Recurrent Neural Networks

Week 3: Teaching Computers to Remember

NLP Course 2025

Professional Template Edition

September 29, 2025

Week 3: Journey Through Sequential Learning

1. Part 1: Motivation & Fundamentals
2. Part 2: Architecture & Mathematics
3. Part 3: The Vanishing Gradient Problem
4. Part 4: Implementation & Applications

Learning Path: From feedforward to recurrent networks. Master how RNNs maintain memory through time, solve the vanishing gradient problem with LSTMs, and build powerful sequence models that power modern applications.

Part 1: Motivation & Fundamentals

Why Sequential Processing Matters

Why Your Voice Assistant Sometimes Fails

The Conversation:

You: "Set a timer for 10 minutes"
Alexa: "Timer set for 10 minutes"
You: "Actually, make it 15"
Alexa: "I'm not sure what you want me to make"

The Problem: Word embeddings alone don't remember context!

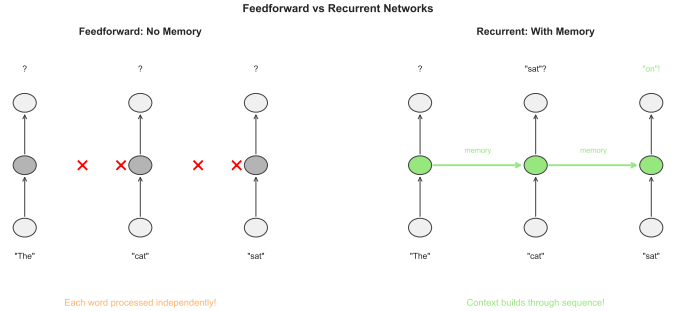
What Went Wrong?

- "it" refers to "timer" from 2 turns ago
- Static embeddings can't track conversation
- No memory of previous utterances
- Each word processed independently

The Solution:
Networks that maintain memory of past inputs - **Recurrent Neural Networks**

Key Insight: Language understanding requires memory - words depend on what came before

Same Words, Different Order, Different Meaning



Example 1:

- "Dog bites man" → Not news
- "Man bites dog" → Front page!

Word Embeddings See: {dog, bites, man} - Same vectors!

Example 2:

- "not bad" → Positive
- "bad, not good" → Negative

Order Creates Meaning: Sequential processing essential

Fundamental Challenge: Language is inherently sequential - we need models that process it that way

How Humans Read: The Inspiration for RNNs

Reading "The movie was really..." step by step:

Human Process:

1. Read "The" → remember it
2. Read "movie" → remember "The movie"
3. Read "was" → remember "The movie was"
4. Read "really" → expect adjective next

We naturally maintain a running memory!

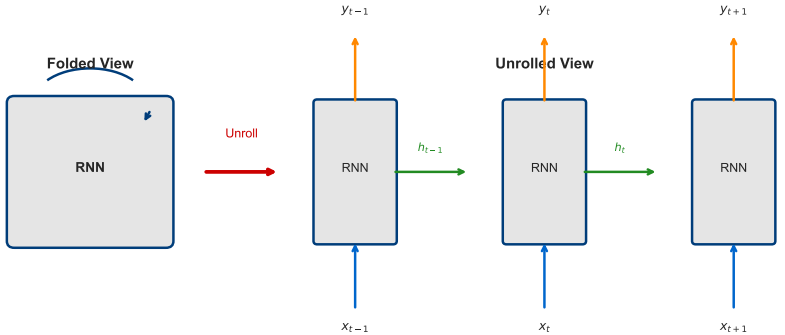
RNN Process:

1. Process one word at a time
2. Maintain "hidden state" (memory)
3. Update memory with each word
4. Use memory to predict next

Hidden state = What the network remembers

Core Concept: RNNs process sequences like humans do - one step at a time with memory

Visualizing RNNs: Unfolding Through Time



Key Insights:

- Same weights W used at every step
- Can handle any sequence length
- Memory flows forward in time
- Each step sees current input + previous memory
- Output depends on entire history
- Training updates all time steps

