots \\
\boxed{x_3} \arrow[r] & h_3 \arrow[r] & \boxed{y_3}
\end{tikzcd}$$

The unfolded view makes it clear that information flows from earlier time steps to later ones through the hidden states.

2 Training RNNs: Backpropagation Through Time

2.1 The Need for a Special Training Algorithm

Training RNNs requires a modified version of backpropagation called Backpropagation Through Time (BPTT). The reason is simple: in a regular neural network, each input produces an output, and we can compute the error and update weights independently. But in an RNN, the output at time t depends on all previous inputs, so the error at time t affects all the weights that contributed to it, going back to the beginning of the sequence.

2.2 How BPTT Works

Imagine we have a sequence of length 3. The unfolded network looks like:

$$x_1 \to h_1 \to y_1$$
$$x_2 \to h_2 \to y_2 \quad (\text{with } h_1 \to h_2)$$
$$x_3 \to h_3 \to y_3 \quad (\text{with } h_2 \to h_3)$$

When we compute the loss (error) at time 3 (L_3), it depends on y_3, which depends on h_3, which depends on h_2, which depends on h_1. So to update the weights that connect h_1 to h_2 (which is W_{hh}), we need to propagate the error backward through time:

$$\frac{\partial L_3}{\partial W_{hh}} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_{hh}} + \cdots \tag{3}$$

This chain of derivatives goes back through all time steps.

2.3 The Mathematics of BPTT

Let's derive the gradients properly. For simplicity, let's consider the loss at the final time step T, denoted as L_T.

The gradient with respect to W_{hh} is:

$$\frac{\partial L_T}{\partial W_{hh}} = \sum_{k=1}^{T} \frac{\partial L_T}{\partial h_T}\frac{\partial h_T}{\partial h_k}\frac{\partial h_k}{\partial W_{hh}} \tag{4}$$

The tricky part is $\frac{\partial h_T}{\partial h_k}$. Using the chain rule:

$$\frac{\partial h_T}{\partial h_k} = \prod_{t=k}^{T-1} \frac{\partial h_{t+1}}{\partial h_t} \tag{5}$$

From Equation 1, we have $h_t = f(z_t)$ where $z_t = W_{hh}h_{t-1} + W_{xh}x_t + b_h$. So:

$$\frac{\partial h_{t+1}}{\partial h_t} = f'(z_{t+1}) \cdot W_{hh} \tag{6}$$

Therefore:

$$\frac{\partial h_T}{\partial h_k} = \prod_{t=k}^{T-1} f'(z_{t+1}) \cdot W_{hh} \tag{7}$$

This is a product of terms, each containing W_{hh}.

3 The Vanishing Gradient Problem

3.1 Understanding the Problem

The vanishing gradient problem occurs when these repeated multiplications cause gradients to become extremely small (vanish) or extremely large (explode). Let's understand why this happens.

Consider what happens when we multiply many numbers together:

- If each number is less than 1, the product gets smaller and smaller
- If each number is greater than 1, the product gets larger and larger
- If the numbers are exactly 1, the product stays 1

In our gradient computation, we're multiplying terms of the form $f'(z_{t+1}) \cdot W_{hh}$. The derivative $f'(z)$ for common activation functions like tanh or sigmoid is always between 0 and 1 (for sigmoid, it's between 0 and 0.25; for tanh, between 0 and 1).

So we have:

$$\frac{\partial h_T}{\partial h_k} = \prod_{t=k}^{T-1} (\text{number between 0 and 1}) \cdot W_{hh} \tag{8}$$

If W_{hh} has values less than 1 (which is typical), then each term in the product is less than 1. Multiplying many numbers less than 1 gives a result that approaches zero exponentially fast.

3.2 Why This Matters

The vanishing gradient problem has serious consequences:

1. **The network becomes short-sighted**: It can only learn dependencies between events that are close together in time. For weather prediction, this means it might learn that today's temperature is similar to yesterday's, but not that seasonal patterns repeat yearly.
2. **Training becomes very slow**: Early layers in time (early time steps) receive almost no gradient updates, so they learn very slowly or not at all.
3. **The network cannot capture long-term patterns**: Any pattern that requires remembering information from many steps ago cannot be learned.

3.3 The Exploding Gradient Problem

The opposite can also happen: if the values in W_{hh} are large, the gradients can explode (become extremely large). This causes numerical instability and makes training impossible. Fortunately, this problem is easier to fix with techniques like gradient clipping.

