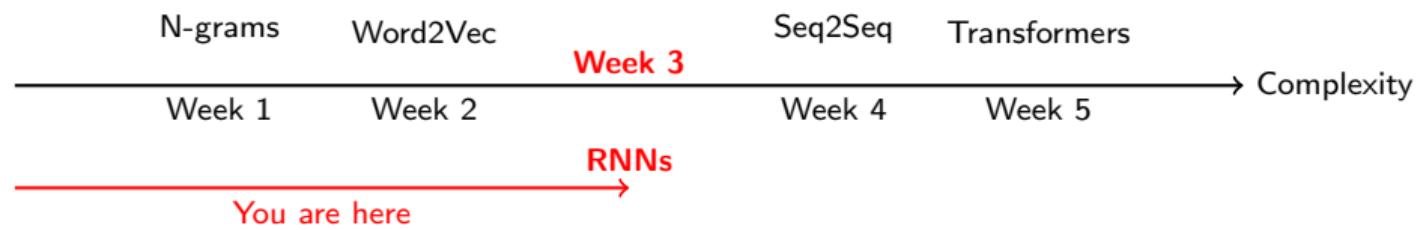


Natural Language Processing Course

Week 3: Recurrent Neural Networks
Teaching Networks to Remember

Joerg R. Osterrieder

Course Journey: Where We Are



Journey so far:

- Week 1: Words as counts (n-grams)
- Week 2: Words as vectors (embeddings)
- **Week 3: Processing sequences with memory**

Week 3: Learning Objectives

By the end of this week, you will:

- ① **Understand** why sequential processing matters in language
- ② **Build** intuition for how RNNs maintain memory
- ③ **Implement** a vanilla RNN from scratch
- ④ **Diagnose** the vanishing gradient problem
- ⑤ **Master** LSTM gates through hands-on coding
- ⑥ **Choose** between RNN, LSTM, and GRU for your tasks

Practical Skills:

- Build a character-level text generator
- Debug gradient flow in sequential models
- Implement LSTM gates step-by-step
- Know when RNNs beat Transformers (yes, it happens!)

Part 1

The Sequential Nature of Language

Why Order Matters

Evolution: From Bag of Words to Sequential Processing

The Bag of Words Era:

- Words as unordered sets
- "dog bites man" = {dog, bites, man}
- Lost all sequential information
- Still used in: Document classification

Word Embeddings (Week 2):

- Words as vectors
- Captured meaning
- Still no sequence modeling

The Sequential Revolution:

- Process words in order
- Maintain memory of past
- Understand context
- Handle variable lengths

Language is not a bag of words.
It's a carefully ordered sequence!

Why Order Matters: Simple but Profound Examples

Example 1: Same words, opposite meanings

- "Dog bites man" → Normal event
- "Man bites dog" → News headline!

Example 2: Negation changes everything

- "This movie is not bad" → Positive
- "This movie is bad, not good" → Negative

Example 3: Questions vs statements

- "Can you help?" → Request
- "You can help" → Statement

Example 4: Time and causality

- "I ate because I was hungry" → Cause after effect
- "Because I was hungry, I ate" → Cause before effect

Word embeddings see these as identical - they have the same vectors!

How Humans Read: Natural Sequential Processing

Reading this sentence: "The student who studied hard..."

- ① Read "The" → *Expect noun*
- ② Read "student" → *Remember: subject is student*
- ③ Read "who" → *Expect description of student*
- ④ Read "studied" → *Remember: student studied*
- ⑤ Read "hard" → *Remember: studied hard*
- ⑥ Expect: → *What happened to this student?*

Your brain maintains:

- Subject tracking (who/what)
- Verb tracking (actions)
- Modifier accumulation
- Expectation updating

This is exactly what RNNs do - maintain and update memory!

When Systems Forget: Real Voice Assistant Failures

Actual conversation with Alexa (2024):

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: No memory of "timer" from previous turn

More examples of context loss:

- "Play some Beatles... actually, make it Queen" → Confused
- "What's the weather?... And tomorrow?" → Lost context
- "Call mom... wait, call dad instead" → Doesn't understand "instead"

Solution needed: Networks that remember previous inputs!

