Natural Language Processing Course

Week 3: Recurrent Neural Networks (RNNs)

Joerg R. Osterrieder
www.joergosterrieder.com

Week 3

Recurrent Neural Networks

Teaching Computers to Remember

Why Your Voice Assistant Sometimes Fails

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"

Word embeddings don't remember what came before!

The problem: Understanding "it" requires remembering "timer"
The solution: Networks that maintain memory of past inputs

RNNs Power Sequential Understanding Everywhere

Voice Assistants (2024):

- Siri: LSTM for complex commands[1]
- Google: RNN-T model (450MB)[2]
- Context maintained across turns

Text Generation:

- Gmail Smart Compose
- Code completion (before Copilot)
- Character-by-character prediction

Still Best For:

- Stock price prediction[3]
- Speech recognition on phones
- Music generation
- Energy demand forecasting

Key Advantage:
Processes sequences step-by-step, just like humans read!

[1] Apple's on-device processing
[2] Gboard's efficient on-device model
[3] LSTMs dominate financial forecasting in 2024

Week 3: What You'll Master

By the end of this week, you will:

- **Understand** why order matters in language
- **Build** intuition for how RNNs maintain memory
- **Implement** an LSTM from scratch
- **Solve** the vanishing gradient problem
- **Create** a text generator that remembers context

Core Insight: Process sequences like humans do - one step at a time, remembering what came before

Why Order Matters: A Simple Example

Same words, different order, different meaning:

1. "Dog bites man" (Not news)
2. "Man bites dog" (Front page news!)

Word embeddings alone can't distinguish these!
Both have same word vectors: {dog, bites, man}

More examples where order is crucial:

- "not bad" vs "bad, not good"
- "can you?" vs "you can"
- "barely passed" vs "passed barely" (different emphasis)

Language is fundamentally sequential - we need models that process it that way

The Brilliant Idea: Networks with Memory

How humans read:
"The movie was really..."

- Read "The" → remember it
- Read "movie" → remember "The movie"
- Read "was" → remember "The movie was"
- Read "really" → expect adjective next

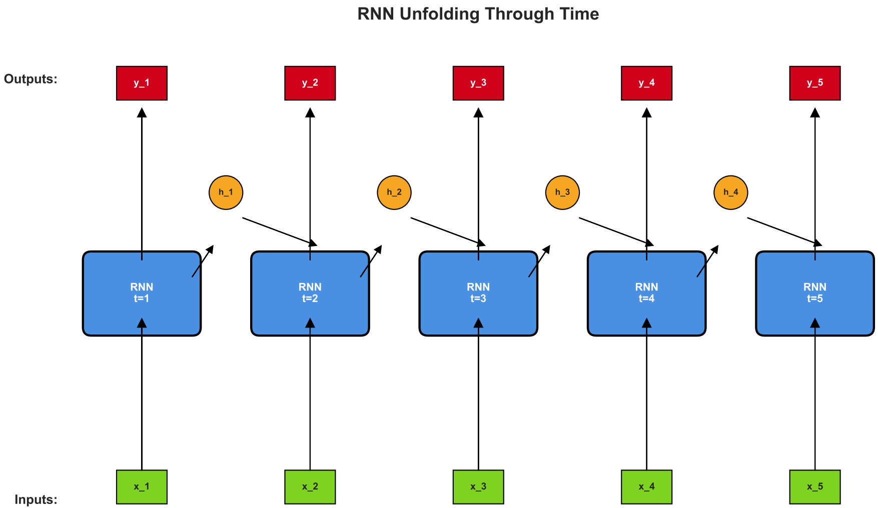
RNN does exactly this:[4]

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

Hidden state = What the network remembers so far

[4] Rumelhart, Hinton & Williams (1986). "Learning representations by back-propagating errors", Nature

Visualizing RNNs: Unfolding Through Time



Same weights W shared across all time steps!

RNN Mathematics: Surprisingly Simple!

Just two equations:

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

Where:

- h_t: Hidden state (memory) at time t
- x_t: Input word embedding at time t
- y_t: Output prediction at time t
- W: Weight matrices (shared across time!)

