Complete Guide to Recurrent Neural Networks

RNN, LSTM, GRU, and Advanced LSTM Architectures

Table of Contents

---

1. Introduction to Sequential Data {#introduction}

What is Sequential Data?

Sequential data is information where the order of elements matters. Examples include:

- **Text**: "I love this movie" vs "This movie I love" (different meanings)
- **Time Series**: Stock prices, weather data, sensor readings
- **Speech**: Audio signals where timing is crucial
- **DNA Sequences**: Order of nucleotides determines genetic information

Why Traditional Neural Networks Fail?

Standard feedforward neural networks treat each input independently:

```
Input: [word1, word2, word3]
Processing: f(word1), f(word2), f(word3) - No connection between words
```

Problems:

- No memory of previous inputs
- Fixed input size requirement
- Cannot capture temporal dependencies
- Loses context information

The Need for Recurrent Networks

Recurrent networks process sequences by maintaining **hidden states** that carry information from previous time steps:

```
h₀ → h₁ → h₂ → h₃ → ...
 ↑    ↑    ↑    ↑
 x₁   x₂   x₃   x₄
```

---

2. Recurrent Neural Networks (RNN) {#rnn}

2.1 Basic RNN Architecture

Mathematical Foundation

```
h_t = tanh(W_hh × h_{t-1} + W_xh × x_t + b_h)
y_t = W_hy × h_t + b_y
```

Where:

- `h_t`: Hidden state at time t
- `x_t`: Input at time t
- `y_t`: Output at time t
- `W_hh`: Hidden-to-hidden weight matrix
- `W_xh`: Input-to-hidden weight matrix
- `W_hy`: Hidden-to-output weight matrix
- `b_h`, `b_y`: Bias vectors

Visual Representation

```
 y₁    y₂    y₃
 ↑     ↑     ↑
[h₁] → [h₂] → [h₃]
 ↑     ↑     ↑
 x₁    x₂    x₃
```

2.2 RNN Processing Flow

Step-by-Step Example: Sentiment Analysis

Input Sentence: "This movie is great"

Step 1: Initialize $h_0 = [0, 0, 0, 0]$ (4-dimensional hidden state)

Step 2: Process "This"

```
x₁ = embedding("This") = [0.2, -0.1, 0.5, 0.3]
h₁ = tanh(W_hh × h₀ + W_xh × x₁ + b_h)
h₁ = [0.1, 0.3, -0.2, 0.4]
```

Step 3: Process "movie"

```
x₂ = embedding("movie") = [0.4, 0.1, -0.3, 0.2]
h₂ = tanh(W_hh × h₁ + W_xh × x₂ + b_h)
h₂ = [0.3, 0.2, 0.1, 0.5]
```

Continue for remaining words...

2.3 Strengths of RNN

1. **Variable Length Input**: Can handle sequences of any length
2. **Parameter Sharing**: Same weights used across all time steps
3. **Memory**: Maintains information from previous inputs
4. **Contextual Understanding**: Can capture dependencies between elements

2.4 Critical Limitations

Vanishing Gradient Problem

Mathematical Explanation: During backpropagation through time, gradients are computed as:

```
∂L/∂h_t = ∂L/∂h_{t+1} × ∂h_{t+1}/∂h_t
```

Since `∂h_{t+1}/∂h_t` involves the tanh derivative (max value = 1), gradients shrink exponentially:

```
∂L/∂h_1 = ∂L/∂h_T × (∂h_T/∂h_{T-1}) × ... × (∂h_2/∂h_1)
```

Consequence: Earlier time steps receive negligible gradient updates.

