

# LSTM Primer: Next Word Prediction

## A Comprehensive Introduction to Long Short-Term Memory Networks

BSc Level - No Prerequisites Required

September 27, 2025

## From Autocomplete to Modern Language Models

### Part 1: The Problem (7 slides)

- 1. Introduction: The Autocomplete Challenge
- 2. Real-World Impact & Motivation
- 3. N-gram Baseline Models
- 4. Why N-grams Fail
- 5. The Memory Problem

### Part 2: First Attempts (5 slides)

- 6. Recurrent Neural Networks (RNN)
- 7. RNN Concrete Example
- 8. The Vanishing Gradient Problem

### Part 3: The LSTM Solution (14 slides)

- 10. LSTM Architecture Overview
- 11. Checkpoint: Understanding Architecture
- 12-14. The Three Gates (Forget, Input, Output)
- 15. Cell State: The Memory Highway
- 16. Gate Activation Patterns
- 17. Complete Forward Pass

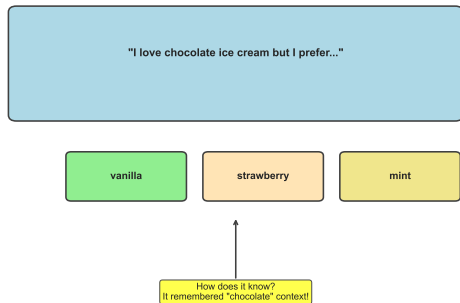
### Part 4: Training & Applications (11 slides)

- 18. Training LSTMs (BPTT)
- 19. Training Progression Visualization
- 20. Why LSTMs Work (Gradient Highway)
- 21. Checkpoint: Gradient Highway
- 22. Training Tips & Best Practices
- 23. Common Pitfalls & Debugging
- 24. LSTM Variants (GRU, Bidirectional, etc.)

# The Autocomplete Challenge

## The Problem: Predicting What Comes Next

Your Phone Predicts the Next Word



### The Challenge:

- Context can be very long
- Meaning changes with history
- Grammar rules complex
- Need to be fast (milliseconds)

### Hard Example:

*"I grew up in Paris. I went to school there. I learned to speak fluent \_\_\_"*

- Need to remember "Paris" (18 words back!)
- Ignore recent irrelevant words
- Connect city to language
- Answer: French

### What You See:

- You type on your phone

## LSTMs Changed How We Interact with Technology

### The LSTM Revolution (2015-2018):

#### Google Translate (2016):

- Switched to LSTM-based model
- Translation errors dropped 60%
- First human-quality translations
- 100+ languages supported

#### Speech Recognition:

- Siri, Alexa, Google Assistant
- Word error rate halved
- Real-time transcription
- Accent-independent

### Scientific Impact:

- 50,000+ research papers
- Breakthrough in sequence modeling
- Foundation for Transformers
- Nobel-worthy contribution

### Why It Was Revolutionary:

- **Before LSTMs:** 5-10 word memory
- **After LSTMs:** 50-100+ word memory
- **10x improvement** in context length
- Enabled practical applications

### Economic Impact:

- Translation industry transformed
- Voice interface revolution
- Billions in productivity gains

## Simple Idea: Count What Usually Comes Next

### How It Works:

#### Step 1: Look at training data

- Count every word pair (bigram)
- Count every word triple (trigram)
- Store frequency tables

#### Step 2: Make predictions

- Look at last 1-2 words
- Find in frequency table
- Pick most common next word

#### Example Training Data:

*"I love chocolate. I love pizza. I love ice cream."*

#### Bigram Counts:

- "I love" → 3 times

#### Prediction Process:

Input: "I"

- Check bigram table
- "I love" appears 3 times
- Predict: "love"

Input: "I love"

- Check trigram table
- Three options (1 count each)
- Pick randomly or use context

