Complete RNN Guide: From Theory to Implementation

Table of Contents

---

1. How RNN Works - Complete Mathematical Foundation {#rnn-theory}

1.1 The Core Concept

Traditional Neural Network Problem:

```
Input: [word1, word2, word3, word4]
Processing: Each word processed independently
Problem: No memory of previous words
```

RNN Solution:

```
h₀ → h₁ → h₂ → h₃ → h₄
↑    ↑    ↑    ↑    ↑
x₁   x₂   x₃   x₄   output
```

1.2 Mathematical Foundation

Core RNN Equations

```
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 (the "memory")

- `x_t`: Input at time t
- `y_t`: Output at time t
- `W_hh`: Hidden-to-hidden weight matrix (memory transformation)
- `W_xh`: Input-to-hidden weight matrix (input processing)
- `W_hy`: Hidden-to-output weight matrix (output generation)
- `b_h`, `b_y`: Bias vectors

1.3 Detailed Step-by-Step Processing

Let's trace through a complete example: **Sentiment Analysis of "This movie rocks"**

Step 0: Initialization

```python
# Assume we have:
vocab_size = 10000
hidden_size = 4
embedding_dim = 3

# Initialize hidden state
h_0 = [0.0, 0.0, 0.0, 0.0]  # Zero vector

# Weight matrices (randomly initialized)
W_hh = [[0.1, -0.2, 0.3, 0.4],   # 4x4 matrix
        [0.2, 0.1, -0.1, 0.3],
        [-0.3, 0.4, 0.2, -0.1],
        [0.1, 0.2, 0.3, -0.2]]

W_xh = [[0.5, -0.1, 0.2],        # 4x3 matrix
        [0.3, 0.4, -0.2],
        [-0.1, 0.2, 0.3],
        [0.2, -0.3, 0.1]]

b_h = [0.1, -0.1, 0.0, 0.2]      # Bias vector
```

Step 1: Process "This"

```python
```

```python
# Word "This" → embedding
x_1 = [0.2, -0.1, 0.3]  # 3-dimensional embedding

# Compute hidden state
# h_1 = tanh(W_hh × h_0 + W_xh × x_1 + b_h)

# Matrix multiplication W_hh × h_0
Whh_h0 = [0.0, 0.0, 0.0, 0.0]  # Since h_0 is zeros

# Matrix multiplication W_xh × x_1
Wxh_x1 = [0.5×0.2 + (-0.1)×(-0.1) + 0.2×0.3,    # = 0.17
         0.3×0.2 + 0.4×(-0.1) + (-0.2)×0.3,     # = -0.04
         (-0.1)×0.2 + 0.2×(-0.1) + 0.3×0.3,     # = 0.05
         0.2×0.2 + (-0.3)×(-0.1) + 0.1×0.3]     # = 0.10

# Add bias
pre_activation = [0.17 + 0.1, -0.04 + (-0.1), 0.05 + 0.0, 0.10 + 0.2]
              = [0.27, -0.14, 0.05, 0.30]

# Apply tanh activation
h_1 = [tanh(0.27), tanh(-0.14), tanh(0.05), tanh(0.30)]
    = [0.264, -0.139, 0.050, 0.291]
```

Step 2: Process "movie"

```python
```

```
# Word "movie" → embedding
x_2 = [0.4, 0.1, -0.2]

# Now we have previous hidden state h_1 = [0.264, -0.139, 0.050, 0.291]

# Matrix multiplication W_hh × h_1
Whh_h1 = [0.1×0.264 + (-0.2)×(-0.139) + 0.3×0.050 + 0.4×0.291,   # = 0.179
        0.2×0.264 + 0.1×(-0.139) + (-0.1)×0.050 + 0.3×0.291,     # = 0.126
        (-0.3)×0.264 + 0.4×(-0.139) + 0.2×0.050 + (-0.1)×0.291,  # = -0.165
        0.1×0.264 + 0.2×(-0.139) + 0.3×0.050 + (-0.2)×0.291]     # = -0.028

