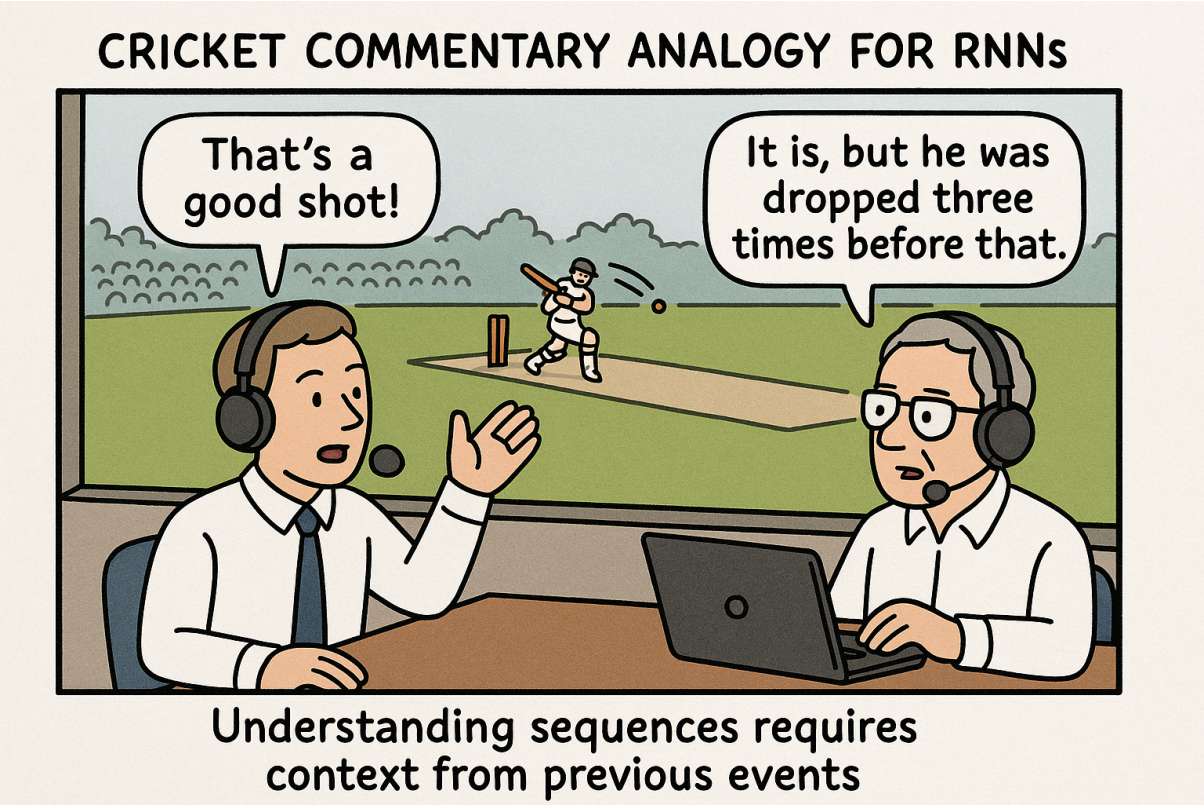
Recurrent Neural Networks (RNNs)

Minor in AI

May 2, 2025



Introduction to Sequence Modeling and RNNs

Imagine you're watching a cricket match commentary. The commentator doesn't describe each ball in isolation - they build upon what happened in previous balls to create a coherent narrative. "Virat Kohli took a single last ball, and now Rohit Sharma faces a full toss..." This sequential understanding is exactly what Recurrent Neural Networks (RNNs) bring to artificial intelligence. Just as humans understand language, music, or stock markets by connecting current information with past context, RNNs maintain a "memory" of previous inputs to make sense of sequential data.

Traditional neural networks treat each input independently, which works well for isolated images or data points but fails miserably when dealing with sequences where context matters. This limitation became painfully apparent in our class example of named entity recognition (identifying names, places, organizations in text), where we initially tried using a standard Multi-Layer Perceptron (MLP) approach.