The Fundamental Problem: Networks Need Memory

Feed-forward networks:

- Process each input independently
- No connection between time steps
- Can't handle sequences
- Fixed input size

What we need:

- Connect information across time
- Variable sequence lengths
- Memory of previous inputs
- Context-aware predictions

Mathematical requirement:

$$h_t = f(x_t, h_{t-1})$$

Where:

- h_t = hidden state (memory)
- x_t = current input
- h_{t-1} = previous memory

This recursion is the key insight!

RNN = Neural Network + Memory

Part 2

Simple RNNs

Networks with Memory

The Brilliant Idea: Hidden States as Memory

How to give a network memory?

The RNN approach:

- ① Start with empty memory $h_0 = 0$
- ② Read first word x_1
- ③ Update memory: $h_1 = f(x_1, h_0)$
- ④ Read second word x_2
- ⑤ Update memory: $h_2 = f(x_2, h_1)$
- ⑥ Continue...

Memory accumulates information!

Processing "The cat sat":

- $h_0 = [0, 0, 0, \dots]$ (blank)
- $h_1 = [0.2, -0.1, \dots]$ ("The")
- $h_2 = [0.3, 0.5, \dots]$ ("The cat")
- $h_3 = [0.1, 0.8, \dots]$ ("The cat sat")

h_3 contains compressed history!

RNN Mathematics: Just Two Equations!

The entire RNN in two lines:

$$\begin{aligned} h_t &= \tanh(W_h h_{t-1} + W_x x_t + b_h) && \text{Update memory} \\ y_t &= W_y h_t + b_y && \text{Generate output} \end{aligned}$$

Concrete example with actual numbers:

Let's say hidden size = 3, input size = 2:

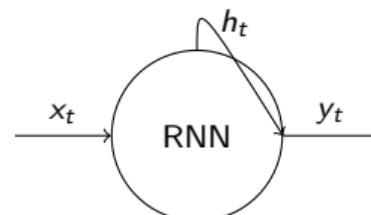
- $x_t = [0.5, -0.3]$ (current word embedding)
- $h_{t-1} = [0.1, 0.2, -0.1]$ (previous memory)
- W_x is 3×2 , W_h is 3×3 , W_y is vocab_size $\times 3$

Step-by-step computation:

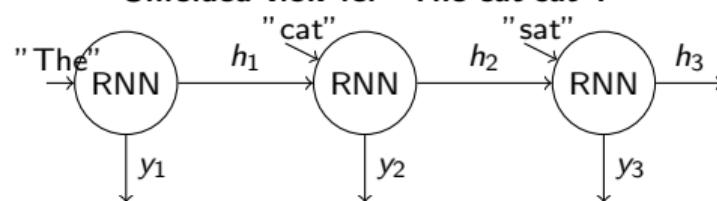
- ❶ $W_x x_t = [0.2, -0.1, 0.3]$ (transform input)
- ❷ $W_h h_{t-1} = [0.1, 0.0, -0.2]$ (transform memory)
- ❸ Sum + bias = $[0.3, -0.1, 0.1]$
- ❹ $h_t = \tanh([0.3, -0.1, 0.1]) = [0.29, -0.10, 0.10]$

Visualizing RNNs: Unfolding Through Time

Compact view:



Unfolded view for "The cat sat":



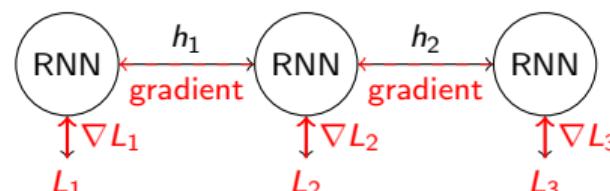
Same weights (RNN box) used at every time step!