4 Long Short-Term Memory (LSTM) Networks

4.1 The Core Idea

Long Short-Term Memory (LSTM) networks were invented to solve the vanishing gradient problem. The key insight is to create a path through time where gradients can flow unchanged. This is achieved through a clever architecture with **gates** that control information flow.

Think of it like a conveyor belt running through the entire sequence. Information can hop on the conveyor belt and ride along unchanged for as long as needed. Gates control what gets on the belt, what stays on, and what gets off.

4.2 Anatomy of an LSTM Cell

An LSTM cell has three gates and maintains two states:

1. **Cell state (C_t)**: The conveyor belt that runs through the entire sequence
2. **Hidden state (h_t)**: The output of the cell at each time step
3. **Forget gate (f_t)**: Decides what information to remove from the cell state
4. **Input gate (i_t)**: Decides what new information to add to the cell state
5. **Output gate (o_t)**: Decides what parts of the cell state to output

4.3 The Mathematical Formulation

Let's go through each component step by step.

4.3.1 Forget Gate

The forget gate looks at the previous hidden state h_{t-1} and the current input x_t, and outputs a number between 0 and 1 for each element in the cell state C_{t-1}. A 1 means "keep this completely", a 0 means "forget this completely".

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{9}$$

Here:

- σ is the sigmoid function, which outputs values between 0 and 1
- $[h_{t-1}, x_t]$ means we concatenate (combine) h_{t-1} and x_t into one vector
- W_f and b_f are weights and biases that the network learns

4.3.2 Input Gate and Candidate Values

The input gate decides which values we'll update. We also create candidate values $\tilde{C}_t$ that could be added to the cell state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{10}$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{11}$$

4.3.3 Updating the Cell State

Now we update the old cell state C_{t-1} into the new cell state C_t:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{12}$$

Here $\odot$ means element-wise multiplication (multiplying corresponding elements). This equation is the heart of the LSTM.

- $f_t \odot C_{t-1}$: We multiply the old cell state by the forget gate. If f_t is 0 for an element, that information is forgotten. If it's 1, it's kept.
- $i_t \odot \tilde{C}_t$: We multiply the candidate values by the input gate. This determines how much of each candidate value to add.
- The sum gives us the new cell state.

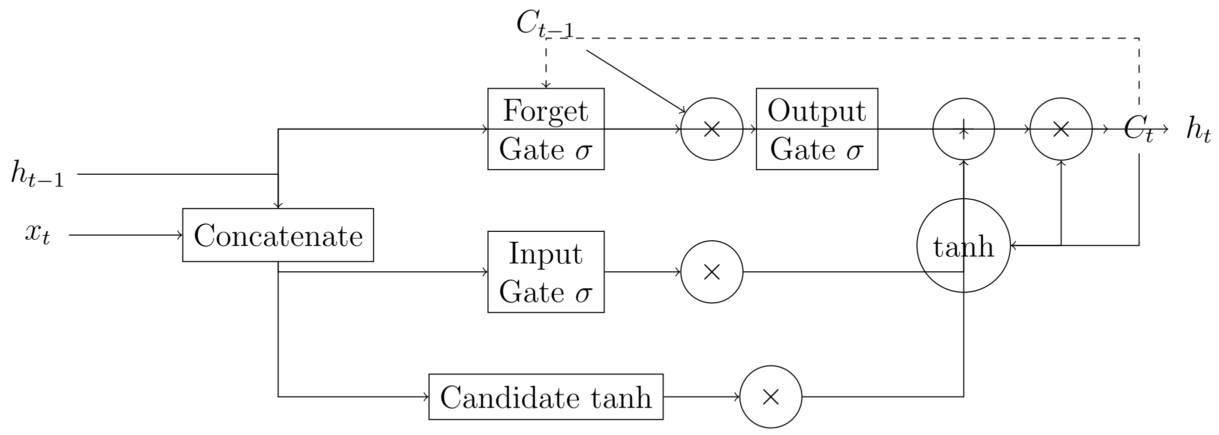
4.3.4 Output Gate

Finally, we decide what to output. The output is based on the cell state, but we filter it first:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{13}$$
$$h_t = o_t \odot \tanh(C_t) \tag{14}$$

The output gate o_t decides which parts of the cell state to output. We put the cell state through tanh (to push values between -1 and 1) and multiply by the output gate.