In plain English:
New memory = function(old memory + new input)

Building an RNN: Complete Implementation

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

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size,
         output_size):
        """Initialize RNN with typical hidden size 256"""
        super().__init__()
        self.hidden_size = hidden_size

        # Learnable parameters
        self.i2h = nn.Linear(input_size + hidden_size,
             hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.tanh = nn.Tanh()

    def forward(self, input, hidden):
        """Process one time step"""
        # Combine input and previous hidden state
        combined = torch.cat((input, hidden), 1)

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

        # Generate output
        output = self.h2o(hidden)

        return output, hidden

    def init_hidden(self, batch_size):
```

Design Choices:

- Hidden size typically 128-512[5]
- Tanh keeps values in [-1, 1]
- Same weights for all time steps

Usage Pattern: `hidden = rnn.init_hidden(32)`

```
for word in sentence:
    out, hidden = rnn(word, hidden)
```

256 is memory-bandwidth optimal

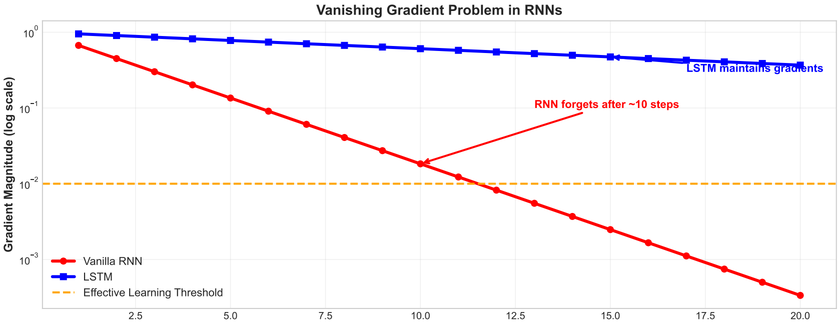
The Fatal Flaw: Vanishing Gradients

Try to learn from this sentence:
"The student who the professor who won the Nobel Prize taught was brilliant"

Problem: "was" agrees with "student" (15 words back!)

What happens during training:[6]

- Gradient flows backward through time
- Gets multiplied by weights at each step
- After 10-15 steps: gradient ≈ 0
- Network can't learn long dependencies!



The LSTM Solution: Gated Memory

The breakthrough (1997):[7] Add gates to control memory!

Three gates, like a smart filing system:

1. **Forget Gate:** What to throw away
2. **Input Gate:** What new info to store
3. **Output Gate:** What to use right now

Analogy: Reading a mystery novel

- See new character $\rightarrow$ Store in memory (input gate)
- Character becomes irrelevant $\rightarrow$ Forget them (forget gate)
- Need to solve mystery $\rightarrow$ Recall important clues (output gate)

LSTMs can remember for 100+ steps (vs 10-15 for vanilla RNNs)

[7] Hochreiter & Schmidhuber (1997). "Long Short-Term Memory", Neural Computation

LSTM Implementation: The Gated Architecture

```python
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        """LSTM with typical hidden size 256-512"""
        super().__init__()
        self.hidden_size = hidden_size

        # Gates
        self.forget_gate = nn.Linear(input_size + hidden_size,
              hidden_size)
        self.input_gate = nn.Linear(input_size + hidden_size,
              hidden_size)
        self.candidate_gate = nn.Linear(input_size + hidden_size,
              hidden_size)
        self.output_gate = nn.Linear(input_size + hidden_size,
              hidden_size)

        # Output projection
        self.h2o = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden, cell):
        """Process one time step with gated memory"""
        combined = torch.cat((input, hidden), 1)

        # Forget gate: what to discard from memory
        f_gate = torch.sigmoid(self.forget_gate(combined))

        # Input gate: what new info to store
        i_gate = torch.sigmoid(self.input_gate(combined))
        candidate = torch.tanh(self.candidate_gate(combined))