# Matrix multiplication W_xh × x_2
Wxh_x2 = [0.5×0.4 + (-0.1)×0.1 + 0.2×(-0.2),     # = 0.15
        0.3×0.4 + 0.4×0.1 + (-0.2)×(-0.2),       # = 0.20
        (-0.1)×0.4 + 0.2×0.1 + 0.3×(-0.2),       # = -0.08
        0.2×0.4 + (-0.3)×0.1 + 0.1×(-0.2)]       # = 0.03

# Combine and add bias
pre_activation = [0.179 + 0.15 + 0.1,    # = 0.429
              0.126 + 0.20 + (-0.1), # = 0.226
              -0.165 + (-0.08) + 0.0, # = -0.245
              -0.028 + 0.03 + 0.2]    # = 0.202

# Apply tanh
h_2 = [tanh(0.429), tanh(0.226), tanh(-0.245), tanh(0.202)]
   = [0.406, 0.222, -0.240, 0.199]
```

Step 3: Process "rocks"

```python
# Similar process continues...
# The hidden state h_2 now contains information about "This movie"
# When processing "rocks", the network can use this context
```

1.4 Why This Creates Memory

Key Insight: Each hidden state `h_t` contains:

1. Information from current input `x_t`
2. Information from previous hidden state `h_{t-1}`
3. Which transitively contains information from all previous inputs

Information Flow:

```
h_1 contains: "This"
h_2 contains: "This" + "movie" (combined through h_1)
h_3 contains: "This" + "movie" + "rocks" (combined through h_2)
```

1.5 The Vanishing Gradient Problem - Mathematical Explanation

During backpropagation, gradients flow backward through time:

```python
# Gradient computation (simplified)
∂L/∂h_1 = ∂L/∂h_3 × ∂h_3/∂h_2 × ∂h_2/∂h_1

# Each ∂h_t/∂h_{t-1} involves:
∂h_t/∂h_{t-1} = W_hh × diag(tanh'(pre_activation))

# Since tanh'(x) ≤ 1, and we multiply many such terms:
# Gradient diminishes exponentially with sequence length
```

Result: Earlier time steps receive very small gradients and learn slowly.

---

2. Complete RNN Implementation Guide {#implementation}

2.1 Environment Setup

```python
```

```python
# Required libraries
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns
from collections import Counter
import re
import pickle
```

2.2 Custom RNN Implementation (From Scratch)

python

```
class RNNFromScratch:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights (Xavier initialization)
        self.W_hh = np.random.randn(hidden_size, hidden_size) * np.sqrt(1.0 / hidden_size)
        self.W_xh = np.random.randn(hidden_size, input_size) * np.sqrt(1.0 / input_size)
        self.W_hy = np.random.randn(output_size, hidden_size) * np.sqrt(1.0 / hidden_size)

        # Initialize biases
        self.b_h = np.zeros((hidden_size, 1))
        self.b_y = np.zeros((output_size, 1))

    def forward(self, inputs):
        """
        Forward pass through RNN
        inputs: list of input vectors (sequence_length, input_size)
        returns: outputs, hidden_states
        """
        h = np.zeros((self.hidden_size, 1))  # Initial hidden state
        outputs = []
        hidden_states = [h.copy()]

        for x in inputs:
            x = x.reshape(-1, 1)  # Ensure column vector

            # RNN forward step
            h = np.tanh(np.dot(self.W_hh, h) + np.dot(self.W_xh, x) + self.b_h)
            y = np.dot(self.W_hy, h) + self.b_y

            outputs.append(y)
            hidden_states.append(h.copy())

        return outputs, hidden_states

    def backward(self, inputs, targets, outputs, hidden_states):
        """
        Backward pass (Backpropagation Through Time)
        """
        # Initialize gradients
```