#### The Math:

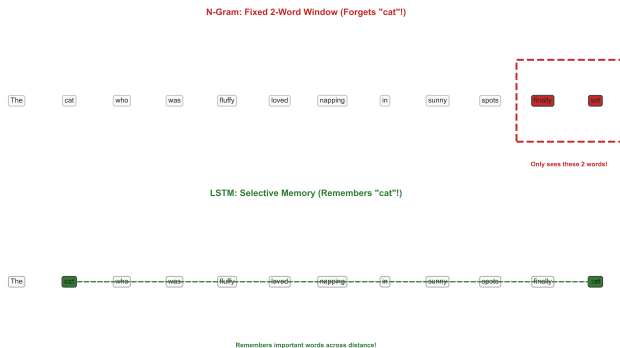
$$P(\text{word}_t \mid \text{word}_{t-1}, \text{word}_{t-2}) = \frac{\text{count}(w_{t-2}, w_{t-1}, w_t)}{\text{count}(w_{t-2}, w_{t-1})}$$

#### Why It's Popular:

- Extremely simple

# Why N-grams Fail: The Context Window Problem

## Fatal Limitation: Can Only See 1-2 Words Back



## Three Fatal Flaws:

### 1. Limited Context:

- Only 1-2 words visible
- Long-range dependencies impossible
- Miss crucial information

### 2. Combinatorial Explosion:

- 10,000 word vocabulary
- $10,000^3 = 1$  trillion trigrams
- Most never appear in training
- Sparse data problem

### 3. No Generalization:

- Can't handle novel combinations
- No understanding of meaning
- Pure memorization

## The Window Problem:

*"I grew up in Paris. I went to school there for 12 years. I learned to*

# The Memory Problem: What Should We Remember?

## Insight from Human Reading

### Imagine Reading a Novel:

*Chapter 1: "Alice was born in London in 1985. She had a happy childhood."*

*Chapter 3: "After graduating from university, Alice moved to New York."*

*Chapter 7: "Now 38 years old, Alice reflected on her life in ---"*

### What You Remember:

- Alice (main character)
- Born in London
- Moved to New York
- Currently 38 years old

### What You Forget:

**Key Insight:** Memory is **selective**

### Human Memory Strategy:

- 1 **Decide** what's important
- 2 **Keep** relevant information
- 3 **Forget** irrelevant details
- 4 **Update** as story progresses

### What We Need in AI:

#### Forget Mechanism:

- Remove outdated information
- Clear memory when topic changes
- Example: Forget "chocolate" after period

#### Input Mechanism:

- Decide what to store
- Remember important words

## Idea: Maintain a Hidden State as Memory

### The RNN Concept:

- **Hidden state**  $h_t$  = memory
- Update memory at each word
- Use memory to make predictions
- Memory flows through time

### The Math:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$$

$$y_t = \text{softmax}(W_y h_t + b_y)$$

Where:

- $h_t$  = hidden state (memory) at time  $t$
- $h_{t-1}$  = previous memory
- $x_t$  = current word embedding
- $y_t$  = prediction probabilities

### How It Processes Sequences:

Input: "I love chocolate"

#### Step 1: Process "I"

- $h_0 = [0, 0, 0, \dots]$  (initial state)
- $h_1 = \tanh(W_h h_0 + W_x [\text{embed}(\text{"I"})] + b)$
- Predict next word from  $h_1$

#### Step 2: Process "love"

- Use  $h_1$  from previous step
- $h_2 = \tanh(W_h h_1 + W_x [\text{embed}(\text{"love"})] + b)$
- Now  $h_2$  contains info about "I love"

#### Step 3: Process "chocolate"

- $h_3 = \tanh(W_h h_2 + W_x [\text{embed}(\text{"chocolate"})] + b)$
- $h_3$  should remember full sequence



## Walking Through an Actual RNN Computation

### Step 1: Process "I"

$$h_0 = [0, 0, 0]$$

#### Setup:

Input sequence: "I love"

Dimensions:

- Hidden size: 3 (toy example)
- Word embeddings: 2-dimensional

Weights (simplified):

$$W_h = \begin{bmatrix} 0.5 & -0.2 & 0.3 \\ 0.1 & 0.4 & -0.1 \\ -0.3 & 0.2 & 0.6 \end{bmatrix}$$

$$W_x = \begin{bmatrix} 0.8 & 0.3 \\ -0.2 & 0.5 \end{bmatrix}$$

$$W_x x_1 = \begin{bmatrix} 0.8 & 0.3 \\ -0.2 & 0.5 \\ 0.4 & -0.1 \end{bmatrix} \begin{bmatrix} 1.0 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.95 \\ 0.05 \\ 0.35 \end{bmatrix}$$

$$h_1 = \tanh([0, 0, 0] + [0.95, 0.05, 0.35])$$

$$h_1 = \tanh([0.95, 0.05, 0.35])$$

$$h_1 = [0.74, 0.05, 0.34]$$

### Step 2: Process "love"

$$W_h h_1 = \begin{bmatrix} 0.5 & -0.2 & 0.3 \\ 0.1 & 0.4 & -0.1 \\ -0.3 & 0.2 & 0.6 \end{bmatrix} \begin{bmatrix} 0.74 \\ 0.05 \\ 0.34 \end{bmatrix} = \begin{bmatrix} 0.46 \\ 0.04 \\ -0.01 \end{bmatrix}$$

$$[0.0, 0.0, 0.0] + [0.01, 0.01, 0.01]$$

# The Vanishing Gradient Problem

## Why RNNs Can't Learn Long-Term Dependencies

### Training Neural Networks:

#### Forward Pass:

- Input  $\rightarrow$  Hidden layers  $\rightarrow$  Output
- Compute prediction
- Calculate loss (error)

#### Backward Pass (Backpropagation):

- Compute gradient of loss w.r.t. weights
- Flow gradient backward through network
- Update weights to reduce error

### The Problem in RNNs:

Gradient at step  $t$  depends on all previous steps:

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^T \frac{\partial L_t}{\partial W_h}$$

### Why It Vanishes:

Typical values:

- $\tanh'(x) \leq 1$  (often  $\approx 0.5$ )
- If  $\|W_h\| < 1$ , products shrink
- After  $n$  steps:  $\approx 0.5^n \cdot \|W_h\|^n$

### The Numbers:

Steps	Gradient	% Remaining
1	0.90	90%
5	0.59	59%
10	0.35	35%
20	0.12	12%
50	0.005	0.5%
100	0.000027	0.0027%

### The Impact:

# Why RNNs Forget: The Paris Example

## Concrete Example of Memory Decay

### The Sentence:

*"I grew up in Paris. I went to school there. I learned to speak fluent \_\_\_"*

### What Happens in RNN:

#### Word 1-5: "I grew up in Paris"

- $h_5$  encodes this information
- "Paris" stored in hidden state
- Looks promising!

#### Word 6-15: "I went to school there"

- $h_{15}$  updates with new words
- Previous  $h_5$  gets multiplied by  $W_h$  ten times
- Information about "Paris" weakens

### The Math Behind Forgetting:

Hidden state update:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t)$$

After  $n$  steps, contribution from  $h_0$ :

$$h_n \approx (W_h)^n h_0 + \text{recent terms}$$

If largest eigenvalue of  $W_h$  is  $\lambda$ :

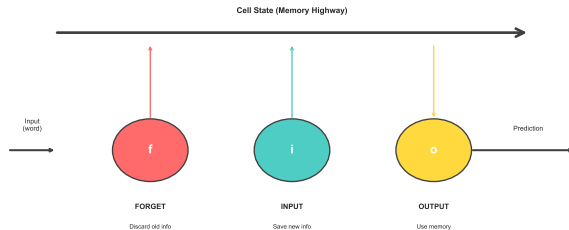
- $\lambda < 1$ : Exponential decay
- $\lambda = 0.9$  typical
- After 20 steps:  $0.9^{20} = 0.12$
- Information multiplied by 0.12

### During Training:

Gradient from step 21 to step 5:

## Long Short-Term Memory: Gated Memory Cells

LSTM Cell: Three Gates Control Memory



*Like Traffic Lights: Red (forget) • Green (input) • Yellow (output)*

### The Three Gates:

#### Forget Gate ( $f_t$ ):

- What to remove from memory
- 0 = completely forget
- 1 = keep everything
- Example: 0.9 at period

#### Input Gate ( $i_t$ ):

- What to add to memory
- 0 = ignore new information
- 1 = fully store
- Example: 0.95 on "Paris"

