Recurrent Neural Networks

Week 3 - Teaching Networks to Remember

NLP Course 2025

September 22, 2025

From Feedforward to Recurrent: Adding Memory to Neural Networks

Week 3: The Memory Revolution

Processing Sequences with State

The Challenge

- Sequential data everywhere
- Order matters
- Variable length inputs
- Long-term dependencies

The Solution

- **Recurrent connections**
- Hidden state memory
- Parameter sharing
- Backprop through time

The Evolution

- Vanilla RNN $\rightarrow$ LSTM
- Solving gradients
- Gating mechanisms
- Modern variants (GRU)

The foundation of sequence modeling before transformers

Course Journey: Where We Are

Journey So Far:

- Week 1: Statistical language models (n-grams)
- Week 2: Word embeddings (Word2Vec, dense vectors)
- **Week 3: Sequential processing with memory**

Coming Next:

- Week 4: Encoder-decoder architectures
- Week 5: Attention mechanisms
- Week 6+: Transformers and beyond

Before We Dive Into RNNs

What We'll Cover:

- What is a neural network?
- How do neurons compute?
- Why activation functions?
- What are weight matrices?
- How do networks learn?

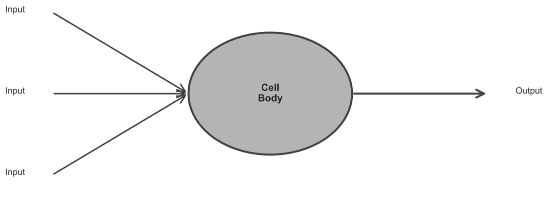
Why This Matters:

- RNNs are neural networks with loops
- Need to understand basic building blocks
- Gradients crucial for training
- Matrices organize computations

10 minutes to build your neural network intuition

From Biology to Math: What is a Neural Network?

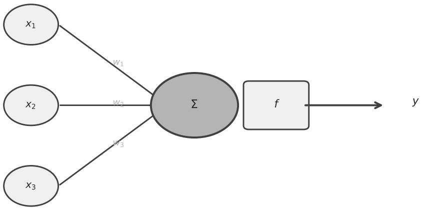
Biological Neuron



Biological Neuron

- Receives signals from other neurons
- Processes in cell body
- Fires if signal strong enough
- Sends output to next neurons

Mathematical Neuron



$y = f(\sum w_i x_i + b)$

- Inputs: numbers (x_1, x_2, x_3)
- Weights: importance (w_1, w_2, w_3)
- Sum: $z = \sum w_i x_i + b$
- Output: $y = f(z)$ where f is activation

The Simplest Neural Network: One Neuron

Single Neuron Computation

1. Inputs: $x_1 = 0.5$, $x_2 = 0.3$

2. Weights: $w_1 = 2.0$, $w_2 = -1.0$

3. Weighted sum: $(0.5 \times 2.0) + (0.3 \times -1.0) = 0.7$

4. Add bias: $0.7 + 0.1 = 0.8$

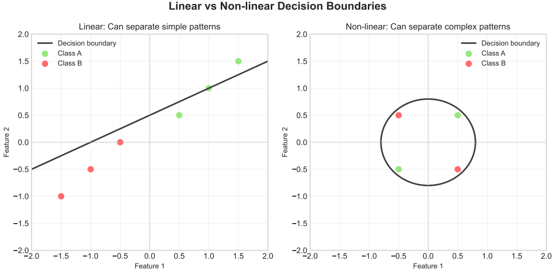
5. Apply activation: $\tanh(0.8) = 0.664$

Output: 0.664

Why Activation Functions?

Without Activation (Linear)

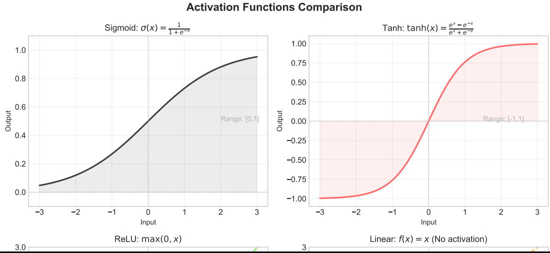
- Just weighted sums
- $y = w_1 x_1 + w_2 x_2$
- Can only learn straight lines
- Multiple layers = still just lines!
- Cannot solve XOR problem



With Activation (Non-linear)

- Adds curves and bends
- Can learn complex patterns
- Different activations for different uses:

Function	Use Case
Sigmoid	Probabilities [0,1]
Tanh	Centered [-1,1]
ReLU	Hidden layers
Softmax	Multi-class



Deep Dive: The Tanh Activation

What is tanh?

