LSTM Primer: Next Word Prediction

A Comprehensive Introduction to Long Short-Term Memory Networks

BSc Level - No Prerequisites Required

September 27, 2025

Course Overview

From Autocomplete to Modern Language Models

Part 1: The Problem

- 1. Introduction: The Autocomplete Challenge
- 2. N-gram Baseline Models
- 3. Why N-grams Fail
- 4. The Memory Problem

Part 2: First Attempts

- 5. Recurrent Neural Networks (RNN)
- 6. How RNNs Process Sequences
- 7. The Vanishing Gradient Problem
- 8. Why RNNs Forget

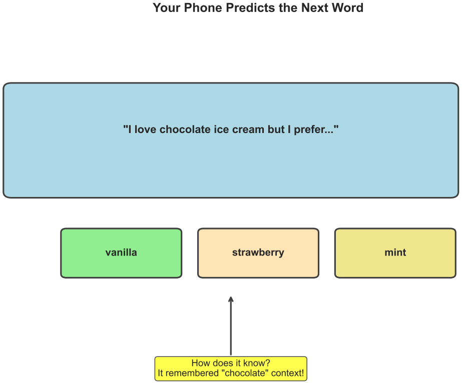
Part 3: The LSTM Solution

- 9. LSTM Architecture Overview
- 10-12. The Three Gates (Forget, Input, Output)
- 13. Cell State: The Memory Highway
- 14. Complete Forward Pass Example

Part 4: Training & Applications

- 15. Training LSTMs
- 16. Why LSTMs Work
- 17. Variants & Extensions
- 18. Real-World Applications
- 19. Implementation & Summary

The Problem: Predicting What Comes Next



What You See:

- You type on your phone

The Challenge:

- Context can be very long
- Meaning changes with history
- Grammar rules are 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

N-gram Models: The Baseline

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" $\rightarrow$ 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:

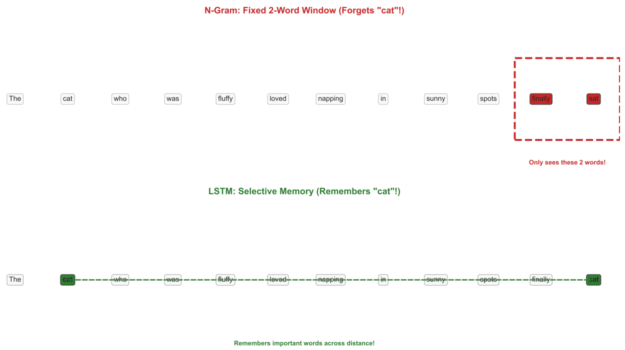
$$P(\text{word}_t \mid \text{word}_{t-1}, \text{word}_{t-2}) = \frac{\text{count}(w_{t-2}, w_{t-1}, w_t)}{\text{count}(w_{t-2}, w_{t-1})}$$

Why It's Popular:

- Extremely simple

Why N-grams Fail: The Context Window Problem

Fatal Limitation: Can Only See 1-2 Words Back



The Window Problem:

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

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 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

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 Gate:

- Remove outdated information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

Input Gate:

Recurrent Neural Networks (RNN): First Attempt

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"

- $h_0 = [0, 0, 0, \ldots]$ (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

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 ≈ 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: A Concrete Example

The Paris Example Revisited

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 the Forgetting:

Hidden state update:

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

After n steps, contribution from h_0:

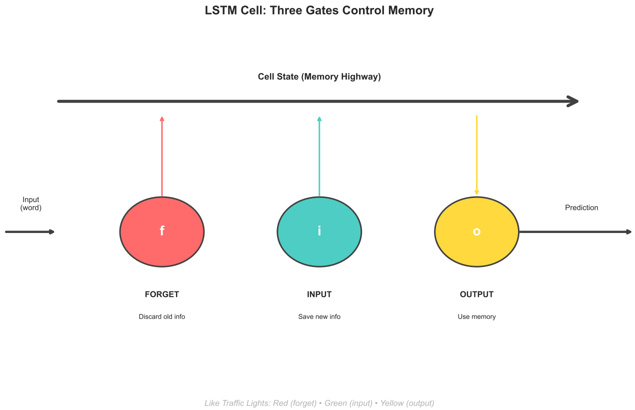
$$h_n \approx (W_h)^n h_0 + \text{recent terms}$$

If largest eigenvalue of W_h is λ:

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

During Training (Backpropagation):
Gradient from step 21 to step 5:

Long Short-Term Memory: Gated Memory Cells



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

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
- σ = sigmoid function
- $[h_{t-1}, x_t]$ = concatenation

What Sigmoid Does:

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

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

Concrete Example:

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

At word "chocolate":

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

At word "." (period):

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

At word "But":

- $f_t = [0.3, 0.4, 0.25, \ldots]$
- Contrast coming
- Remember some context

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:

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

Concrete Example:
Input: “I grew up in Paris”

At word “I”:

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

At word “Paris”:

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

At word “grew”:

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

The Output Gate: Revealing Memory

Output Gate: What Should We Use Now?

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

Final Prediction:

Concrete Example:
Sequence: "I grew up in Paris. I learned to speak fluent ___"

At word "learned":

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

At word "fluent":

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

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:

Complete Forward Pass: Step-by-Step Example

Full LSTM Computation

All Four Equations:

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

Dimensions (example):

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

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

2. Backward Pass:

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

3. Weight Update:

∂L

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:

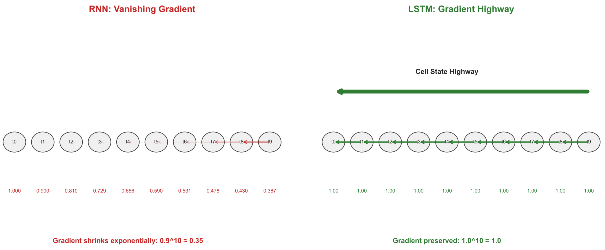
- **Sequence length**: Longer = more memory
- **Batch size**: Typically 32-128 sequences
- **Learning rate**: Must be carefully tuned
- **Gradient clipping**: Prevent explosions

Hyperparameters:

- Hidden size: 128-1024

Why LSTMs Work: 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 ≈ 1:

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

LSTM Variants and Extensions

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:

3. Stacked/Deep LSTMs:

- Multiple LSTM layers
- Layer 1 output → 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

Real-World Applications

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

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

Impact Statistics (2015-2020):

- Google Translate: 2016 LSTM reduced errors 60%
- Speech recognition: Word error rate halved
- Citations: 50,000+ research papers

Building LSTM Models in PyTorch

Basic PyTorch Implementation:

```
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 pass
        output = model(input_seq)
        loss = criterion(
            output.view(-1, vocab_size),
            target_seq.view(-1))

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(
            model.parameters(), 1.0)
        optimizer.step()
```

Key Hyperparameters:

- **Hidden dim**: 128-512
- **Layers**: 2-3
- **Dropout**: 0.2-0.5

Summary & Key Takeaways

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

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)

Next Steps:

- Practice implementation
- Study Transformers (2017+)
- Learn attention mechanisms