Unfolding Insight: One network becomes many - but all share the same parameters

Interactive Exercise: Trace the Hidden State

Task: Manually compute RNN hidden states for "cat sat"

Given: $h_t = \tanh(W_h h_{t-1} + W_x x_t)$ where $W_h = 0.5$, $W_x = 0.8$

Step 1: Process "cat"

- $x_1 = 1.0$ (embedding for "cat")
- $h_0 = 0$ (initial state)
- $h_1 = \tanh(0.5 \cdot 0 + 0.8 \cdot 1.0)$
- $h_1 = \tanh(0.8) = ?$

Your answer: $h_1 =$ _____

Step 2: Process "sat"

- $x_2 = 0.5$ (embedding for "sat")
- $h_1 = 0.664$ (from step 1)
- $h_2 = \tanh(0.5 \cdot 0.664 + 0.8 \cdot 0.5)$
- $h_2 = \tanh(0.732) = ?$

Your answer: $h_2 =$ _____

Learning by Doing: Computing by hand builds intuition for how memory accumulates

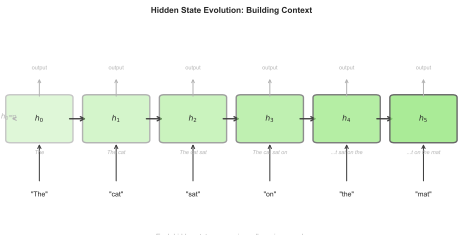
Part 1 Summary: Why We Need RNNs

What We Learned:

- Language is fundamentally sequential
- Order changes meaning dramatically
- Static embeddings lack memory
- Context spans multiple time steps
- Humans read with running memory

RNN Solution:

- Process sequences step-by-step
- Maintain hidden state (memory)
- Share weights across time
- Handle variable lengths



Key Takeaway:
RNNs = Neural Networks + Memory

Next: Dive deep into the mathematics and architecture of RNNs

Part 2: Architecture & Mathematics

The Elegant Mathematics of Memory

RNN Mathematics: Just Two Simple Equations

The Complete RNN:

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

Components:

- x_t: Input at time t (word embedding)
- h_t: Hidden state at time t (memory)
- y_t: Output at time t (predictions)
- W_{hh}: Hidden-to-hidden weights
- W_{xh}: Input-to-hidden weights
- W_{hy}: Hidden-to-output weights

In Plain English:

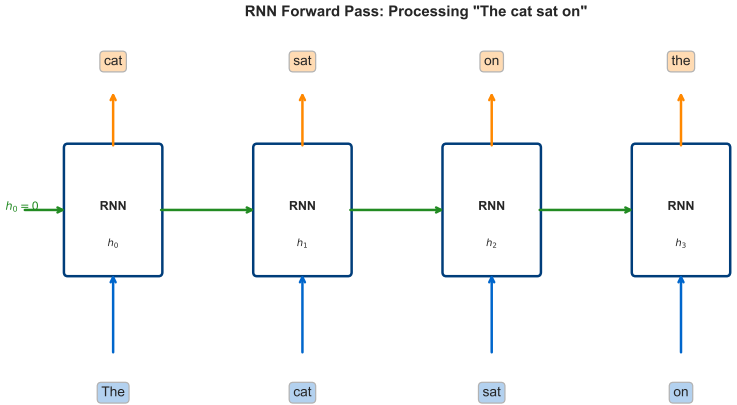
- **Line 1:** Combine old memory with new input
- **tanh:** Squash to [-1, 1] range
- **Line 2:** Generate output from memory
- **Shared** W: Same transformation always

New memory = f(old memory, new input)

Mathematical Elegance: Two equations capture the essence of sequential processing

Forward Pass: Concrete Example with Real Numbers

Processing "I love NLP" step by step:



Time $t = 1$ ("I"):

Time $t = 2$ ("love"):

Time $t = 3$ ("NLP"):

Weight Sharing: The Power and the Problem

Why Share Weights?