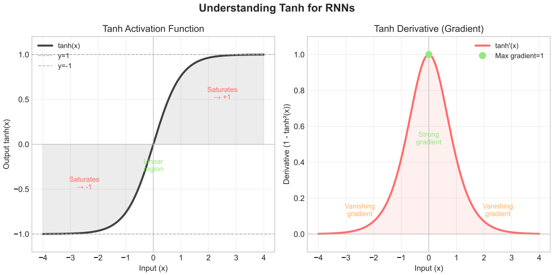
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Properties:

- Output range: [-1, 1]
- Centered at zero
- Smooth and differentiable
- Derivative: $1 - \tanh^2(x)$

Why RNNs use tanh:

- Keeps values bounded
- Zero-centered helps learning
- Smooth gradients
- Historical convention



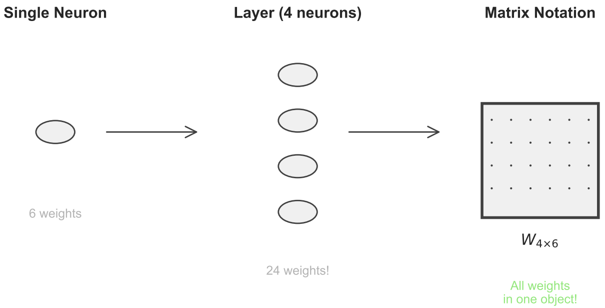
Intuition:

- Strong positive $\rightarrow +1$
- Strong negative $\rightarrow$ -1
- Near zero $\rightarrow$ linear
- Natural "squashing"

Tanh prevents values from exploding while maintaining gradients

Matrices: Organizing Many Connections

From Single Neurons to Layers



Matrix Multiplication:

$$y = Wx + b$$

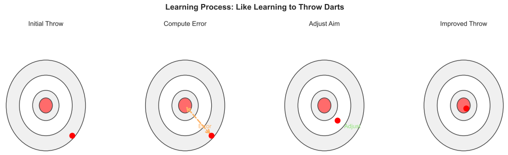
Computes all neurons at once!

How Neural Networks Learn: The Intuition

Learning = Adjusting Weights

Analogy: Learning to throw darts

1. Throw dart (forward pass)
2. See where it lands (compute error)
3. Adjust aim (update weights)
4. Repeat until bullseye!



Neural Network Version:

1. Make prediction with current weights
2. Compare to correct answer
3. Calculate error (loss)
4. Adjust weights to reduce error
5. Repeat thousands of times

The Learning Rule:

$$\text{new\_weight} = \text{old\_weight} - \alpha \cdot \text{gradient}$$

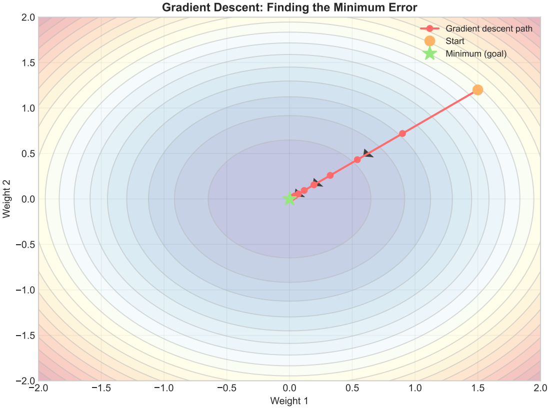
Where:

- α = learning rate (step size)
- gradient = direction of error increase
- minus = we go opposite direction

Learning is just intelligent trial and error

Gradients: The Direction to Improve

Mountain Hiking Analogy



- Goal: Reach valley (minimum error)
- Gradient: Steepest uphill direction
- We go opposite way (downhill)
- Step size: Learning rate

What is a Gradient?