```
    dW_hh = np.zeros_like(self.W_hh)
    dW_xh = np.zeros_like(self.W_xh)
    dW_hy = np.zeros_like(self.W_hy)
    db_h = np.zeros_like(self.b_h)
    db_y = np.zeros_like(self.b_y)

    dh_next = np.zeros_like(hidden_states[0])

    # Backward through time
    for t in reversed(range(len(inputs))):
        x = inputs[t].reshape(-1, 1)
        h = hidden_states[t + 1]
        h_prev = hidden_states[t]
        y = outputs[t]
        target = targets[t].reshape(-1, 1)

        # Output layer gradients
        dy = y - target  # Assuming MSE loss
        dW_hy += np.dot(dy, h.T)
        db_y += dy

        # Hidden layer gradients
        dh = np.dot(self.W_hy.T, dy) + dh_next
        dh_raw = dh * (1 - h * h)  # tanh derivative

        # Weight gradients
        dW_hh += np.dot(dh_raw, h_prev.T)
        dW_xh += np.dot(dh_raw, x.T)
        db_h += dh_raw

        # Gradient for next iteration
        dh_next = np.dot(self.W_hh.T, dh_raw)

    # Update weights
    self.W_hh -= self.learning_rate * dW_hh
    self.W_xh -= self.learning_rate * dW_xh
    self.W_hy -= self.learning_rate * dW_hy
    self.b_h -= self.learning_rate * db_h
    self.b_y -= self.learning_rate * db_y
```

---

3. Data Preparation Pipeline {#data-preparation}

3.1 Text Preprocessing Class

python

```python
class TextPreprocessor:
    def __init__(self, max_vocab_size=10000, max_sequence_length=100):
        self.max_vocab_size = max_vocab_size
        self.max_sequence_length = max_sequence_length
        self.word_to_idx = {'<PAD>': 0, '<UNK>': 1, '<START>': 2, '<END>': 3}
        self.idx_to_word = {0: '<PAD>', 1: '<UNK>', 2: '<START>', 3: '<END>'}
        self.vocab_size = 4

    def clean_text(self, text):
        """Clean and normalize text"""
        # Convert to lowercase
        text = text.lower()

        # Remove special characters except spaces and basic punctuation
        text = re.sub(r'[^a-zA-Z0-9\s\.\!\?\,]', '', text)

        # Remove extra whitespace
        text = re.sub(r'\s+', ' ', text).strip()

        return text

    def build_vocabulary(self, texts):
        """Build vocabulary from training texts"""
        print("Building vocabulary...")

        # Clean all texts
        cleaned_texts = [self.clean_text(text) for text in texts]

        # Count word frequencies
        word_counts = Counter()
        for text in cleaned_texts:
            words = text.split()
            word_counts.update(words)

        # Select most frequent words
        most_common = word_counts.most_common(self.max_vocab_size - 4)  # -4 for special tokens

        # Add to vocabulary
        for word, count in most_common:
            if word not in self.word_to_idx:
                self.word_to_idx[word] = self.vocab_size
                self.idx_to_word[self.vocab_size] = word
                self.vocab_size += 1
```

```
        print(f"Vocabulary size: {self.vocab_size}")
        return cleaned_texts

    def text_to_sequence(self, text, add_special_tokens=True):
        """Convert text to sequence of indices"""
        words = self.clean_text(text).split()

        # Add special tokens
        if add_special_tokens:
            words = ['<START>'] + words + ['<END>']

        # Convert to indices
        sequence = []
        for word in words:
            if word in self.word_to_idx:
                sequence.append(self.word_to_idx[word])
            else:
                sequence.append(self.word_to_idx['<UNK>'])

        return sequence

    def pad_sequence(self, sequence, max_length=None):
        """Pad or truncate sequence to fixed length"""
        if max_length is None:
            max_length = self.max_sequence_length

        if len(sequence) > max_length:
            return sequence[:max_length]
        else:
            return sequence + [self.word_to_idx['<PAD>']] * (max_length - len(sequence))

    def sequences_to_padded_batch(self, sequences):
        """Convert list of sequences to padded batch"""
        padded_sequences = [self.pad_sequence(seq) for seq in sequences]
        return np.array(padded_sequences)