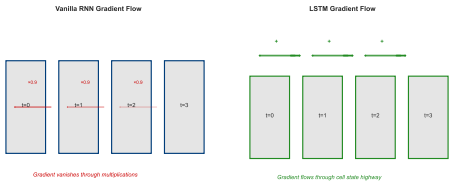
- **Generalization:** Learn patterns once
- **Efficiency:** Fewer parameters
- **Flexibility:** Any sequence length
- **Consistency:** Same processing always

The Dark Side:

- **Gradient flow:** Through same W repeatedly
- **Vanishing:** Gradients $\to 0$
- **Exploding:** Gradients $\to \infty$
- **Long-term memory:** Difficult

Example Benefits:

- "not good" at position 1 or 10
- Learn "ing" ending anywhere
- Transfer learning across positions



Trade-off: Weight sharing enables generalization but creates gradient challenges

Complete RNN Implementation in PyTorch

```python
import torch
import torch.nn as nn

class SimpleRNN(nn.Module):
    def __init__(self, input_size,
                 hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size

        # Define layers
        self.i2h = nn.Linear(
            input_size + hidden_size,
            hidden_size
        )
        self.h2o = nn.Linear(
            hidden_size, output_size
        )

    def forward(self, input, hidden):
        # Combine input and hidden
        combined = torch.cat(
            (input, hidden), 1
        )

        # Update hidden state
        hidden = torch.tanh(
            self.i2h(combined)
        )

        # Generate output
```

Usage Example:

```python
# Initialize
rnn = SimpleRNN(
    input_size=100,
    hidden_size=256,
    output_size=100
)

# Process sequence
hidden = torch.zeros(1, 256)
for word in sentence:
    output, hidden = rnn(
        word, hidden
    )
```

Key Points:

- Hidden size typically 128-512
- Input/output = vocabulary size
- Hidden state persists
- Same weights all steps

Hands-On: Debug This RNN Code

Find and fix the 3 bugs in this RNN implementation:

```
Bug 1: What's wrong here? self.embedding = nn.Embedding(
           vocab_size, vocab_size)
Bug 2: What's missing? self.rnn = nn.RNN(
           input_size = 128, hidden_size = 256)
def forward(self, x):    Bug 3: What's the issue? embedded =
           self.embedding(x) output = self.rnn(embedded)
           return output
```

Your Fixes:

Bug 1: _____

Bug 2: _____

Bug 3: _____

Hints:

- Check dimensions
- RNN needs 2 outputs
- Embedding size $\neq$ vocab size

Debugging Practice: Common RNN mistakes - dimension mismatches and missing components

Checkpoint Quiz: Test Your Understanding

Questions:

Q1: Why do RNNs use tanh activation?

a) To make training faster
b) To keep values in [-1, 1]
c) To add non-linearity
d) Both b and c

Q2: What makes RNNs different from feedforward networks?

a) More parameters
b) Recurrent connections
c) Deeper architecture
d) Faster training

Q3: Hidden state size 256 means:

a) 256 words vocabulary
b) 256-dimensional memory

Answers:

A1: d) Both b and c

- tanh bounds outputs
- Adds essential non-linearity
- Prevents value explosion

A2: b) Recurrent connections

- Hidden state feeds back
- Creates memory mechanism
- Key architectural difference

A3: b) 256-dimensional memory

- Size of h_t vector
- Network's memory capacity
- Independent of vocab/length

Common Misconceptions About RNNs

Misconception 1: "RNNs can remember everything"
Reality:

- Memory degrades over time
- Typically 5-10 steps effective
- Vanishing gradients limit range

Misconception 2: "Bigger hidden state = better"
Reality:

- Overfitting risk increases
- Training becomes harder
- Sweet spot: 128-512

Misconception 3: "RNNs process in parallel"
Reality:

- Strictly sequential
- Can't parallelize time steps
- This limits speed

Misconception 4: "RNNs are obsolete"
Reality:

- Still best for some tasks
- Efficient for short sequences
- Active research area

Clarity Check: Addressing these misconceptions prevents future confusion

Part 2 Summary: RNN Architecture Mastered

Key Equations to Remember:

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

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

What We Covered:

- Mathematical foundation
- Forward pass computation
- Weight sharing benefits/problems
- PyTorch implementation
- Common bugs and fixes

Key Insights:

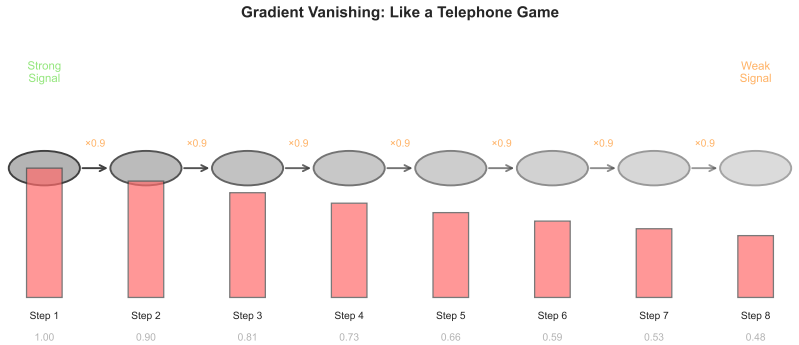
- Simple equations, complex behavior
- Memory through recurrence
- Same weights everywhere
- Sequential processing only
- Foundation for LSTMs/GRUs

Next: The critical problem that almost killed RNNs - vanishing gradients

Part 3: The Vanishing Gradient Problem

Why RNNs Forget and How LSTMs Remember

The Telephone Game: Understanding Gradient Degradation

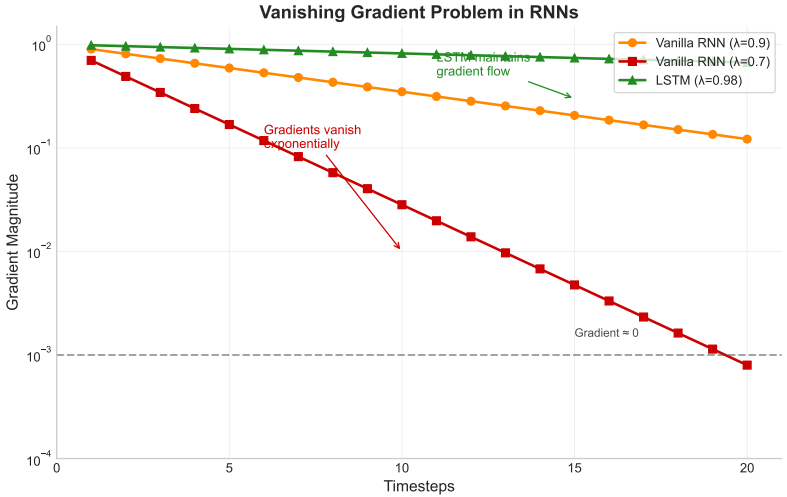


The Game:

- Person 1: "Buy milk and bread"
- Person 5: "Buy milk and ???"

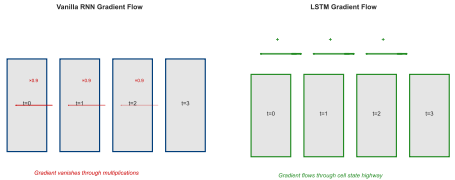
Mathematical Reality:
Gradient after t steps:

Backpropagation Through Time (BPTT):



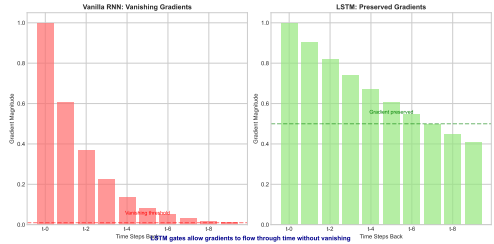
Vanishing vs Exploding: Two Sides of the Same Coin

Vanishing Gradients:



- Weights < 1: Signal dies
- Can't learn long dependencies
- Network "forgets" early inputs
- Most common problem

Exploding Gradients:



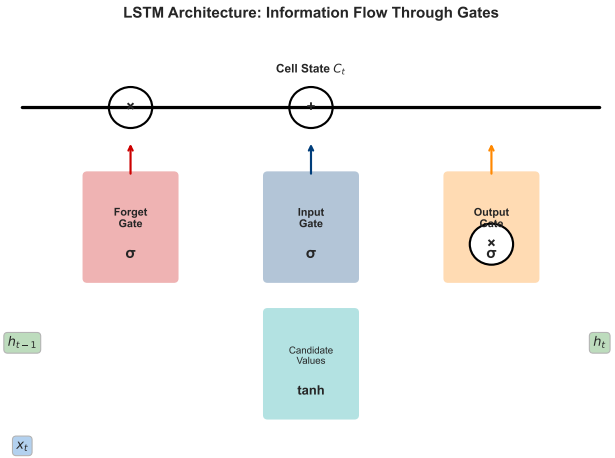
- Weights > 1: Signal explodes
- NaN values crash training
- Gradient clipping helps
- Easier to detect/fix

Critical Issue: Standard RNNs can effectively use only 5-10 steps of context

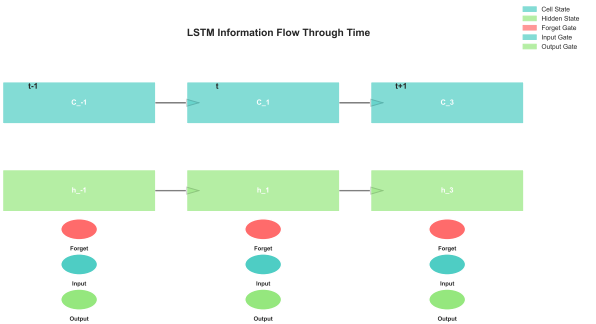
The Dilemma: Too small weights = vanishing, too large = exploding, just right = impossible

LSTM: The Elegant Solution to Memory

Long Short-Term Memory (LSTM) - Hochreiter & Schmidhuber, 1997



LSTM Architecture: Three Gates to Rule Them All



Gate Equations:

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

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (2)$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (3)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (4)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \quad (5)$$

What Each Gate Does:

Forget Gate (f_t): "Forget that we saw 'not' "

Input Gate (i_t): "Store 'important' strongly"

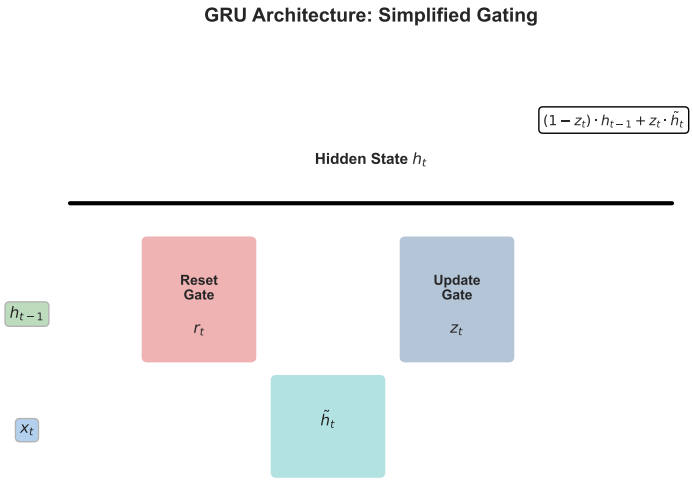
Output Gate (o_t): "Output positive sentiment"

Key: Line 5

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

Addition creates gradient highway!