Building an RNN: Complete Implementation

```
1 import numpy as np
2
3 class SimpleRNN:
4     def __init__(self, input_size, hidden_size, output_size):
5         # Initialize weights randomly
6         self.Wxh = np.random.randn(hidden_size, input_size) * 0.01
7         self.Whh = np.random.randn(hidden_size, hidden_size) * 0.01
8         self.Why = np.random.randn(output_size, hidden_size) * 0.01
9         self.bh = np.zeros((hidden_size, 1))
10        self.by = np.zeros((output_size, 1))
11
12    def step(self, x, h_prev):
13        """Process one time step"""
14        # Update hidden state (memory)
15        h = np.tanh(self.Wxh @ x + self.Whh @ h_prev + self.bh)
16
17        # Compute output
18        y = self.Why @ h + self.by
19
20        return y, h
21
22    def forward(self, inputs):
23        """Process entire sequence"""
24        h = np.zeros((self.Whh.shape[0], 1)) # Initial memory
25        outputs = []
26
27        for x in inputs:
28            y, h = self.step(x, h)
29            outputs.append(y)
30
31        return outputs, h
```

Training RNNs: Backpropagation Through Time (BPTT)

The challenge: Error must flow back through all time steps



BPTT algorithm:

- ① Forward pass: Compute all hidden states and outputs
- ② Compute loss at each time step
- ③ Backward pass: Accumulate gradients through time
- ④ Update weights using total gradient

Key insight: Gradient has to flow through many matrix multiplications!

Example: Character-Level Text Generation

Task: Train RNN to generate "hello"

```
1 # Training data
2 text = "hello"
3 chars = list(set(text)) # unique chars: h,e,l,o
4 char_to_idx = {ch:i for i,ch in enumerate(chars)}
5
6 # Prepare training sequences
7 # Input: h e l l
8 # Target: e l l o
9
10 # After training, generate:
11 def generate(rnn, seed_char='h', length=10):
12     h = np.zeros((hidden_size, 1))
13     x = one_hot(seed_char)
14     result = seed_char
15
16     for _ in range(length):
17         y, h = rnn.step(x, h)
18         next_char = sample(softmax(y))
19         result += next_char
20         x = one_hot(next_char)
21
22     return result
23
24 # Output: "hellohelllo..." (learns the pattern!)
```

Part 3

The Vanishing Gradient Problem

Why Simple RNNs Fail

The Fatal Flaw: Gradients Vanish Over Time

Try to process this sentence:

"The student who the professor who won the Nobel Prize taught **was** brilliant"

- "was" agrees with "student" (15 words back!)
- RNN must remember "student" is singular
- But after 15 steps, memory is almost gone

Mathematical explanation:

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

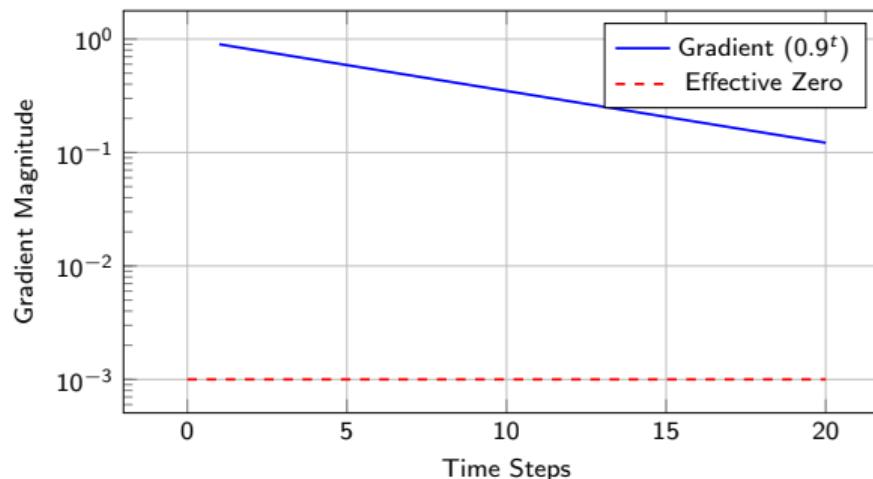
$$\text{Each } \frac{\partial h_t}{\partial h_{t-1}} = W_h^T \cdot \text{diag}(\tanh'(h_{t-1}))$$

Since $|\tanh'(x)| \leq 1$ and typical $\|W_h\| < 1$:

After T steps: gradient $\approx (0.9)^T \rightarrow$ exponentially small!

Gradient Vanishing: A Visual Demonstration

Gradient magnitude after T time steps:



Consequence: After 15 steps, gradient $\approx 0 \rightarrow$ No learning!