The Problem with Traditional Approaches

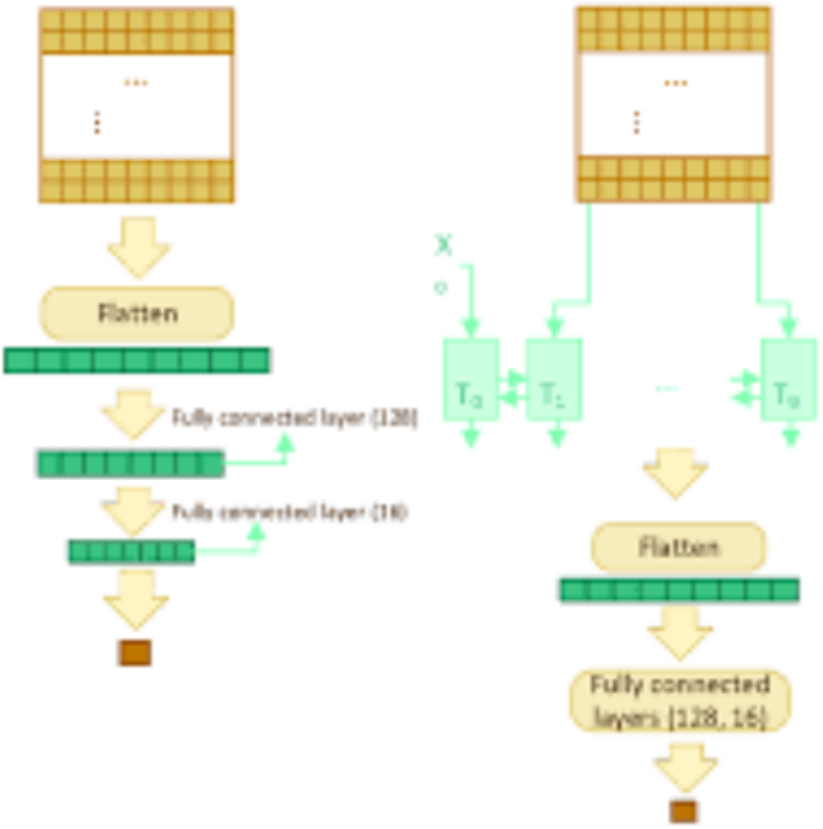


Figure 1: Comparison between MLP (left) and RNN (right) architectures for sequence processing

In our classroom session, we began with a simple sentence: "Virat Kohli and Rohit Sharma play cricket for India." When attempting named entity recognition (identifying Virat Kohli, Rohit Sharma, and India as special entities), our first approach using a standard MLP revealed two critical flaws:

1. **Fixed Input Size Problem**: MLPs require fixed-size inputs. Sentences vary in length, forcing us to either truncate long sentences or pad short ones with meaningless zeros, both inefficient solutions.

2. **Lack of Sequential Context**: Each word was processed independently without considering its relationship to neighboring words. The network couldn't learn that "Virat Kohli" is a person's name because it treated each word in isolation.

These limitations mirror how ineffective it would be if a cricket commentator described each ball without any reference to the match situation, score, or previous deliveries. The commentary would make no sense because cricket, like language, is inherently sequential and contextual.

Recurrent Neural Networks: The Conceptual Breakthrough

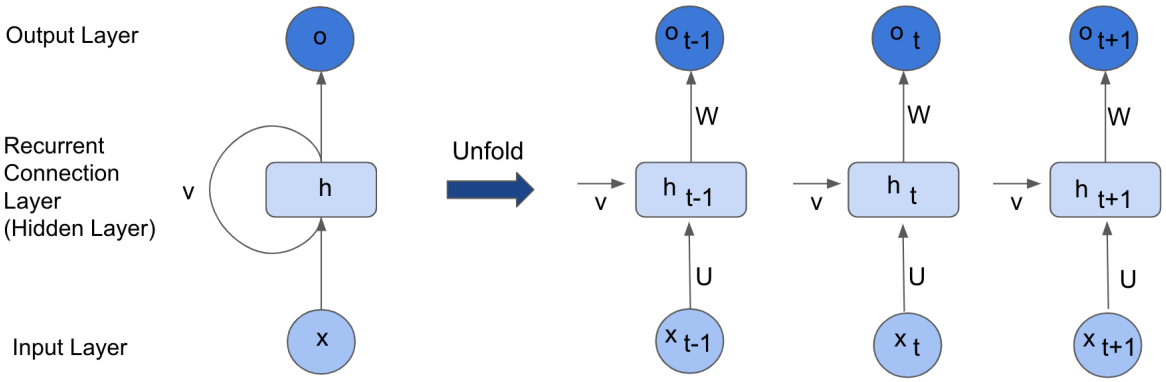


Figure 2: Unrolled RNN architecture showing the flow of hidden states through time

RNNs solve these problems by introducing the concept of memory - each step in the sequence maintains information about what came before it. Think of it like reading a book where you remember characters and plot points from previous chapters to understand the current page.

The key innovation in RNNs is the recurrent connection - the hidden state that gets passed from one time step to the next. In our classroom example, when processing "Rohit Sharma," the network remembers it recently saw "Virat Kohli and," helping it recognize this as another player's name rather than random words.

Detailed Architecture of RNNs

The Basic Building Block

An RNN can be visualized as a chain of connected neural networks, where each link in the chain represents processing at one time step (one word in our sentence example). Each unit has:

- **Input (x_t)**: The current element in the sequence (e.g., the word "Kohli")
- **Hidden State (h_t or a_t)**: The "memory" containing information about previous inputs
- **Output (y_t)**: The prediction at the current step

The magic happens through the recurrent connection where the hidden state flows from one time step to the next, creating a form of memory.