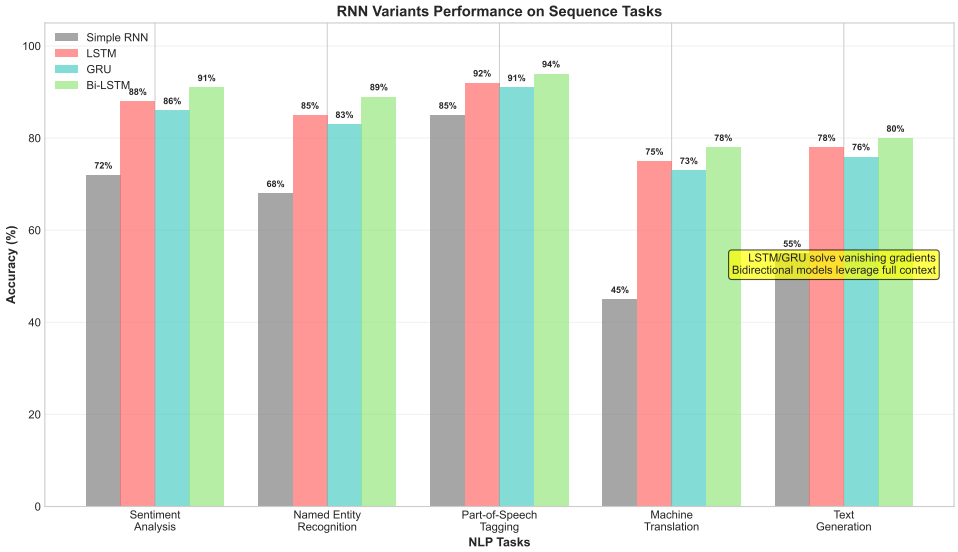
GRU: LSTM's Streamlined Sibling



GRU (Gated Recurrent Unit):

LSTM vs GRU:

RNN vs LSTM vs GRU: Head-to-Head Comparison



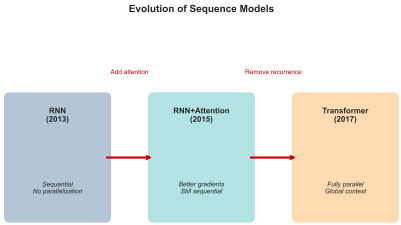
Part 3 Summary: From Problem to Solution

The Problem:

- Gradients vanish exponentially
- Can't learn long dependencies
- Telephone game effect
- 5-10 step practical limit
- Made RNNs nearly useless

The Solution:

- LSTM: Gated architecture
- Addition instead of multiplication
- Gradient highway preserved
- 100+ step dependencies
- Revolutionized sequence modeling



Historical Impact:
LSTMs enabled modern NLP (1997-2017 dominance)

Next: Implement these concepts in code and explore real applications

Part 4: Implementation & Applications

From Theory to Practice

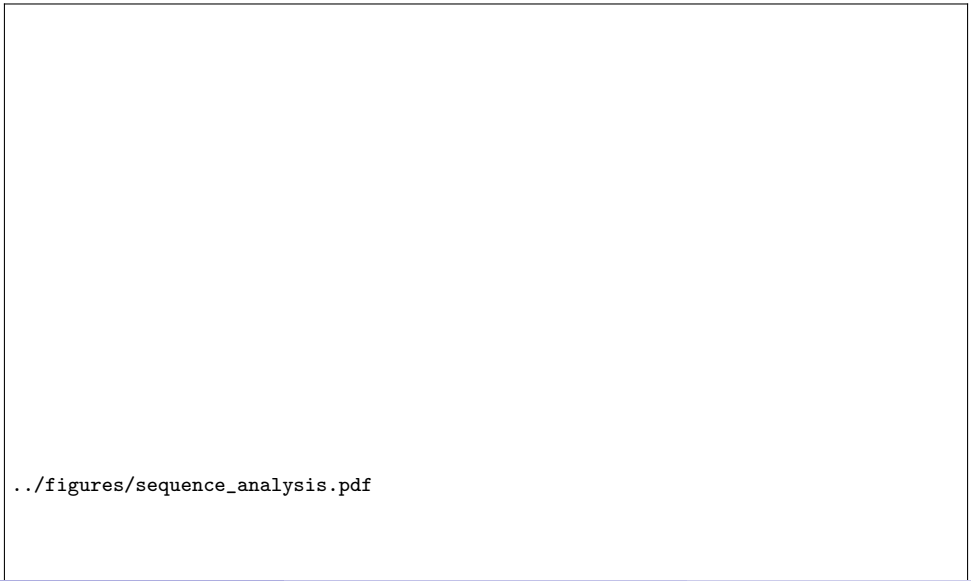
Complete LSTM Implementation in PyTorch

```python
import torch
import torch.nn as nn

class TextGenerator(nn.Module):
    def __init__(self, vocab_size,
                 embed_size=128,
                 hidden_size=256,
                 num_layers=2):
        super().__init__()

        # Embedding layer
        self.embed = nn.Embedding(
            vocab_size, embed_size
        )

        # LSTM layers
        self.lstm = nn.LSTM(
            embed_size, hidden_size,
            num_layers, batch_first=True,
            dropout=0.2
        )

        # Output layer
        self.linear = nn.Linear(
            hidden_size, vocab_size
        )

    def forward(self, x, hidden=None):
        # Embed input
        embeds = self.embed(x)
```