Simply: **Rate of change**

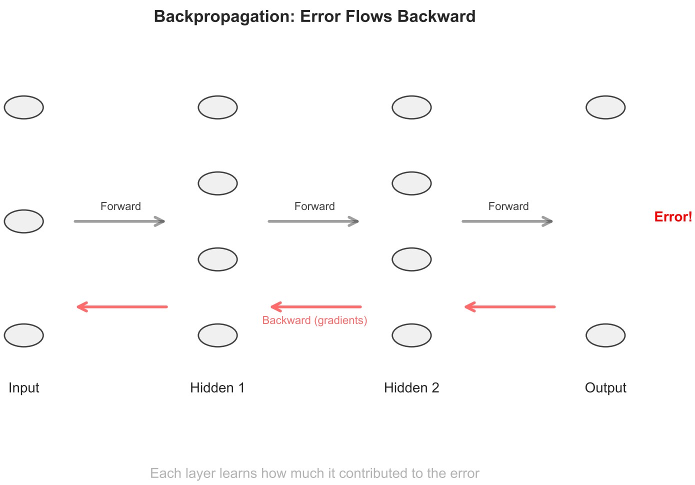
For function $f(x) = x^2$:

- Gradient $= 2x$ (derivative)
- At $x = 3$: gradient $= 6$
- Means: "increasing x increases f by $6\times$"
- So we decrease x to reduce f

In Neural Networks:

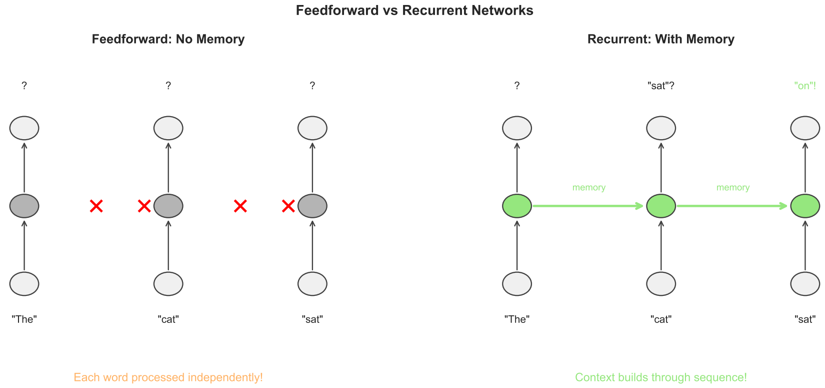
- Gradient tells how each weight affects error
- Computed via backpropagation
- Update all weights simultaneously

Backpropagation: Distributing the Blame



From Feedforward to Recurrent Networks

The Limitation of Feedforward



Feedforward: No Memory

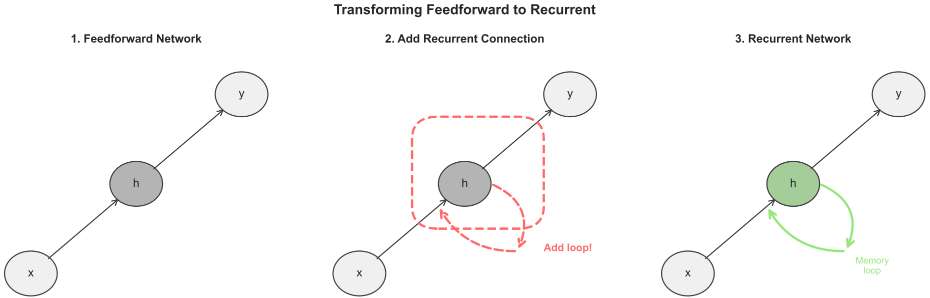
- Process one input → one output
- Forget everything, start fresh
- "The cat" → predict

Recurrent: With Memory

- Output feeds back as input
- Maintains hidden state
- "The cat" → remember "cat"

The Recurrence Idea: Adding Memory

Transforming Feedforward to Recurrent



Key Innovation:

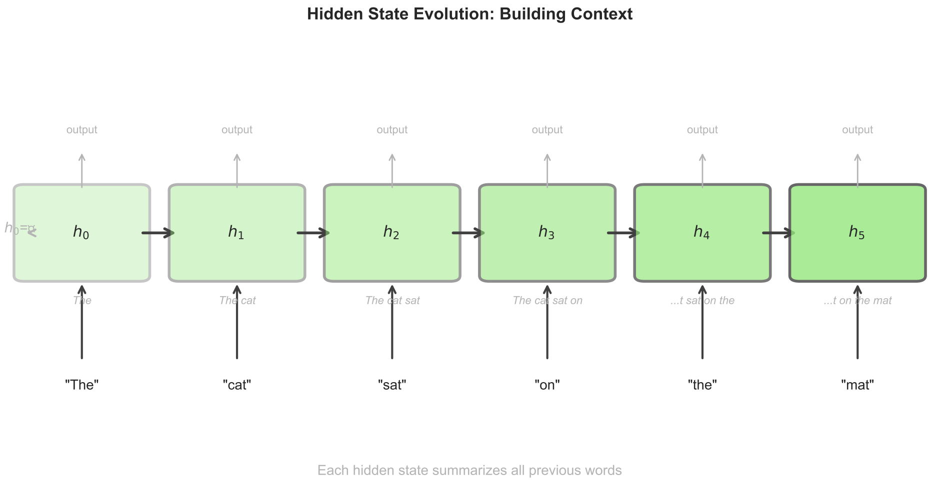
1. Take feedforward network
2. Add connection from output to input
3. Now output influences next computation
4. Creates a "memory loop"

Reading Analogy:

- Feedforward: Read each word, forget immediately
- Recurrent: Remember story while reading
- Each word updates your understanding
- Context accumulates naturally

One simple loop transforms static network to sequential processor

Hidden State: The Network's Memory



Putting It All Together: The RNN Equation

Building the equation step by step

1. Start simple:
$$\text{new\_memory} = f(\text{old\_memory}, \text{new\_input})$$
2. Add weights to control influence:
$$\text{new\_memory} = f(W_1 \cdot \text{old\_memory} + W_2 \cdot \text{new\_input})$$
3. Add bias for threshold:
$$\text{new\_memory} = f(W_1 \cdot \text{old\_memory} + W_2 \cdot \text{new\_input} + b)$$
4. Use proper notation:
$$\mathbf{h}_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b_h)$$$$\mathbf{h}_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b_h)$$
$$\mathbf{h}_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b_h)$$
$$\mathbf{h}_t = \tanh(W_{hh} \cdot h$$

Where:

- h_t = hidden state (memory) at time t
- x_t = input at time t
- W_{hh} = weights for previous memory
- W_{xh} = weights for new input
- tanh = activation function (squashing)

This one equation is the heart of RNNs!

The Importance of Order

Word Order Changes Meaning

- "Dog bites man" $\neq$ "Man bites dog"
- "Not bad" $\neq$ "Bad, not!"
- Context flows through sequence

Feedforward Limitations

- Fixed input size
- No memory between inputs
- Can't model sequences naturally
- Position information lost

Sequential Tasks in NLP

Task	Type
Language Model	Many-to-many
Translation	Seq-to-seq
Sentiment	Many-to-one
Named Entity	Many-to-many
Speech Rec.	Seq-to-seq

Most NLP is inherently sequential

Sequential processing is fundamental to understanding language

The Core Idea: Recurrence

Mathematical Definition

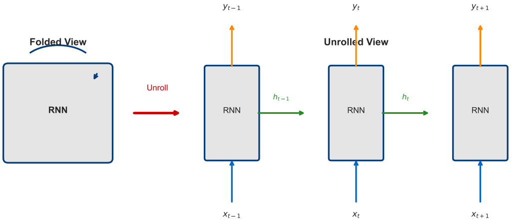
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

Where:

- h_t = hidden state at time t
- x_t = input at time t
- y_t = output at time t
- W_* = weight matrices (shared!)

Key Insight: Same weights at every timestep

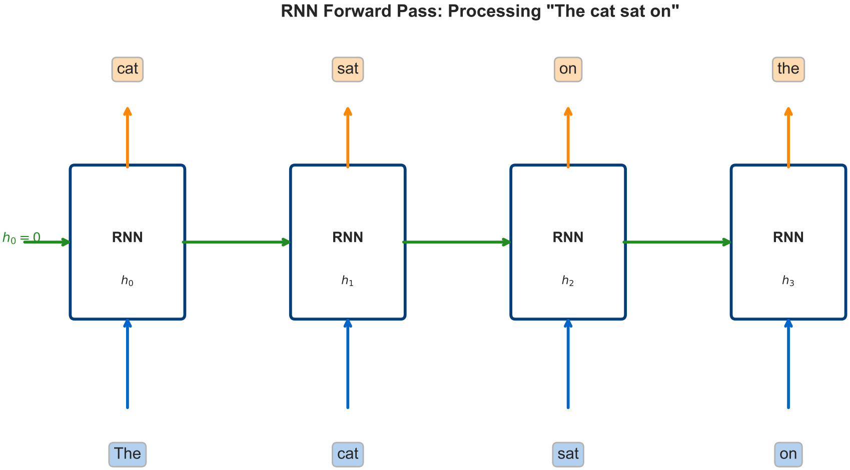


Unrolled View Shows:

- Information flows left-to-right
- Hidden state carries memory
- Parameters shared across time
- Can handle any sequence length

Remember: we just learned what each part means!

Forward Pass: Step by Step



Implementation: Simple RNN Cell

```python
import numpy as np

class RNNCell:
    def __init__(self, input_size, hidden_size):
        # Initialize weights
        self.Wxh = np.random.randn(input_size,
            hidden_size) * 0.01
        self.Whh = np.random.randn(hidden_size,
            hidden_size) * 0.01
        self.Why = np.random.randn(hidden_size,
            output_size) * 0.01
        self.bh = np.zeros((1, hidden_size))
        self.by = np.zeros((1, output_size))

    def step(self, x, h_prev):
        # Single timestep forward
        h = np.tanh(np.dot(x, self.Wxh) +
                    np.dot(h_prev, self.Whh) + self.bh)
        y = np.dot(h, self.Why) + self.by
        return y, h

    def forward(self, inputs):
        h = np.zeros((1, self.hidden_size))
        outputs = []

        for x in inputs:
```

PyTorch Equivalent:

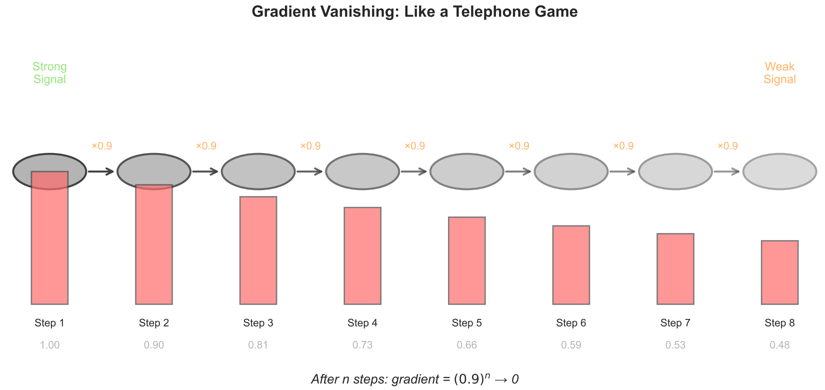
```python
import torch.nn as nn

# Built-in RNN
rnn = nn.RNN(
    input_size=100,
    hidden_size=256,
    num_layers=1,
    batch_first=True
)

# Or use LSTM/GRU
lstm = nn.LSTM(
    input_size=100,
    hidden_size=256,
    num_layers=2,
    dropout=0.2,
    bidirectional=True
)

# Forward pass
output, (hn, cn) = lstm(input_seq)
```

Modern frameworks handle the complexity

Part 2: The Vanishing Gradient Problem

Why Simple RNNs Fail

Intuition First: The Telephone Game



- Whisper message through 20 people
- Each person changes message slightly (multiply by 0.9)

The Vanishing Gradient: Mathematical View

The Problem

Gradient through time:

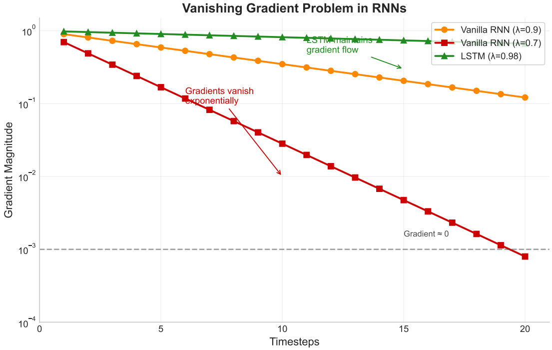
$$\frac{\partial L}{\partial h_0} = \frac{\partial L}{\partial h_T} \prod_{t=1}^{T} \frac{\partial h_t}{\partial h_{t-1}}$$