Where Simple RNNs Break: Real Examples

1. Long-range dependencies:

- "The keys that I left on the table in the kitchen yesterday are missing"
- RNN forgets "keys" are plural by the time it reaches "are"

2. Question answering:

- "What did John give Mary? ... [long story] ... John gave Mary a book"
- RNN can't connect answer to question after long context

3. Code understanding:

```
1 def complex_function(x):
2     # ... 20 lines of code ...
3     return result # What is result? RNN forgot!
```

4. Document summarization:

- Introduction mentions key point
- 10 paragraphs later, conclusion should reference it
- RNN has completely forgotten the introduction

The Solution: Gated Memory (Preview of LSTM)

The key insight: Create shortcuts for gradient flow!

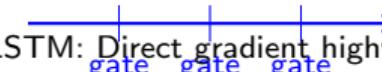
Simple RNN:

- Gradient must flow through tanh
- Multiplied by weights each step
- Exponential decay
- No control over memory

LSTM Solution:

- Add "highway" for gradients
- Gates control information flow
- Can preserve gradient for 100+ steps
- Selective memory

RNN: Many operations


LSTM: Direct gradient highway


Part 4

LSTM & GRU

The Gated Solution

LSTM: The Breakthrough (1997)

Long Short-Term Memory by Hochreiter & Schmidhuber

The genius idea: Separate memory into two parts:

- ① **Cell state (c_t):** Long-term memory highway
- ② **Hidden state (h_t):** Short-term working memory

Three gates control memory:

- **Forget gate:** What to discard from memory
- **Input gate:** What new information to store
- **Output gate:** What to output now

Analogy: Email inbox management

- Forget gate = Delete spam
- Input gate = Save important emails
- Output gate = What to reply to now

LSTM Architecture: Gates Explained with Numbers

Processing "The cat sat on the mat"

At time step for "sat":

- ➊ **Forget gate:** $f_t = \sigma(W_f \cdot [h_{t-1}, x_t])$
 - Output: $f_t = 0.9 \rightarrow$ Keep 90% of previous memory
 - "The cat" is still relevant
- ➋ **Input gate:** $i_t = \sigma(W_i \cdot [h_{t-1}, x_t])$
 - Output: $i_t = 0.8 \rightarrow$ Store 80% of new info
 - "sat" is important action
- ➌ **Candidate values:** $\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t])$
 - New information to potentially store
- ➍ **Update cell state:** $c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$
 - Combines old (90%) and new (80%) memory
- ➎ **Output gate:** $o_t = \sigma(W_o \cdot [h_{t-1}, x_t])$
 - Output: $o_t = 0.7 \rightarrow$ Use 70% for current output

LSTM Implementation: Complete Code

```
1 import numpy as np
2
3 def sigmoid(x):
4     return 1 / (1 + np.exp(-x))
5
6 class LSTM:
7     def __init__(self, input_size, hidden_size):
8         # Weight matrices for each gate
9         self.Wf = np.random.randn(hidden_size, input_size + hidden_size) * 0.01
10        self.Wi = np.random.randn(hidden_size, input_size + hidden_size) * 0.01
11        self.Wc = np.random.randn(hidden_size, input_size + hidden_size) * 0.01
12        self.Wo = np.random.randn(hidden_size, input_size + hidden_size) * 0.01
13
14    def step(self, x, h_prev, c_prev):
15        # Concatenate input and previous hidden state
16        combined = np.concatenate([h_prev, x])
17
18        # Forget gate: decide what to forget
19        f_t = sigmoid(self.Wf @ combined)
20
21        # Input gate: decide what to store
22        i_t = sigmoid(self.Wi @ combined)
23
24        # Candidate values: create new information
25        c_tilde = np.tanh(self.Wc @ combined)
26
27        # Update cell state (long-term memory)
28        c_t = f_t * c_prev + i_t * c_tilde
29
30        # Output gate: decide what to output
31        o_t = sigmoid(self.Wo @ combined)
32
33        # Hidden state (short-term memory)
34        h_t = o_t * np.tanh(c_t)
35
36        return h_t, c_t
```

GRU: The Simplified Alternative (2014)

Gated Recurrent Unit - Fewer parameters, similar performance

Simplifications from LSTM:

- Combines forget and input gates → Update gate
- No separate cell state (only hidden state)
- 25% fewer parameters

GRU equations:

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

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

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t]) \quad \text{Candidate}$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \quad \text{Final output}$$

When to use GRU vs LSTM:

- GRU: Smaller datasets, faster training needed
- LSTM: Complex long-range dependencies, more data available

RNN vs LSTM vs GRU: Complete Comparison

Aspect	RNN	LSTM	GRU
Year introduced	1986	1997	2014
Parameters	$O(h^2)$	$O(4h^2)$	$O(3h^2)$
Gates	0	3	2
Memory	Short	Long	Long
Gradient flow	Poor	Excellent	Good
Training speed	Fast	Slow	Medium
Max sequence	10-20	100+	100+

Practical recommendations:

- **Start with:** GRU (good balance)
- **If underfitting:** Switch to LSTM
- **If overfitting:** Try simpler RNN
- **For production:** LSTM (most proven)

When RNNs Beat Transformers (Yes, in 2024!)

RNNs are not dead! They excel at:

① Streaming/Online processing:

- Process one token at a time
- $O(1)$ memory per step
- Perfect for live transcription

② Time series with clear patterns:

- Stock prices, weather data
- Energy consumption
- IoT sensor streams

③ Resource-constrained devices:

- Mobile phones (Gboard uses RNN-T)
- Embedded systems
- Edge AI applications

④ Variable-length generation:

- Music generation (MuseNet uses LSTM)
- Handwriting synthesis
- Speech synthesis

Rule: Use RNNs when sequential processing is natural and efficient

Key Takeaways: Week 3

What you've learned:

- ① **Sequential processing** is fundamental to language
- ② **RNNs** add memory through hidden states
- ③ **Vanishing gradients** limit simple RNNs to 20 steps
- ④ **LSTM gates** create gradient highways for long-range dependencies
- ⑤ **GRU** simplifies LSTM with similar performance

Key formulas to remember:

- RNN: $h_t = \tanh(W_h h_{t-1} + W_x x_t)$
- LSTM: $c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$
- Gradient vanishing: $(0.9)^T \rightarrow 0$

Next week: Sequence-to-Sequence models (Translation, Summarization)

References and Further Reading

Original Papers:

- Rumelhart et al. (1986): "Learning representations by back-propagating errors"
- Hochreiter & Schmidhuber (1997): "Long Short-Term Memory"
- Cho et al. (2014): "GRU - Gated Recurrent Unit"

Recommended Tutorials:

- Karpathy (2015): "The Unreasonable Effectiveness of RNNs"
- Olah (2015): "Understanding LSTM Networks" (colah.github.io)
- PyTorch RNN Tutorial: pytorch.org/tutorials

Practice Resources:

- Week 3 Lab: Character-level language model
- Assignment: Build LSTM from scratch
- Challenge: Implement attention mechanism

Complete LSTM Mathematics

Forward pass equations:

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

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

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \quad \text{Candidate} \quad (3)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad \text{Cell state} \quad (4)$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad \text{Output gate} \quad (5)$$

$$h_t = o_t \odot \tanh(c_t) \quad \text{Hidden state} \quad (6)$$

Gradient flow analysis:

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t \quad (\text{can be close to } 1!)$$

This allows gradient to flow unchanged if forget gate ≈ 1

Computational Complexity Analysis

Time complexity per time step:

Model	Operations	Memory
RNN	$O(h^2 + hx)$	$O(h)$
LSTM	$O(4(h^2 + hx))$	$O(2h)$
GRU	$O(3(h^2 + hx))$	$O(h)$
Transformer	$O(n^2d)$	$O(n^2)$

Where:

- h = hidden size (typically 256-1024)
- x = input size
- n = sequence length
- d = model dimension

Key insight: RNNs are $O(n)$ in sequence length, Transformers are $O(n^2)$

Real-World RNN Applications (2024)

1. Speech Recognition:

- Google's RNN-T on Pixel phones
- Apple's on-device Siri processing
- Works offline, low latency

2. Time Series Forecasting:

- Stock price prediction (LSTM dominant)
- Weather forecasting
- Traffic flow prediction

3. Music Generation:

- MuseNet (OpenAI) uses LSTM
- Magenta project (Google)
- Real-time composition

4. Healthcare:

- ECG anomaly detection
- Patient monitoring
- Drug discovery sequences