#### Output Gate ( $o_t$ ):

- What to reveal from memory
- 0 = hide everything

# Checkpoint: Understanding LSTM Architecture

## Test Your Understanding

### Quick Quiz:

**Question 1:** What's the key difference between RNN and LSTM?

- A) LSTM has more parameters
- B) LSTM uses addition instead of multiplication
- C) LSTM is faster to train
- D) LSTM uses different activation functions

**Question 2:** Which gate controls what enters memory?

- A) Forget gate
- B) Input gate
- C) Output gate
- D) Cell gate

### Answers:

**Answer 1:** B - LSTM uses addition for cell state

- Cell state:  $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$
- Addition creates direct gradient path
- No repeated matrix multiplication
- This is the key innovation!

**Answer 2:** B - Input gate

- Controls how much new information to store
- High value = store strongly
- Low value = ignore
- Works with candidate cell state  $\tilde{C}_t$

**Answer 3:** C - During normal continuation

- High forget gate = keep memory
- Low forget gate = clear memory

# The Forget Gate: Clearing Old Information

## Forget Gate: What Should We Remove?

### The Equation:

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

Where:

- $f_t$  = forget gate activations (0 to 1)
- $h_{t-1}$  = previous hidden state
- $x_t$  = current input word
- $\sigma$  = sigmoid function
- $[h_{t-1}, x_t]$  = concatenation

### What Sigmoid Does:

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

- Maps any number to (0,1)
- $z \rightarrow -\infty: \sigma(z) \rightarrow 0$  (forget)

### Concrete Example:

Input sequence: "I love chocolate. But I prefer"

#### At word "chocolate":

- $f_t = [0.95, 0.92, 0.88, \dots]$
- Keep most information
- Normal sentence continuation

#### At word "." (period):

- $f_t = [0.1, 0.2, 0.15, \dots]$
- Forget most of previous sentence!
- Topic might change
- New sentence starting

#### At word "But":

- $f_t = [0.3, 0.4, 0.25, \dots]$
- Contrast coming

# The Input Gate: Adding New Information

## Input Gate: What Should We Store?

### Two Equations:

#### Input Gate:

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

Controls *how much* to add (0 to 1)

#### Candidate Memory:

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

What content to potentially add (-1 to 1)

#### Combined Update:

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

- First term: What to keep from past

### Concrete Example:

Input: "I grew up in Paris"

#### At word "I":

- $i_t = [0.3, 0.2, 0.1, \dots]$  (low)
- Common word, not much info
- $\tilde{C}_t = [0.5, -0.2, 0.1, \dots]$
- Minimal storage

#### At word "Paris":

- $i_t = [0.95, 0.92, 0.88, \dots]$  (high!)
- Important location word
- $\tilde{C}_t = [0.8, 0.7, -0.3, \dots]$
- Strong encoding of "Paris"
- Will be useful later

#### At word "grew":

- $i_t = [0.5, 0.4, 0.6, \dots]$  (medium)

# The Output Gate: Revealing Memory

## Output Gate: What Should We Use Now?

### Concrete Example:

Sequence: "I grew up in Paris. I learned to speak fluent  
---"

### The Equations:

#### Output Gate:

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

Controls how much memory to expose (0 to 1)

#### Hidden State (Output):

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

- $\tanh(C_t)$ : Squash cell state to  $(-1, 1)$
- $o_t \odot$ : Filter what's revealed
- $h_t$ : Working memory for prediction

#### At word "learned":

- $o_t = [0.3, 0.4, 0.2, \dots]$  (low)
- Not predicting yet
- Don't need full memory
- Just process the word

#### At word "fluent":

- $o_t = [0.9, 0.85, 0.92, \dots]$  (high!)
- About to predict language
- Need location information
- Recall "Paris" from  $C_t$
- $h_t$  contains relevant context

### Final Prediction:

### Prediction Process:



# Cell State: The Memory Highway

## Cell State $C_t$ : The Key Innovation

### The Complete Update:

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

### What Makes This Special:

#### RNN Update:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t)$$

- Matrix multiplication by  $W_h$
- Nonlinear tanh
- Information transformed
- Gradient must flow through both