4.4 Visualizing the LSTM Cell



4.5 Why LSTMs Solve the Vanishing Gradient Problem

The key is in the cell state update equation:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{15}$$

Let's look at the gradient flow. The derivative of C_t with respect to C_{t-1} is:

$$\frac{\partial C_t}{\partial C_{t-1}} = f_t + (\text{terms from gates}) \tag{16}$$

f_t appears **additively**, not multiplicatively. If the forget gate f_t is close to 1 (which the network can learn to do), then:

$$\frac{\partial C_t}{\partial C_{t-1}} \approx 1 \tag{17}$$

This means the gradient can flow through the cell state almost unchanged. Even over many time steps:

$$\frac{\partial C_T}{\partial C_1} = \prod_{t=2}^{T} \frac{\partial C_t}{\partial C_{t-1}} \approx 1 \cdot 1 \cdot \dots \cdot 1 = 1 \tag{18}$$

The gradient doesn't vanish. The network learns to set the forget gate to 1 for information that needs to be remembered for a long time.

This is often called the **constant error carousel** because errors can circulate in the cell state without decaying.

5 Gated Recurrent Units (GRU)

5.1 Motivation for GRUs

While LSTMs work very well, they have a lot of parameters (three sets of gates plus the candidate values). Gated Recurrent Units (GRUs) were proposed as a simpler alternative that combines some of the LSTM concepts into a more compact form.

GRUs merge the cell state and hidden state into a single state, and combine the forget and input gates into a single update gate.

5.2 GRU Architecture

A GRU has two gates:

1. **Update gate** (z_t): Controls how much of the previous state to keep
2. **Reset gate** (r_t): Controls how much of the previous state to use for computing the new candidate

5.3 Mathematical Formulation

5.3.1 Update Gate

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{19}$$

5.3.2 Reset Gate

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{20}$$

5.3.3 Candidate Activation

$$\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t] + b) \tag{21}$$

Notice that we multiply h_{t-1} by the reset gate r_t before using it to compute the candidate. If r_t is 0, we ignore the previous state completely.

5.3.4 Hidden State Update

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{22}$$

This is similar to the LSTM cell state update. The update gate z_t decides the blend of old and new information.

5.4 Comparing GRU to LSTM

- **Similarities**: Both use gating mechanisms to control information flow. Both have additive updates that help with gradient flow.
- **Differences**:
  1. GRUs have 2 gates (update and reset) vs. LSTM's 3 gates (forget, input, output)
  2. GRUs combine the cell state and hidden state into a single state
  3. GRUs have fewer parameters, so they train faster
  4. The reset gate allows GRUs to effectively drop irrelevant information from the past
- **When to use which**:
  - LSTMs are often more powerful for very complex tasks with long sequences
  - GRUs are good when training data is limited or computation resources are constrained
  - In practice, the performance difference is often small, and GRUs are becoming more popular due to their simplicity

5.5 Why GRUs Also Solve the Vanishing Gradient Problem

Look at the GRU update equation:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{23}$$

The derivative of h_t with respect to h_{t-1} has a term $(1 - z_t)$. If the network learns to set z_t close to 0 (don't update much), then:

$$\frac{\partial h_t}{\partial h_{t-1}} \approx 1 \tag{24}$$

Again, we get near-unity gradients that don't vanish.

6 The Progression from RNNs to GRUs to LSTMs

The development of these architectures followed a logical progression:

1. **Basic RNNs (1980s)**: Introduced the idea of recurrence but suffered from vanishing gradients
2. **LSTMs (1997)**: Solved vanishing gradients with gating mechanisms and a separate cell state
3. **GRUs (2014)**: Simplified LSTMs by combining gates and states

6.1 Key Innovations at Each Stage

- **RNNs**: Introduced parameter sharing across time and hidden states as memory
- **LSTMs**: Introduced:
  1. Separate cell state for long-term memory
  2. Gating mechanisms (forget, input, output gates)
  3. Additive updates for gradient flow
  4. The ability to learn what to remember and what to forget
- **GRUs**: Simplified by:
  1. Merging cell state and hidden state
  2. Combining forget and input gates into update gate
  3. Adding reset gate for flexible memory management

6.2 Mathematical Patterns in the Evolution

All three architectures can be seen as different ways to compute a hidden state h_t:

- **RNN**: $h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$
- **LSTM**: More complex but with the key cell state update: $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$
- **GRU**: $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$

Notice the pattern: both LSTM and GRU use **additive updates** (sums) rather than just transformations. This is what prevents gradient vanishing.

7 Conclusion

The journey from basic RNNs to LSTMs and GRUs represents a significant advancement in neural network design for sequential data. The key breakthrough was understanding and solving the vanishing gradient problem through gating mechanisms and additive updates.