# Example usage
preprocessor = TextPreprocessor(max_vocab_size=5000, max_sequence_length=50)

# Sample data
texts = [
    "This movie is really great!",
    "I love this film so much",
```

```
    "Terrible movie, waste of time",
    "Amazing acting and great story"
]
labels = [1, 1, 0, 1]  # 1 = positive, 0 = negative

# Build vocabulary
cleaned_texts = preprocessor.build_vocabulary(texts)

# Convert to sequences
sequences = [preprocessor.text_to_sequence(text) for text in cleaned_texts]
print("Sample sequences:", sequences[:2])

# Pad sequences
padded_sequences = preprocessor.sequences_to_padded_batch(sequences)
print("Padded batch shape:", padded_sequences.shape)
```

3.2 Dataset Class for PyTorch

```python
```

```python
class SentimentDataset(Dataset):
    def __init__(self, sequences, labels, preprocessor):
        self.sequences = sequences
        self.labels = labels
        self.preprocessor = preprocessor

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        sequence = torch.LongTensor(self.sequences[idx])
        label = torch.FloatTensor([self.labels[idx]])
        return sequence, label

    def collate_fn(self, batch):
        """Custom collate function for DataLoader"""
        sequences, labels = zip(*batch)

        # Pad sequences to same length in batch
        max_len = max(len(seq) for seq in sequences)
        padded_sequences = []

        for seq in sequences:
            if len(seq) < max_len:
                padded = torch.cat([seq, torch.zeros(max_len - len(seq), dtype=torch.long)])
            else:
                padded = seq
            padded_sequences.append(padded)

        return torch.stack(padded_sequences), torch.stack(labels)
```

---

4. RNN Model Architecture & Parameters {#parameters}

4.1 Complete RNN Model with All Options

```python
```

```
class ComprehensiveRNN(nn.Module):
    def __init__(self,
                 vocab_size,          # Size of vocabulary
                 embedding_dim,       # Dimension of word embeddings
                 hidden_size,         # Size of hidden state
                 output_size,         # Number of output classes
                 num_layers=1,        # Number of RNN layers
                 dropout=0.5,         # Dropout rate
                 bidirectional=False, # Whether to use bidirectional RNN
                 rnn_type='RNN',      # Type: 'RNN', 'LSTM', 'GRU'
                 use_attention=False): # Whether to use attention mechanism

        super(ComprehensiveRNN, self).__init__()

        # Store parameters
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bidirectional = bidirectional
        self.use_attention = use_attention

        # Calculate actual hidden size considering bidirectional
        self.actual_hidden_size = hidden_size * (2 if bidirectional else 1)

        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)

        # RNN layer
        if rnn_type == 'RNN':
            self.rnn = nn.RNN(embedding_dim, hidden_size, num_layers,
                              dropout=dropout if num_layers > 1 else 0,
                              bidirectional=bidirectional, batch_first=True)
        elif rnn_type == 'LSTM':
            self.rnn = nn.LSTM(embedding_dim, hidden_size, num_layers,
                               dropout=dropout if num_layers > 1 else 0,
                               bidirectional=bidirectional, batch_first=True)
        elif rnn_type == 'GRU':
            self.rnn = nn.GRU(embedding_dim, hidden_size, num_layers,
                              dropout=dropout if num_layers > 1 else 0,
                              bidirectional=bidirectional, batch_first=True)

        # Attention mechanism
        if use_attention:
            self.attention = nn.Linear(self.actual_hidden_size, 1, bias=False)
```