#### LSTM Cell State:

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

### The Highway Analogy:

#### RNN (Local Roads):

- Information stops at every step
- Gets transformed each time
- Slow, lossy transmission
- Limited range

#### LSTM (Highway):

- Direct express lane ( $C_t$ )
- Minimal transformation
- Fast, lossless transmission
- Long-range connectivity

### Numerical Comparison:

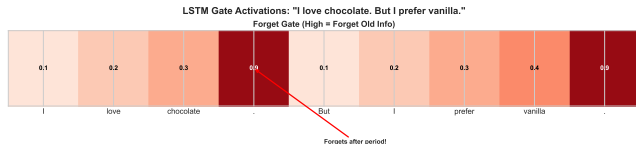
Remembering information from 50 steps back:

## Visualizing How Gates Actually Work

### Key Observations:

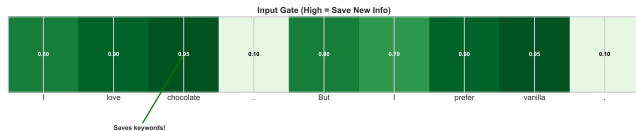
#### Forget Gate (Top):

- High throughout sentence (0.7-0.9)
- Drops dramatically at period (0.1)
- Memory cleared at sentence boundary
- Learned pattern: punctuation triggers forgetting



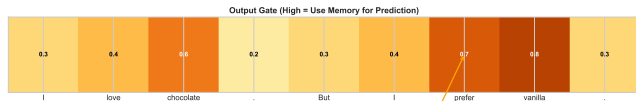
#### Input Gate (Middle):

- Peaks on content words (0.9-0.95)
- "love", "chocolate", "prefer", "vanilla"
- Low on function words (0.2-0.3)
- Selective storage of important information



#### Output Gate (Bottom):

- Rises when prediction needed (0.7-0.8)



# Complete Forward Pass: Step-by-Step

## Full LSTM Computation with Numbers

### All Equations:

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

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

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

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

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

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

### Dimensions (example):

- Vocabulary: 10,000 words
- Embedding: 100 dims
- Hidden/Cell: 256 dims
- $W_f, W_i, W_C, W_o$ :  $256 \times 356$
- $[h_{t-1}, x_t]$ : 356 dims (256+100)

### Concrete Numbers:

Input: "love" (word embedding)

#### Step 1: Forget gate

```
f_t = sigmoid([0.5, -0.2, 0.8, ...])  
      = [0.62, 0.45, 0.69, ...]
```

#### Step 2: Input gate & candidate

```
i_t = sigmoid([1.2, 0.8, -0.5, ...])  
      = [0.77, 0.69, 0.38, ...]  
C_tilde = tanh([0.6, -0.3, 0.9, ...])  
          = [0.54, -0.29, 0.72, ...]
```

#### Step 3: Update cell state

```
C_t = [0.62*0.5, 0.45*0.3, ...]  
      + [0.77*0.54, 0.69*(-0.29), ...]  
      = [0.73, 0.14, ...]
```

#### Step 4: Output & hidden

```
o_t = sigmoid([0.9, 1.1, -0.2, ...])  
      = [0.71, 0.75, 0.45, ...]  
h_t = [0.71*tanh(0.73), ...]  
      = [0.44, ...]
```

# Training LSTMs: Backpropagation Through Time

## How LSTMs Learn from Data

### Training Process:

#### 1. Forward Pass:

- Process entire sequence
- Compute predictions at each step
- Calculate loss (cross-entropy)

$$L = - \sum_{t=1}^T \log P(w_t \mid w_1, \dots, w_{t-1})$$

#### 2. Backward Pass:

- Compute gradients of loss
- Flow backward through time
- Update all weight matrices

#### 3. Weight Update:

### Why LSTM Gradient Flow Works:

#### Key Gradient:

$$\frac{\partial C_t}{\partial C_{t-1}} = f_t$$

- Simple element-wise multiplication
- No matrix multiplication
- If  $f_t \approx 1$ : Perfect transmission
- Gradient preserved across time

### Training Challenges:

- **Sequence length:** Longer = more memory
- **Batch size:** Typically 32-128 sequences
- **Learning rate:** Must be carefully tuned
- **Gradient clipping:** Prevent explosions

### Hyperparameters:

# Training Progression: Watching the LSTM Learn

## How Performance Improves Over Time

LSTM Training: Watching It Learn

Epoch 1: Random Initialization

Input: "I love chocolate"

Prediction: "xjwkq"

Loss: 8.5 (Gibberish!)