Practical Example of Vanishing Gradients

```
Sentence: "The cat, which was sitting on the mat that my grandmother bought years ago, was sleeping."

Problem: By the time RNN processes "was sleeping",
it has forgotten "cat" due to vanishing gradients.

Result: Poor understanding of long-range dependencies.
```

Short-Term Memory Problem

RNNs struggle with information that occurred many steps ago:

```
Text: "I grew up in France... [100 words later] ...so I speak French fluently."

RNN often fails to connect "France" with "French" due to the gap.
```

2.5 RNN Variants

Bidirectional RNN

Processes sequences in both directions:

```
Forward:  h₁ → h₂ → h₃ → h₄
Backward: h₄ ← h₃ ← h₂ ← h₁
```

Advantage: Access to both past and future context **Use Case**: When entire sequence is available (not real-time)

Deep RNN

Multiple RNN layers stacked:

```
Layer 2: h₁² → h₂² → h₃²
         ↑    ↑    ↑
Layer 1: h₁¹ → h₂¹ → h₃¹
         ↑    ↑    ↑
Input:   x₁   x₂   x₃
```

---

3. Long Short-Term Memory (LSTM) {#lstm}

3.1 The LSTM Solution

LSTM addresses RNN's vanishing gradient problem through a sophisticated **gating mechanism** that controls information flow.

Key Innovation: Cell State

```
C_t: Cell state (long-term memory)
h_t: Hidden state (short-term memory)
```

3.2 LSTM Architecture Components

3.2.1 Forget Gate

Purpose: Decide what information to discard from cell state

```
f_t = σ(W_f × [h_{t-1}, x_t] + b_f)
```

Example in Sentiment Analysis:

```
Text: "The movie started boring but became exciting"
When processing "exciting", forget gate might decide to forget
the negative sentiment from "boring"
```

3.2.2 Input Gate

Purpose: Decide what new information to store

```
i_t = σ(W_i × [h_{t-1}, x_t] + b_i)
C̃_t = tanh(W_C × [h_{t-1}, x_t] + b_C)
```

Example:

```
When processing "exciting", input gate decides to store
this positive sentiment information
```

3.2.3 Cell State Update

Purpose: Update long-term memory

```
C_t = f_t * C_{t-1} + i_t * C̃_t
```

Intuition:

- Forget irrelevant old information (f_t * C_{t-1})
- Add relevant new information (i_t * C̃_t)

3.2.4 Output Gate

Purpose: Control what parts of cell state to output

```
o_t = σ(W_o × [h_{t-1}, x_t] + b_o)
h_t = o_t * tanh(C_t)
```

3.3 Complete LSTM Forward Pass

Mathematical Formulation

```
f_t = σ(W_f × [h_{t-1}, x_t] + b_f)    # Forget gate
i_t = σ(W_i × [h_{t-1}, x_t] + b_i)    # Input gate
C̃_t = tanh(W_C × [h_{t-1}, x_t] + b_C) # Candidate values
C_t = f_t * C_{t-1} + i_t * C̃_t        # Cell state
o_t = σ(W_o × [h_{t-1}, x_t] + b_o)    # Output gate
h_t = o_t * tanh(C_t)                  # Hidden state
```

Visual Flow Diagram

```
×──────────────────×
│    C_{t-1}   │ Cell State
│      │     │
│      ▼     │
│    ┌──────┐    │
│ ┌─→│ × f │    │ Forget Gate
│ │  └──────┘    │
│ │     │     │
│ │     ▼     │
│ │  ┌──────┐ ┌──┴──┐
│ │  │ + ← │ │ i× │ Input Gate & Candidate
│ │  └──────┘ └─────┘
│ │     │
│ │     ▼   C_t
└─┴────────┼──────────→
    │
    ▼
   ┌──────┐
 ┌─→│ ×o │ Output Gate
 │  └──────┘
 │   │
 │   ▼
 │  h_t ──────────→
 │
[h_{t-1}, x_t]
```

3.4 Why LSTM Solves Vanishing Gradients

Gradient Flow Analysis

The cell state provides an **additive** path for gradients:

```
C_t = f_t * C_{t-1} + i_t * C̃_t

∂C_t/∂C_{t-1} = f_t (element-wise multiplication, not matrix multiplication)
```

Key Insight: Gradients can flow through the cell state without diminishing, as long as forget gate values are close to 1.