        # Update cell state (long-term memory)
        cell = f_gate * cell + i_gate * candidate

        # Output gate: what to output based on memory
        o_gate = torch.sigmoid(self.output_gate(combined))
```

Why Gates Work:

- Sigmoid: 0-1 range (percentage)
- Multiplication: gating mechanism
- Addition: gradient highway

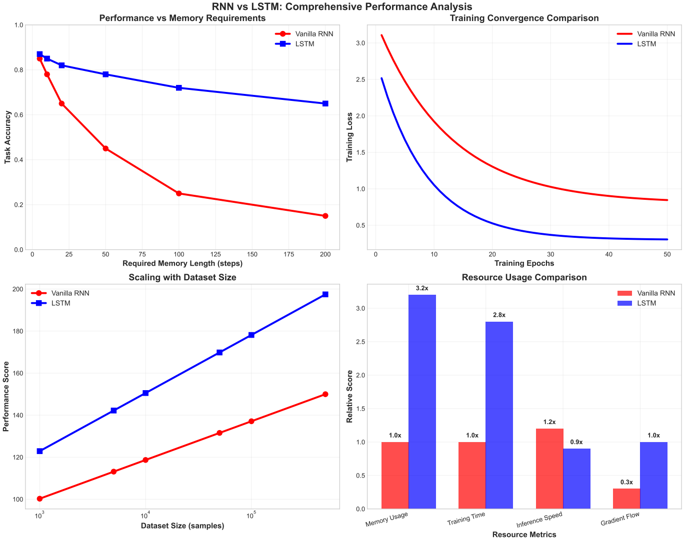
Memory Management:

- `cell`: Long-term memory
- `hidden`: Working memory
- Gates control information flow

Training Benefits:

- Gradients flow through addition
- Can learn 100+ step dependencies
- Solves vanishing gradient!

RNN vs LSTM: The Difference is Dramatic



Key Insights

RNNs vs Transformers: When Sequential Wins (2024)

Despite transformer dominance, RNNs still excel at:

1. **Resource-Constrained:**
   - Google's RNN-T: 450MB[8]
   - Transformer equivalent: 2-5GB
   - Perfect for phones/IoT
2. **Streaming/Real-time:**
   - Process as data arrives
   - No need to see entire sequence
   - Live transcription, translation
3. **Time Series:**
   - Stock prediction[9]
   - Energy demand forecasting
   - ES-RNN won M4 competition
4. **Truly Sequential:**
   - Music generation
   - Handwriting synthesis
   - Robot control sequences

Rule: Use RNNs when order truly matters and resources are limited

[1] Gboard on-device speech recognition

[2] LSTMs still dominate financial forecasting in 2024

Common RNN Pitfalls and Solutions

1. **Exploding Gradients**
   - Problem: Gradients grow exponentially
   - Solution: Gradient clipping (max norm = 5)
2. **Exposure Bias**[10]
   - Problem: Train with truth, test with predictions
   - Solution: Scheduled sampling (mix both)
3. **Slow Training**
   - Problem: Can't parallelize across time
   - Solution: Truncated backprop, smaller sequences

Real Example - Text Generation:

- Without fixes: "The the the the..."
- With fixes: "The movie was really entertaining"

[10] Major cause of repetition and hallucination in generated text

Week 3 Exercise: Build a Context-Aware Chatbot

Your Mission: Create a chatbot that remembers conversation context

Example Conversation:

- User: "My name is Alice"
- Bot: "Nice to meet you, Alice!"
- User: "What's my name?"
- Bot: "Your name is Alice"

Implementation Steps:

1. Implement LSTM-based encoder
2. Maintain conversation state
3. Generate contextual responses
4. Handle 5-10 turn conversations

Bonus Challenges:

- Compare RNN vs LSTM memory retention
- Visualize hidden states over conversation
- Implement attention to see what it remembers
- Try different hidden sizes (128, 256, 512)

You'll discover: Why Siri sometimes forgets context mid-conversation!

Key Takeaways: Sequential Processing Matters

What we learned:

- Language is inherently sequential - order matters!
- RNNs process sequences step-by-step with memory
- Vanilla RNNs suffer from vanishing gradients (10 steps)
- LSTMs use gates to remember for 100+ steps
- Still best for resource-constrained and streaming applications

The evolution:

N-grams (no memory) → Word2Vec (no order) → RNNs (sequential memory)

Next week: Sequence-to-Sequence
How do we use RNNs for translation, where input and output lengths differ?

References and Further Reading

Foundational Papers:

- Rumelhart et al. (1986). "Learning representations by back-propagating errors", Nature
- Hochreiter & Schmidhuber (1997). "Long Short-Term Memory", Neural Computation
- Bengio et al. (1994). "Learning long-term dependencies with gradient descent is difficult"

Modern Applications:

- Google's RNN-T for on-device speech (2024)
- ES-RNN winning M4 forecasting competition
- Financial time series with LSTMs

Recommended Resources:

- Colah's Blog: "Understanding LSTM Networks" (visual guide)
- Karpathy's "The Unreasonable Effectiveness of RNNs"
- PyTorch RNN tutorial with Shakespeare generation