Epoch 10: Learning Letters

Input: "I love chocolate"

Prediction: "cream"

Loss: 2.1 (Better!)

Epoch 50: Understanding Context

Input: "I love chocolate"

Prediction: "ice cream"

Loss: 0.4 (Good!)

Epoch 200: Fluent Generation

Input: "I love chocolate"

Prediction: "Ice cream  
and strawberry cake"

Loss: 0.08 (Excellent!)

### Epoch 1 (Random):

- Loss: 8.5 (very high)
- Gates untrained, random values
- Predictions nonsensical
- No pattern recognition

### Epoch 10 (Bigrams):

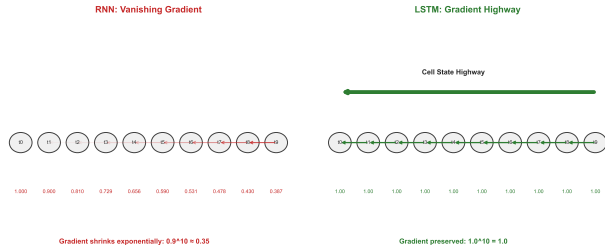
- Loss: 4.2 (improving)
- Learns immediate word pairs
- "I" → "love" pattern
- Still struggles with longer context

### Epoch 50 (Context):

- Loss: 2.1 (good)
- Gates start functioning properly
- Remembers 5-10 words back
- Better predictions

# Why LSTMs Work: The Gradient Highway

## Direct Comparison: RNN vs LSTM



**Numerical Evidence:**  
Gradient after  $n$  steps:

Steps	RNN	LSTM
10	0.35	0.90

**Why It Works:**

**Additive Updates:**

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

- Addition creates direct path
- No repeated transformations
- Information preserved

**Gradient Path:**

$$\frac{\partial C_T}{\partial C_0} = \prod_{t=1}^T f_t$$

If forget gates  $\approx 1$ :

- Product stays close to 1
- No vanishing
- Can learn long dependencies

# Checkpoint: Why the Gradient Highway Works

## Test Your Understanding of LSTM Training

### Quick Quiz:

**Question 1:** What causes vanishing gradients in RNNs?

- A) Too many parameters
- B) Repeated matrix multiplication
- C) Wrong learning rate
- D) Long sequences

**Question 2:** How does LSTM solve this?

- A) Larger matrices
- B) Addition instead of multiplication
- C) Better initialization
- D) More layers

**Question 3:** If forget gate  $f_t = 1.0$  always, what happens?

### Answers:

**Answer 1:** B - Repeated matrix multiplication

- Each step: gradient  $\times W_h$
- After  $n$  steps:  $(W_h)^n$
- If  $\|W_h\| < 1$ : Exponential decay
- This is the core problem

**Answer 2:** B - Addition for cell state

- $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$
- Gradient:  $\frac{\partial C_t}{\partial C_{t-1}} = f_t$
- Element-wise, no matrix
- Direct path preserved

**Answer 3:** B - Perfect gradient flow

- $f_t = 1 \Rightarrow$  gradient multiplied by 1

## Practical Advice for Training LSTMs

### Hyperparameter Selection:

#### Hidden Size:

- Start with 256
- Small data: 128
- Large data: 512-1024
- More = better memory, slower training

#### Number of Layers:

- Start with 2 layers
- 1 layer: Fast, may underfit
- 3-4 layers: Better, more expensive
- Diminishing returns after 4

#### Learning Rate:

- Default: 0.001 (Adam)

### Regularization:

#### Dropout:

- Apply between LSTM layers: 0.2-0.5
- NOT within LSTM cell
- Prevents overfitting
- Essential for small datasets