Comparison: RNN vs LSTM Gradient Flow

```
RNN:  ∂h_t/∂h_{t-1} = W × tanh'(...)  (multiplicative, diminishing)
LSTM: ∂C_t/∂C_{t-1} = f_t            (additive path available)
```

3.5 LSTM Variants

3.5.1 Peephole Connections

Allow gates to look at cell state:

```
f_t = σ(W_f × [C_{t-1}, h_{t-1}, x_t] + b_f)
i_t = σ(W_i × [C_{t-1}, h_{t-1}, x_t] + b_i)
o_t = σ(W_o × [C_t, h_{t-1}, x_t] + b_o)
```

3.5.2 Coupled Forget and Input Gates

```
f_t = σ(W_f × [h_{t-1}, x_t] + b_f)
i_t = 1 - f_t
```

---

4. Gated Recurrent Unit (GRU) {#gru}

4.1 GRU Motivation

GRU simplifies LSTM by:

- Combining cell and hidden states
- Using only 2 gates instead of 3
- Reducing computational complexity

4.2 GRU Architecture

Mathematical Formulation

```
z_t = σ(W_z × [h_{t-1}, x_t])    # Update gate
r_t = σ(W_r × [h_{t-1}, x_t])    # Reset gate
h̃_t = tanh(W × [r_t * h_{t-1}, x_t])  # Candidate hidden state
h_t = (1 - z_t) * h_{t-1} + z_t * h̃_t  # Final hidden state
```

Gate Functions Explained

Update Gate (z_t): Controls how much past information to keep

- z_t ≈ 0: Keep old information (h_{t-1})
- z_t ≈ 1: Use new information (h̃_t)

Reset Gate (r_t): Controls how much past information to forget when computing candidate

- r_t ≈ 0: Ignore past hidden state
- r_t ≈ 1: Use full past hidden state

4.3 GRU vs LSTM Comparison

Structural Differences

```
LSTM: Separate cell state (C_t) and hidden state (h_t)
GRU:  Combined state (h_t only)

LSTM: 3 gates (forget, input, output)
GRU:  2 gates (update, reset)
```

Parameter Count

```
LSTM: 4 × (input_size + hidden_size + 1) × hidden_size parameters
GRU:  3 × (input_size + hidden_size + 1) × hidden_size parameters

Example with input_size=100, hidden_size=128:
LSTM: 4 × (100 + 128 + 1) × 128 = 117,504 parameters
GRU:  3 × (100 + 128 + 1) × 128 = 88,128 parameters
```

4.4 GRU Processing Example

Sentiment Analysis: "This movie is not good"

Step 1: Process "This"

```
z₁ = σ([0.6, 0.4, 0.2, 0.8])   # Update gate
r₁ = σ([0.3, 0.7, 0.5, 0.9])   # Reset gate
h̃₁ = tanh([0.1, 0.4, -0.2, 0.6]) # Candidate
h₁ = (1-z₁) * h₀ + z₁ * h̃₁      # Combined state
```

Step 2: Process "movie"

```
Context: Now knows about "This"
Updates: Incorporates "movie" information
```

Step 3: Process "is"

```
Function: Minimal content, gates likely reduce influence
```

Step 4: Process "not"

```
Critical: Reset gate might prepare to change sentiment direction
```

Step 5: Process "good"

```
Integration: Update gate balances "good" with "not" context
Result: Overall negative sentiment despite "good"
```

---

5. Advanced LSTM Architectures {#advanced-lstm}

5.1 Stacked LSTM

Architecture

```
Layer 3: LSTM₃ → h₃,t
         ↑
Layer 2: LSTM₂ → h₂,t
         ↑
Layer 1: LSTM₁ → h₁,t
         ↑
Input:   x_t
```