```python
        # Classification layers
        self.dropout = nn.Dropout(dropout)
        self.fc1 = nn.Linear(self.actual_hidden_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def attention_mechanism(self, rnn_outputs, lengths):
        """
        Apply attention mechanism to RNN outputs
        rnn_outputs: (batch_size, seq_len, hidden_size)
        lengths: actual sequence lengths for each sample
        """
        # Calculate attention scores
        attention_scores = self.attention(rnn_outputs).squeeze(-1)  # (batch_size, seq_len)

        # Create mask for padding tokens
        batch_size, max_len = attention_scores.size()
        mask = torch.arange(max_len).unsqueeze(0).expand(batch_size, -1) < lengths.unsqueeze(1)

        # Apply mask (set padding positions to -inf)
        attention_scores.masked_fill_(~mask, float('-inf'))

        # Apply softmax
        attention_weights = F.softmax(attention_scores, dim=1)

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

        return context_vector, attention_weights

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

        # Embedding
        embedded = self.embedding(x)  # (batch_size, seq_len, embedding_dim)

        # RNN forward pass
        rnn_out, _ = self.rnn(embedded)  # (batch_size, seq_len, hidden_size * num_directions)

        if self.use_attention and lengths is not None:
            # Use attention mechanism
            context_vector, attention_weights = self.attention_mechanism(rnn_out, lengths)
        else:
            # Use last hidden state (considering padding)
```

```python
        if lengths is not None:
            # Get the last non-padded hidden state for each sequence
            idx = (lengths - 1).unsqueeze(1).unsqueeze(2).expand(-1, -1, rnn_out.size(2))
            context_vector = rnn_out.gather(1, idx).squeeze(1)
        else:
            # Use the last hidden state
            context_vector = rnn_out[:, -1, :]
        attention_weights = None

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

    if self.use_attention:
        return out, attention_weights
    return out

def init_weights(self):
    """Initialize model weights"""
    for name, param in self.named_parameters():
        if 'weight' in name:
            if len(param.shape) >= 2:
                nn.init.xavier_uniform_(param)
            else:
                nn.init.uniform_(param, -0.1, 0.1)
        elif 'bias' in name:
            nn.init.constant_(param, 0)
```

4.2 Parameter Analysis and Guidelines

```python
```

```
def analyze_model_parameters(model):
    """Analyze model parameters and memory usage"""
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

    print("="*50)
    print("MODEL PARAMETER ANALYSIS")
    print("="*50)

    # Parameter breakdown by layer
    for name, module in model.named_modules():
        if len(list(module.parameters())) > 0:
            module_params = sum(p.numel() for p in module.parameters())
            print(f"{name:20s}: {module_params:,} parameters")

    print(f"\nTotal parameters: {total_params:,}")
    print(f"Trainable parameters: {trainable_params:,}")

    # Memory estimation (rough)
    memory_mb = total_params * 4 / (1024 * 1024)  # 4 bytes per float32
    print(f"Estimated memory usage: {memory_mb:.2f} MB")

    return total_params, trainable_params

# Parameter selection guidelines
def suggest_parameters(vocab_size, avg_sequence_length, dataset_size):
    """Suggest model parameters based on data characteristics"""
    print("="*50)
    print("PARAMETER SUGGESTIONS")
    print("="*50)

    # Embedding dimension
    if vocab_size < 1000:
        embedding_dim = 64
    elif vocab_size < 10000:
        embedding_dim = 128
    else:
        embedding_dim = 256

    # Hidden size
    if dataset_size < 1000:
        hidden_size = 32
    elif dataset_size < 10000:
```

```python
        hidden_size = 64
    else:
        hidden_size = 128

    # Number of layers
    if avg_sequence_length < 50:
        num_layers = 1
    elif avg_sequence_length < 200:
        num_layers = 2
    else:
        num_layers = 3