Training Loop:

```python
model = TextGenerator(
    vocab_size=10000
)
optimizer = torch.optim.Adam(
    model.parameters()
)
criterion = nn.CrossEntropyLoss()

for epoch in range(10):
    hidden = None
    for batch in dataloader:
        # Forward pass
        output, hidden = model(
            batch, hidden
        )

        # Detach hidden
        hidden = tuple(
            h.detach()
            for h in hidden
        )

        # Loss & backward
        loss = criterion(
            output, targets
        )
        loss.backward()

        # Gradient clipping
```

Training Progression: Watch the Network Learn



RNNs in Production: 2024 Real-World Applications

Still Dominant:

- **Speech Recognition:**
  - Google's RNN-T on phones
  - Apple's on-device Siri
  - 450MB models, real-time
- **Time Series:**
  - Stock price prediction
  - Weather forecasting
  - Energy demand modeling
- **Music Generation:**
  - MuseNet's LSTM backbone
  - Real-time synthesis
  - Latency-critical tasks

Hybrid Approaches:

- **Transformer + LSTM:**
  - Global + local context
  - Best of both worlds
  - State-of-the-art results
- **Efficient Models:**
  - Mobile deployment
  - Edge computing
  - Battery-powered devices
- **Streaming Applications:**
  - Live transcription
  - Real-time translation
  - Online learning systems

Key Insight: RNNs excel where sequential processing and low latency matter

Industry Reality: Despite transformers, RNNs remain crucial for specific use cases

Lab Preview: Build Your Own Text Generator

Week 3 Lab: Shakespeare Sonnet Generator with LSTM

What You'll Build:

1. Load Shakespeare sonnets
2. Preprocess and tokenize
3. Implement LSTM from scratch
4. Train character-level model
5. Generate new sonnets
6. Visualize hidden states

Key Learning Objectives:

- Understand sequence modeling
- Debug vanishing gradients
- Implement teacher forcing
- Use gradient clipping
- Tune hyperparameters
- Compare RNN vs LSTM

Sample Output:

"Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds..."

Bonus Challenges:

- Bidirectional LSTM
- Attention mechanism
- Beam search decoding
- Style transfer

Hands-On Learning: Theory becomes practice - build what you've learned

Interactive Exercise: Debug Training Problems

Your LSTM is training poorly. Diagnose the issue:

Symptoms:

- Loss = NaN after epoch 2
- Gradients ¿ 1000
- Outputs all same token
- Accuracy stuck at 10%

Your Diagnosis:

Problem: _____

Solution: _____

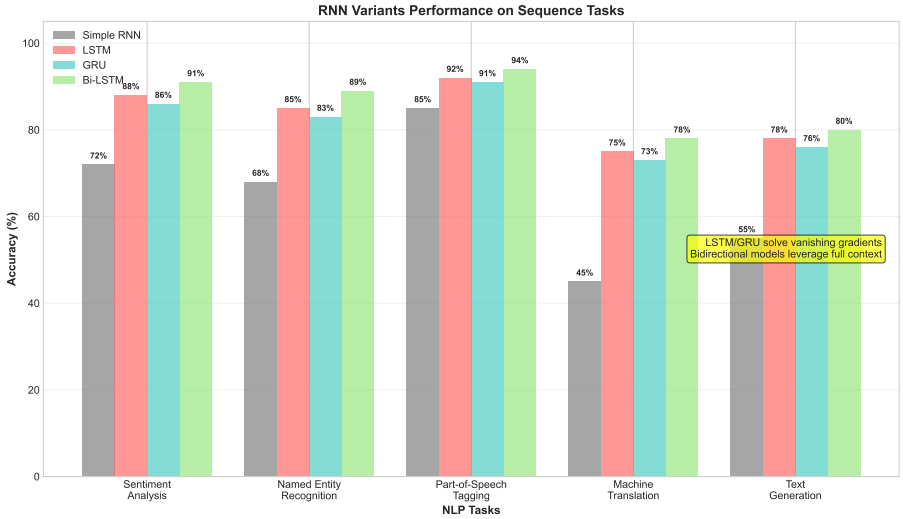
Prevention: _____

Common Issues & Fixes:

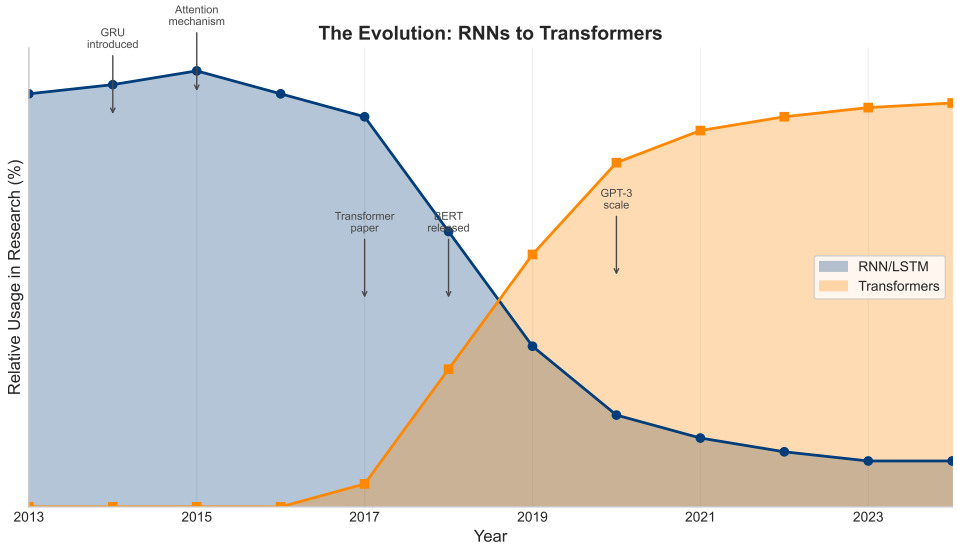
1. **Exploding Gradients**
   - Use gradient clipping
   - Reduce learning rate
   - Check weight initialization
2. **Poor Initialization**
   - Use Xavier/He init
   - Initialize forget gate bias to 1
3. **Wrong Loss Function**
   - Use CrossEntropyLoss
   - Check label dimensions

Debugging Skills: Recognizing symptoms leads to faster fixes

Performance Shootout: When to Use What



Course Roadmap: Where RNNs Fit



Week 3 Complete: You Now Understand Sequential Processing!

Part 1: Motivation

- Language is sequential
- Order creates meaning
- Need memory mechanisms

Part 2: Architecture

- Hidden state = memory
- Weight sharing across time
- Simple equations, complex behavior

Part 3: Problems & Solutions

- Vanishing gradients limit RNNs
- LSTMs use gates for control
- Addition creates gradient highway

Part 4: Implementation

- PyTorch makes it easy
- Gradient clipping essential
- Still used in production

Key Takeaways:

1. RNNs = Neural nets + Memory
2. LSTMs solve vanishing gradients
3. Gates control information flow
4. Foundation for modern NLP

Next Week: Sequence-to-Sequence Models and the Attention Revolution!

Congratulations: You've mastered the fundamentals of sequential neural networks!