Benefits

- **Hierarchical Representation**: Each layer captures different abstraction levels
- **Increased Capacity**: More parameters for complex pattern recognition
- **Feature Extraction**: Lower layers extract basic features, higher layers combine them

Layer Interpretation Example (Text Processing)

```
Layer 1: Word-level features (POS tags, word types)
Layer 2: Phrase-level patterns (noun phrases, verb phrases)
Layer 3: Sentence-level semantics (sentiment, intent)
```

5.2 Bidirectional LSTM

Architecture

Forward LSTM: $h_1^f \rightarrow h_2^f \rightarrow h_3^f \rightarrow h_4^f$
Backward LSTM: $h_4^b \leftarrow h_3^b \leftarrow h_2^b \leftarrow h_1^b$
Combined: $[h_1^f;h_1^b]$ $[h_2^f;h_2^b]$ $[h_3^f;h_3^b]$ $[h_4^f;h_4^b]$

Mathematical Formulation

```
h_t^f = LSTM_forward(x_t, h_{t-1}^f)    # Forward pass
h_t^b = LSTM_backward(x_t, h_{t+1}^b)   # Backward pass
h_t = [h_t^f; h_t^b]                    # Concatenation
```

Use Case Example: Named Entity Recognition

```
Sentence: "Barack Obama was born in Hawaii"

Forward context at "Obama": ["Barack"]
Backward context at "Obama": ["was", "born", "in", "Hawaii"]
Combined understanding: "Barack Obama" is likely a person name
```

5.3 LSTM with Attention Mechanism

Attention Concept

Instead of using only the final hidden state, attention allows the model to focus on relevant parts of the entire sequence.

Architecture

LSTM Hidden States: h_1, h_2, h_3, h_4
Attention Weights: $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ (sum to 1)
Context Vector: $c = \Sigma(\alpha_i \times h_i)$

Mathematical Formulation

```
e_i = score(h_i, h_final)           # Attention scores
α_i = softmax(e_i)                  # Normalized weights
c = Σ(α_i × h_i)                    # Weighted sum
```

Example: Machine Translation

```
English: "The cat sits on the mat"
Hidden states: [h₁, h₂, h₃, h₄, h₅, h₆]

When translating to "El gato":
- High attention on h₁ ("The") and h₂ ("cat")
- Low attention on other positions
```

5.4 LSTM with Dropout

Dropout Variants

1. Standard Dropout: Applied to non-recurrent connections

```python
x_t = dropout(x_t, rate=0.2)
h_t = LSTM(x_t, h_{t-1})
```

2. Recurrent Dropout: Applied to recurrent connections

```python
h_{t-1} = recurrent_dropout(h_{t-1}, rate=0.2)
h_t = LSTM(x_t, h_{t-1})
```

3. Zoneout: Randomly keeps some hidden units unchanged

```python
mask = random_binary_mask(rate=0.1)
h_t = mask * h_{t-1} + (1 - mask) * h_t_new
```

---

6. Detailed Comparisons {#comparisons}

6.1 Performance Comparison

Computational Complexity

Model	Parameters	Training Time	Memory Usage	Inference Speed
RNN	$O(W^2)$	Fast	Low	Fastest
LSTM	$O(4W^2)$	Slow	High	Slow
GRU	$O(3W^2)$	Medium	Medium	Medium
Advanced LSTM	$O(8W^2+)$	Slowest	Highest	Slowest

Where W = hidden_size + input_size

Memory Requirements (Exact Calculations)

```
Given: input_size = 100, hidden_size = 128, sequence_length = 200

RNN Memory:
- Weights: (100 + 128) × 128 = 29,184 parameters
- Hidden states: 200 × 128 = 25,600 values
- Total: ~54K values

LSTM Memory:
- Weights: 4 × (100 + 128 + 1) × 128 = 117,504 parameters
- Hidden + Cell states: 200 × 128 × 2 = 51,200 values
- Gate activations: 200 × 128 × 4 = 102,400 values
- Total: ~271K values

GRU Memory:
- Weights: 3 × (100 + 128 + 1) × 128 = 88,128 parameters
- Hidden states: 200 × 128 = 25,600 values
- Gate activations: 200 × 128 × 3 = 76,800 values
- Total: ~190K values
```