    # Dropout
    if dataset_size < 1000:
        dropout = 0.3
    else:
        dropout = 0.5

    print(f"Vocabulary size: {vocab_size}")
    print(f"Average sequence length: {avg_sequence_length}")
    print(f"Dataset size: {dataset_size}")
    print(f"\nSuggested parameters:")
    print(f"  embedding_dim: {embedding_dim}")
    print(f"  hidden_size: {hidden_size}")
    print(f"  num_layers: {num_layers}")
    print(f"  dropout: {dropout}")

    return {
        'embedding_dim': embedding_dim,
        'hidden_size': hidden_size,
        'num_layers': num_layers,
        'dropout': dropout
    }
```

---

5. Training Process {#training}

5.1 Complete Training Loop

```python
```

```python
class RNNTrainer:
    def __init__(self, model, train_loader, val_loader, device='cpu'):
        self.model = model.to(device)
        self.train_loader = train_loader
        self.val_loader = val_loader
        self.device = device

        # Training history
        self.train_losses = []
        self.train_accuracies = []
        self.val_losses = []
        self.val_accuracies = []

    def train_epoch(self, optimizer, criterion):
        """Train for one epoch"""
        self.model.train()
        total_loss = 0
        correct = 0
        total = 0

        for batch_idx, (sequences, labels) in enumerate(self.train_loader):
            sequences, labels = sequences.to(self.device), labels.to(self.device)

            # Calculate actual sequence lengths (for attention/masking)
            lengths = (sequences != 0).sum(dim=1)

            # Forward pass
            optimizer.zero_grad()

            if hasattr(self.model, 'use_attention') and self.model.use_attention:
                outputs, attention_weights = self.model(sequences, lengths)
            else:
                outputs = self.model(sequences, lengths)

            # Calculate loss
            loss = criterion(outputs.squeeze(), labels.squeeze())

            # Backward pass
            loss.backward()

            # Gradient clipping (important for RNNs)
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=5.0)
```

```
            optimizer.step()

            # Statistics
            total_loss += loss.item()
            predicted = (torch.sigmoid(outputs) > 0.5).float()
            total += labels.size(0)
            correct += (predicted.squeeze() == labels.squeeze()).sum().item()

            # Print progress
            if batch_idx % 10 == 0:
                print(f'Batch {batch_idx}/{len(self.train_loader)}, '
                      f'Loss: {loss.item():.4f}, '
                      f'Accuracy: {100.*correct/total:.2f}%')

        avg_loss = total_loss / len(self.train_loader)
        accuracy = 100. * correct / total

        return avg_loss, accuracy

    def validate(self, criterion):
        """Validate the model"""
        self.model.eval()
        total_loss = 0
        correct = 0
        total = 0

        with torch.no_grad():
            for sequences, labels in self.val_loader:
                sequences, labels = sequences.to(self.device), labels.to(self.device)
                lengths = (sequences != 0).sum(dim=1)

                if hasattr(self.model, 'use_attention') and self.model.use_attention:
                    outputs, _ = self.model(sequences, lengths)
                else:
                    outputs = self.model(sequences, lengths)

                loss = criterion(outputs.squeeze(), labels.squeeze())
                total_loss += loss.item()

                predicted = (torch.sigmoid(outputs) > 0.5).float()
                total += labels.size(0)
                correct += (predicted.squeeze() == labels.squeeze()).sum().item()

        avg_loss = total_loss / len(self.val_loader)
```

```
        accuracy = 100. * correct / total

        return avg_loss, accuracy

    def train(self, num_epochs, learning_rate=0.001, weight_decay=1e-5):
        """Complete training process"""
        print("="*50)
        print("STARTING TRAINING")
        print("="*50)

        # Optimizer and loss function
        optimizer = torch.optim.Adam(self.model.parameters(),
                                     lr=learning_rate,
                                     weight_decay=weight_decay)
        criterion = nn.BCEWithLogitsLoss()

        # Learning rate scheduler
        scheduler = torch.optim.lr_scheduler.
```