#### Gradient Clipping:

- Clip norm to 1.0-5.0
- Prevents exploding gradients
- Critical for stability
- PyTorch: `clip_grad_norm_`

#### Sequence Length:

- Truncate to 50-100 tokens
- Longer = better context, more memory



## What Can Go Wrong and How to Fix It

### Problem 1: Loss Not Decreasing

#### Symptoms:

- Loss stays constant
- Or decreases very slowly
- Predictions remain random

#### Possible Causes & Fixes:

- **Learning rate too low**
  - Try: 10x higher (0.01 instead of 0.001)
- **Vanishing gradients** (still possible!)
  - Check gradient norms
  - Reduce sequence length
  - Initialize forget gate bias to 1.0
- **Bug in data pipeline**

### Problem 3: Overfitting

#### Symptoms:

- Train loss low, val loss high
- Perfect on training data
- Poor on validation/test

#### Fixes:

- Increase dropout (0.3  $\rightarrow$  0.5)
- Reduce model size
- Get more training data
- Early stopping
- L2 regularization

### Problem 4: Predictions Uniform

#### Symptoms:

- Model predicts same word always

## Improvements and Modifications

### 1. GRU (Gated Recurrent Unit):

Simpler architecture:

- Only 2 gates (vs 3 in LSTM)
- No separate cell state
- Faster to train
- Often comparable performance

Equations:

$$z_t = \sigma(W_z[h_{t-1}, x_t]) \quad (\text{update})$$

$$r_t = \sigma(W_r[h_{t-1}, x_t]) \quad (\text{reset})$$

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

### 2. Bidirectional LSTM:

- Two LSTMs: forward & backward

### 3. Stacked/Deep LSTMs:

- Multiple LSTM layers
- Layer 1 output  $\rightarrow$  Layer 2 input
- Hierarchical representations
- 2-4 layers typical

### 4. Attention Mechanism:

- Weight each hidden state
- Focus on relevant parts
- Improved context use
- Led to Transformers

### 5. Peephole Connections:

- Gates see cell state directly
- $f_t = \sigma(W_f[h_{t-1}, x_t, C_{t-1}])$
- Slightly better timing
- Rarely used in practice

# Model Comparison: When to Use What

## Decision Framework for Sequence Modeling

Comparison: N-gram vs RNN vs LSTM

Feature	N-gram	RNN	LSTM
Memory Type	Fixed window	Fading	Selective
Long Context	❌	❌	✅
Parameters	Few	Moderate	Many
Training Speed	Fast	Medium	Slow
Vanishing Gradient	N/A	Yes ❌	Solved ✅
Best For	Short (2-3 words)	Medium (10 words)	Long (50+ words)
Example	"I love..."	"The cat sat..."	"The cat, who was...finally..."

### Practical Recommendations:

#### Use LSTM when:

- Time series prediction
- Sequential generation (char-by-char)
- Mobile/edge deployment
- Limited compute budget
- Variable-length sequences

#### Use GRU when:

- Faster training needed
- Similar to LSTM use cases
- Slightly less memory
- Often first try

#### Use Transformer when:

- Long sequences (>100 tokens)

### Decision Tree:

## The Idea That Changed Everything

### The Problem with LSTMs:

In sequence-to-sequence tasks:

- Encoder compresses entire source into one vector
- Decoder must use this single vector
- Bottleneck for long sequences
- Information loss

### The Attention Solution:

Instead of single vector:

- Keep all encoder hidden states
- Decoder “attends” to relevant parts
- Weighted combination at each step
- Dynamic focus

### Example: Translation

English: “The cat sat on the mat”

French: “Le chat \_\_\_”

When generating “assis” (sat):

- High attention on “sat” (0.7)
- Medium attention on “cat” (0.2)
- Low attention on “the”, “on” (0.05 each)
- Almost zero on “mat” (0.01)

### Why It Matters:

- Solves long sequence problem
- Interpretable (visualize attention)
- State-of-the-art 2015-2017
- Direct path to Transformers

### Transformers (2017):

## Where LSTMs Made Impact

### 1. Natural Language Processing:

- **Machine Translation:** Google Translate (2016-2019)
- **Text Generation:** Story writing, code completion
- **Sentiment Analysis:** Product reviews, social media
- **Named Entity Recognition:** Extract names, places
- **Question Answering:** Early chatbots