6.2 Accuracy Comparison by Task

Task 1: Sentiment Analysis (IMDB)

```
Dataset: 50,000 movie reviews
Metric: Accuracy

Results:
- Simple RNN:     0.831 ± 0.012
- LSTM:           0.871 ± 0.008
- GRU:            0.867 ± 0.009
- Bidirectional LSTM: 0.884 ± 0.006
- Stacked LSTM:   0.892 ± 0.007
```

Task 2: Language Modeling (Penn Treebank)

```
Dataset: Penn Treebank corpus
Metric: Perplexity (lower is better)

Results:
- Simple RNN:     165.2
- LSTM:           78.4
- GRU:            81.9
- Advanced LSTM:  68.7
```

Task 3: Machine Translation (WMT14)

```
Dataset: English-German translation
Metric: BLEU Score (higher is better)

Results:
- LSTM:                   24.8
- Bidirectional LSTM:     26.3
- LSTM + Attention:       28.1
- Stacked LSTM + Attention: 29.7
```

6.3 Training Characteristics

Convergence Speed

```
Epochs to 90% of final performance:

Simple RNN:    15-20 epochs (but lower final performance)
LSTM:          25-30 epochs
GRU:           20-25 epochs
Advanced LSTM: 35-45 epochs (but higher final performance)
```

Gradient Stability

```
Gradient Norm Statistics over Training:

RNN:           High variance, frequent exploding/vanishing
LSTM:          Stable, consistent gradients
GRU:           Stable, slightly more variance than LSTM
Advanced LSTM: Very stable, slower but consistent
```

6.4 Sequence Length Handling

Performance vs Sequence Length

```
Sequence Length: 50    100   200   500   1000
RNN Accuracy:    0.85  0.78  0.65  0.45  0.30
LSTM Accuracy:   0.87  0.86  0.84  0.79  0.72
GRU Accuracy:    0.86  0.85  0.83  0.78  0.70
Adv LSTM Acc:    0.89  0.88  0.87  0.83  0.78
```

6.5 Architecture Decision Matrix

Criteria	RNN	LSTM	GRU	Advanced LSTM
Short sequences (<50)	✅ Good	✅ Good	✅ Good	⚠️ Overkill
Medium sequences (50-200)	⚠️ Limited	✅ Excellent	✅ Excellent	✅ Best
Long sequences (>200)	❌ Poor	✅ Good	✅ Good	✅ Excellent
Limited compute	✅ Best	⚠️ Moderate	✅ Good	❌ Poor
High accuracy needed	❌ Limited	✅ Good	✅ Good	✅ Best
Real-time inference	✅ Fastest	⚠️ Slow	✅ Moderate	❌ Slowest
Small datasets	✅ Good	⚠️ Overfitting	✅ Good	❌ Overfitting
Large datasets	⚠️ Limited	✅ Excellent	✅ Excellent	✅ Best

---

7. Implementation Examples {#examples}

7.1 Simple RNN Implementation

PyTorch Implementation

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

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(SimpleRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # RNN layer
        self.rnn = nn.RNN(input_size, hidden_size, num_layers,
                          batch_first=True)

        # Output layer
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Initialize hidden state
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)

        # Forward propagate RNN
        out, _ = self.rnn(x, h0)

        # Use the last output for classification
        out = self.fc(out[:, -1, :])
        return out

# Usage example
model = SimpleRNN(input_size=100, hidden_size=128, output_size=2)
input_tensor = torch.randn(32, 50, 100)  # (batch, seq_len, features)
output = model(input_tensor)
print(f"Output shape: {output.shape}")  # [32, 2]
```

TensorFlow/Keras Implementation