Each term: $\frac{\partial h_t}{\partial h_{t-1}} = W_h^T \cdot \text{diag}(f'(h_{t-1}))$

For tanh: $|f'(x)| \leq 1$

If $||W_h|| < 1$: gradients $\rightarrow 0$ (vanish)

If $||W_h|| > 1$: gradients $\rightarrow$ (explode)

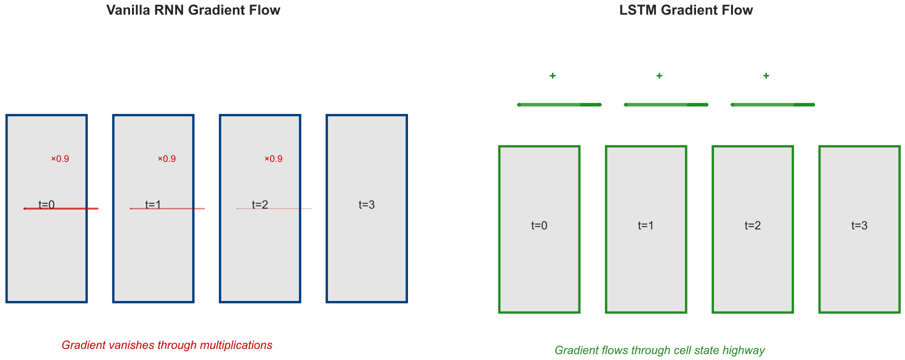


Consequences:

- Can't learn long dependencies
- Gradient 0 after 10-20 steps
- Network "forgets" early inputs
- Training becomes ineffective

The fundamental limitation that led to LSTM/GRU

Visualizing Gradient Flow



Vanilla RNN

- Exponential decay/growth
- Gradient magnitude: $O(\lambda^T)$
- Effective memory: 5-10 steps
- Cannot learn long patterns

LSTM (Next Section)

- Constant error flow
- Gradient highways
- Effective memory: 100+ steps
- Learns long dependencies

Engineering Memory

The Innovation (1997)
Hochreiter Schmidhuber's insight:

- Add a **memory cell** C_t
- Control flow with **gates**
- Create gradient highways
- Selective reading/writing

Three Gates:

1. **Forget**: What to discard
2. **Input**: What to store
3. **Output**: What to expose

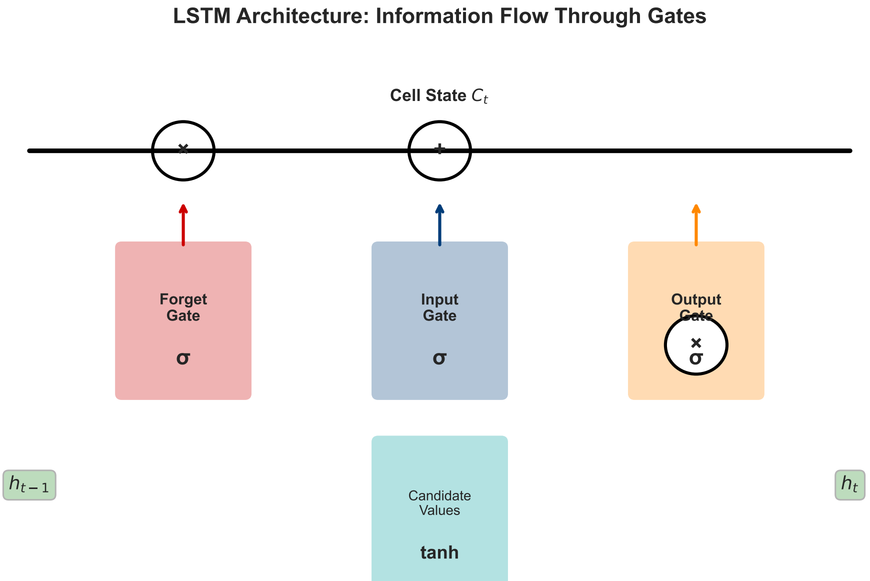
LSTM Equations

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

Gates use sigmoid (0-1) for control

The architecture that made deep sequence modeling possible

LSTM Architecture: Gate Mechanisms



RNN vs LSTM: Key Differences

Aspect	Vanilla RNN	LSTM
Parameters	$O(h^2)$	$O(4h^2)$
Memory	Short (5-10 steps)	Long (100+ steps)
Gradient flow	Multiplicative	Additive
Training speed	Fast	Slower (4x params)
Gradient problem	Severe	Largely solved
Use cases	Short sequences	Most applications