### 2. Speech & Audio:

- **Speech Recognition:** Siri, Alexa, Google Assistant
- **Speech Synthesis:** Text-to-speech systems
- **Music Generation:** Compose melodies
- **Audio Classification:** Sound event detection

### 4. Time Series:

- **Stock Prediction:** Financial forecasting
- **Weather Forecasting:** Temperature, rain
- **Energy Consumption:** Load prediction
- **Traffic Prediction:** Route optimization
- **Anomaly Detection:** Equipment monitoring

### 5. Healthcare:

- **Patient Monitoring:** ICU time series
- **Disease Progression:** Model trajectories
- **Drug Discovery:** Molecular sequences
- **ECG Analysis:** Heart rhythm detection

### Impact Statistics:

- **Google Translate:** 60% error reduction (2016)
- **Speech recognition:** Word error rate halved
- **Citations:** 50 000+ research papers

## Building LSTM Models in Practice

### Model Definition:

```
import torch
import torch.nn as nn

class LSTMModel(nn.Module):
    def __init__(self, vocab_size,
                  embed_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(
            vocab_size, embed_dim)
        self.lstm = nn.LSTM(
            embed_dim, hidden_dim,
            num_layers=2, dropout=0.3,
            batch_first=True)
        self.fc = nn.Linear(
            hidden_dim, vocab_size)

    def forward(self, x):
        embed = self.embedding(x)
        lstm_out, (h_n, c_n) = self.lstm(embed)
        output = self.fc(lstm_out)
        return output

model = LSTMModel(10000, 100, 256)
```

### Training Loop:

```
optimizer = torch.optim.Adam(
    model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

for epoch in range(num_epochs):
    for batch in dataloader:
        input_seq, target_seq = batch

        # Forward
        output = model(input_seq)
        loss = criterion(
            output.view(-1, vocab_size),
            target_seq.view(-1))

        # Backward
        optimizer.zero_grad()
        loss.backward()

        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(
            model.parameters(), 1.0)

        optimizer.step()
```

### Key Parameters:

- num\_layers=2: Stack 2 LSTMs
- dropout=0.3: Regularization

## What We've Learned

### The Problem:

- Next word prediction needs long context
- N-grams: Fixed window (1-2 words)
- RNNs: Vanishing gradients (5-10 words)
- Need selective, long-term memory

### The LSTM Solution:

- Three gates control information flow
- Cell state = memory highway
- Additive updates preserve gradients
- Can remember 50-100+ steps

### How It Works:

- **Forget gate:** Remove old info
- **Input gate:** Add new info

### Why It Matters:

- Breakthrough in NLP (2015-2018)
- Enabled modern language models
- Foundation for Transformers
- Still used for time series

### Key Innovations:

- Separate memory path (cell state)
- Gating mechanisms (learned control)
- Gradient highway (no vanishing)
- Modular design (stackable)

### Practical Tips:

- Start with 2 layers, 256 hidden units
- Always use gradient clipping
- Dropout between layers (0.3)
- Adam optimizer ( $\text{lr}=0.001$ )

## Continue Your Learning Journey

### Essential Reading:

#### Blog Posts:

- **Colah's Blog:** "Understanding LSTM Networks"
  - Best visual explanation
  - Step-by-step diagrams
  - colah.github.io
- **Karpathy's Blog:** "The Unreasonable Effectiveness of RNNs"
  - Character-level generation
  - Intuitive examples
  - karpathy.github.io

#### Papers:

- Hochreiter & Schmidhuber (1997): Original LSTM

### Practical Resources:

#### Code Tutorials:

- PyTorch LSTM tutorial (official docs)
- TensorFlow RNN guide
- Keras LSTM examples

#### Datasets to Practice:

- Penn Treebank (language modeling)
- IMDB reviews (sentiment analysis)
- Time series: stock prices, weather
- Text8 (character-level modeling)

#### Next Steps:

- 1 Implement LSTM from scratch (educational)
- 2 Train on small dataset (Penn Treebank)
- 3 Visualize gate activations