```python
```

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Dropout

def create_simple_rnn(input_shape, hidden_units, output_units):
    model = Sequential([
        SimpleRNN(hidden_units,
                  return_sequences=False,
                  dropout=0.2,
                  recurrent_dropout=0.2,
                  input_shape=input_shape),
        Dense(64, activation='relu'),
        Dropout(0.5),
        Dense(output_units, activation='sigmoid')
    ])

    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Usage
model = create_simple_rnn((200, 100), 128, 1)
model.summary()
```

7.2 LSTM Implementation

Detailed LSTM with Custom Forward Pass

```
python
```

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

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

        # Gate parameters
        self.W_f = nn.Linear(input_size + hidden_size, hidden_size)  # Forget gate
        self.W_i = nn.Linear(input_size + hidden_size, hidden_size)  # Input gate
        self.W_C = nn.Linear(input_size + hidden_size, hidden_size)  # Candidate
        self.W_o = nn.Linear(input_size + hidden_size, hidden_size)  # Output gate

    def forward(self, x, hidden=None):
        batch_size, seq_len, _ = x.size()

        if hidden is None:
            h_t = torch.zeros(batch_size, self.hidden_size)
            C_t = torch.zeros(batch_size, self.hidden_size)
        else:
            h_t, C_t = hidden

        outputs = []

        for t in range(seq_len):
            x_t = x[:, t, :]

            # Concatenate input and hidden state
            combined = torch.cat([x_t, h_t], dim=1)

            # Gate computations
            f_t = torch.sigmoid(self.W_f(combined))    # Forget gate
            i_t = torch.sigmoid(self.W_i(combined))    # Input gate
            C_tilde_t = torch.tanh(self.W_C(combined)) # Candidate values
            o_t = torch.sigmoid(self.W_o(combined))    # Output gate

            # Update cell state
            C_t = f_t * C_t + i_t * C_tilde_t

            # Update hidden state
```

```python
            h_t = o_t * torch.tanh(C_t)

            outputs.append(h_t.unsqueeze(1))

        outputs = torch.cat(outputs, dim=1)
        return outputs, (h_t, C_t)

# Advanced LSTM for Sentiment Analysis
class SentimentLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_layers, dropout=0.5):
        super(SentimentLSTM, self).__init__()

        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # LSTM layers
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers,
                            dropout=dropout, batch_first=True,
                            bidirectional=True)

        # Attention mechanism
        self.attention = nn.Linear(hidden_dim * 2, 1)

        # Classification layers
        self.fc1 = nn.Linear(hidden_dim * 2, hidden_dim)
        self.dropout = nn.Dropout(dropout)
        self.fc2 = nn.Linear(hidden_dim, 1)

    def attention_mechanism(self, lstm_out):
        # lstm_out: (batch_size, seq_len, hidden_dim * 2)
        attention_weights = F.softmax(self.attention(lstm_out).squeeze(-1), dim=1)
        # attention_weights: (batch_size, seq_len)

        # Apply attention weights
        context_vector = torch.sum(lstm_out * attention_weights.unsqueeze(-1), dim=1)
        return context_vector, attention_weights

    def forward(self, x):
        # Embedding
        embedded = self.embedding(x)

        # LSTM
        lstm_out, _ = self.lstm(embedded)
```

```python
        # Attention
        context_vector, attention_weights = self.attention_mechanism(lstm_out)

        # Classification
        out = F.relu(self.fc1(context_vector))
        out = self.dropout(out)
        out = torch.sigmoid(self.fc2(out))

        return out, attention_weights

# Usage example with attention visualization
def train_with_attention_visualization():
    model = SentimentLSTM(vocab_size=10000, embedding_dim=128,
                          hidden_dim=64, num_layers=2)

    # Sample input
    sample_input = torch.randint
```