When to Use RNN:

- Very short sequences
- Real-time constraints
- Limited compute
- Simple patterns

When to Use LSTM:

- Long dependencies
- Complex patterns
- Production systems
- Default choice (pre-2017)

LSTM's complexity is justified by superior performance

GRU: Gated Recurrent Unit

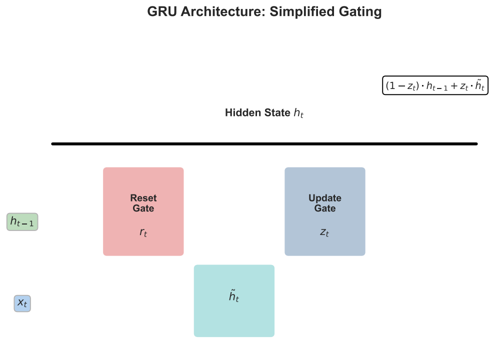
Simplification of LSTM (2014)

Cho et al. merged gates:

- Only **2 gates** instead of 3
- No separate cell state
- Fewer parameters (3x vs 4x)
- Similar performance

GRU Equations:

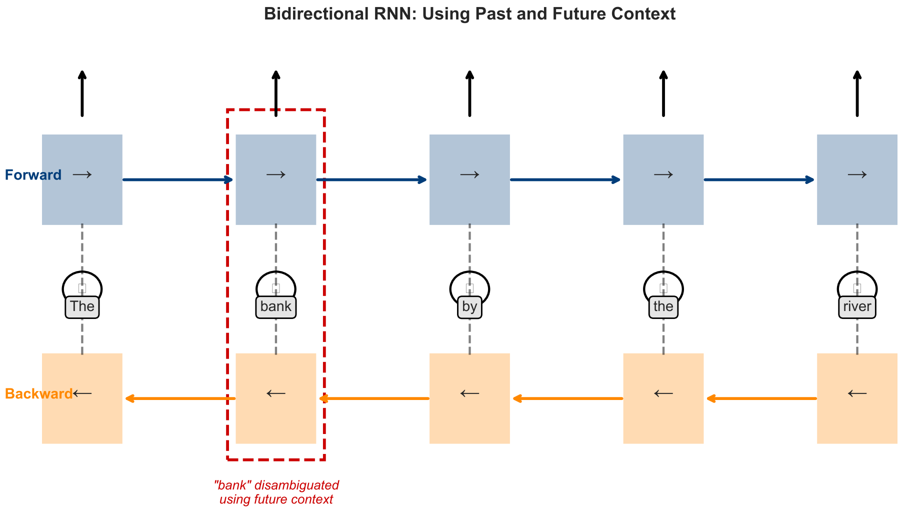
$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\
\tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\
h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t
\end{aligned}
$$



Gates:

- **Update gate** (z_t): How much to update
- **Reset gate** (r_t): How much past matters

Bidirectional RNNs: Using Future Context



Training RNNs: Practical Tips

Common Issues & Solutions

1. **Gradient Explosion**
   - Solution: Gradient clipping
   - 'torch.nn.utils.clip_grad_norm_'
   - Typical value: 1.0 - 5.0
2. **Initialization**
   - Xavier/He initialization
   - Forget gate bias = 1.0 (LSTM)
   - Helps gradient flow
3. **Overfitting**
   - Dropout (between layers)
   - Recurrent dropout (careful!)
   - Weight decay

Hyperparameters

Parameter	Typical Range
Hidden size	128 - 512
Num layers	1 - 3
Learning rate	1e-3 - 1e-2
Batch size	32 - 128
Sequence length	20 - 200
Gradient clip	1.0 - 5.0
Dropout	0.2 - 0.5

Training Strategy:

- Start with small sequences
- Gradually increase length
- Monitor gradient norms
- Use teacher forcing wisely

RNN training requires careful tuning and monitoring

Real-World Applications

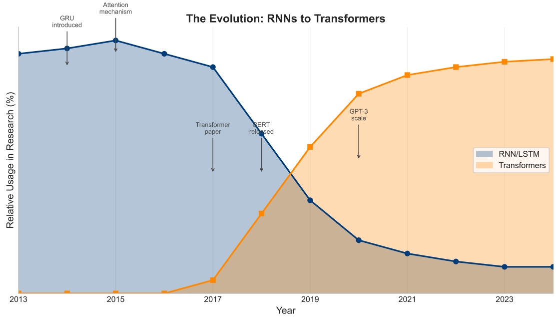
Natural Language Processing

- Language modeling (pre-2018)
- Machine translation (pre-2017)
- Speech recognition (still used)
- Named entity recognition
- Sentiment analysis

Time Series

- Stock price prediction
- Weather forecasting
- Anomaly detection
- Signal processing

Modern Context (2024)

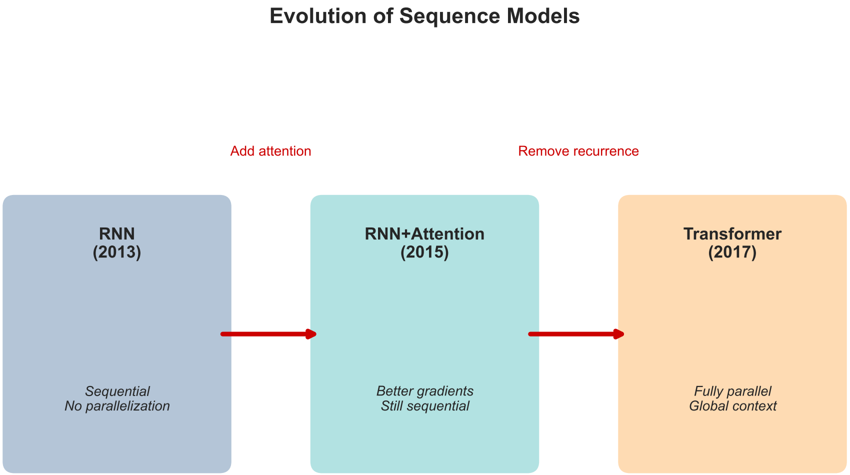


Where RNNs Still Win:

- Streaming/online processing
- Edge devices (memory constraints)
- Variable-length sequences
- Time series with clear temporal patterns

RNNs remain relevant for specific use cases

From RNNs to Transformers: Evolution



Key Takeaways

What We Learned About RNNs

Core Concepts

- **Recurrence** for sequences
- Hidden state as memory
- Parameter sharing
- Backprop through time

Challenges

- Vanishing gradients
- Sequential bottleneck
- Training difficulty
- Limited context window

Solutions

- LSTM/GRU gates
- Gradient clipping
- Bidirectional processing
- Attention (next week!)

RNNs introduced memory to neural networks - a crucial innovation

Next Week: Sequence-to-Sequence Models

How to translate, summarize, and generate with encoder-decoder architectures

References & Further Reading

Foundational Papers:

- Hochreiter & Schmidhuber (1997). "Long Short-Term Memory"
- Cho et al. (2014). "Learning Phrase Representations using RNN Encoder-Decoder" (GRU)
- Graves (2013). "Generating Sequences With RNNs"
- Karpathy (2015). "The Unreasonable Effectiveness of RNNs" (blog)

Practical Resources:

- PyTorch RNN Tutorial: `pytorch.org/tutorials/intermediate/char_rnn`
- Understanding LSTMs: `colah.github.io/posts/2015-08-Understanding-LSTMs/`
- Stanford CS224N Lecture 6: RNNs and Language Models

Code Examples:

- Week 3 Lab: 'week03_rnn_lab.ipynb'
- GitHub: Various char-RNN implementations
- Hugging Face: Modern RNN models

Appendix: Quick Math Reference

Matrix Multiplication

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

Common Derivatives

Function	Derivative
$\tanh(x)$	$1 - \tanh^2(x)$
$\sigma(x)$	$\sigma(x)(1 - \sigma(x))$
$\mathrm{ReLU}(x)$	$\begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$

Notation Guide

- $\mathbf{x}$ = vector (bold)
- W = matrix (capital)
- x_t = value at time t
- h_{t-1} = previous hidden state
- σ = sigmoid function
- $*$ = element-wise multiplication
- $\cdot$ = matrix multiplication
- $[a, b]$ = concatenation

Dimensions

- Input: $(batch, seq, features)$
- Hidden: $(batch, hidden\_size)$
- Output: $(batch, seq, classes)$