Mathematical Formulation

Let's break down the mathematics that makes this work, as explained in the classroom session:

1. **Hidden State Calculation**:

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

Where:

- h_t: Current hidden state
- h_{t-1}: Previous hidden state
- W_{hh}: Weight matrix for hidden-to-hidden connections
- W_{xh}: Weight matrix for input-to-hidden connections
- x_t: Current input
- b_h: Hidden layer bias term
- σ: Activation function (often tanh)

2. **Output Calculation**:

$$y_t = \text{softmax}(W_{hy} \cdot h_t + b_y) \tag{2}$$

Where:

- W_{hy}: Weight matrix for hidden-to-output connections
- b_y: Output layer bias term

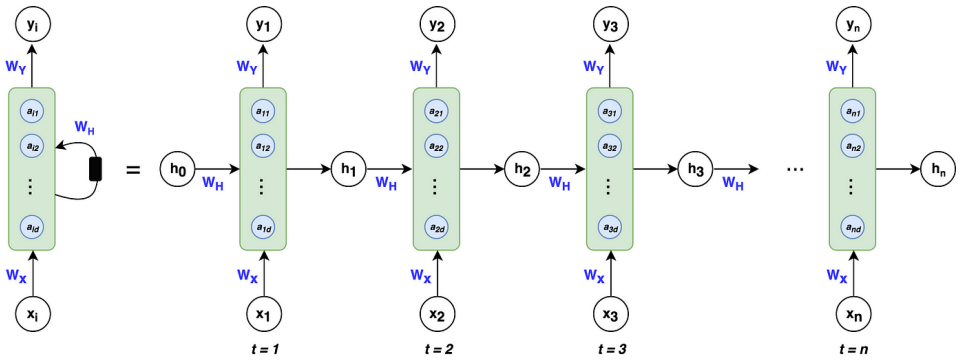


Figure 3: Visual representation of the RNN equations showing the flow of computations

In our named entity recognition example, these equations would work as follows when processing "Virat Kohli":

1. For "Virat":
   - $h_1 = \tanh(W_{hh} \cdot h_0 + W_{xh} \cdot x_1 + b_h)$
   - $y_1 = \sigma(W_{hy} \cdot h_1 + b_y) \rightarrow$ Predict if "Virat" is part of a named entity
2. For "Kohli":
   - $h_2 = \tanh(W_{hh} \cdot h_1 + W_{xh} \cdot x_2 + b_h) \leftarrow$ Notice h_1 is used here
   - $y_2 = \sigma(W_{hy} \cdot h_2 + b_y) \rightarrow$ Predict if "Kohli" is part of a named entity

The critical difference from MLPs is the $W_{hh} \cdot h_{t-1}$ term, which carries forward information from previous steps.

Dimensionality and Weight Matrices

In the classroom, we discussed concrete dimensions to understand how these matrices interact:

1. Assume:
   - Input size (vocabulary): 10,000 words (one-hot encoded)
   - Hidden layer size: 100 neurons
2. Then:
   - W_{xh}: 100 × 10,000 matrix (input to hidden)
   - W_{hh}: 100 × 100 matrix (hidden to hidden)
   - W_{hy}: 1 × 100 matrix (hidden to output for binary classification)

When we process the first word "Virat":

- x_1 is 10,000 × 1 one-hot vector (all zeros except one position)
- $W_{xh} \cdot x_1$ becomes a 100 × 1 vector
- h_0 (initial hidden state) is typically zeros or random, 100 × 1
- $W_{hh} \cdot h_0$ is another 100 × 1 vector
- The sum produces a 100 × 1 hidden state

This dimensional analysis helps understand how information flows through the network at each step.

Different RNN Architectures for Different Tasks

Our classroom discussion revealed that not all sequence tasks are created equal. Just as different cricket formats (Test, ODI, T20) require different strategies, different NLP tasks need different RNN architectures. The choice of architecture depends on the input-output relationship and the temporal nature of the task. Below we examine each architecture in detail with mathematical formulations and practical considerations.

1. One-to-One (Vanilla)

- **Input-Output Relationship**: Single fixed-size input to single output
- **Mathematical Formulation**:

$$y = f(x) \tag{3}$$

  where f is typically a feedforward neural network
- **When to Use**: When there's no sequential relationship in the data
- **Training Considerations**:
  - Standard backpropagation
  - No temporal dependencies to consider
- **Example Applications**:
  - Image classification (where the image is treated as a single input)
  - Simple regression problems
  - Not commonly considered an RNN architecture as it lacks recurrent connections

2. One-to-Many

- **Input-Output Relationship**: Single input to sequence of outputs
- **Mathematical Formulation**:

$$\begin{aligned} h_0 &= \text{encoder}(x) \\ y_t &= \text{decoder}(h_{t-1}) \quad \text{for } t = 1, \ldots, T \\ h_t &= \text{RNN}(h_{t-1}, y_{t-1}) \end{aligned}$$

- **Architecture Details**:

```
# Pseudocode for one-to-many with teacher forcing
h = initial_state
outputs = []
first_input = initial_input
for t in range(output_length):
    # Combine input and previous hidden state
    h = tanh(W_hh * h + W_xh * first_input + b_h)
    output = softmax(W_hy * h + b_y)
    outputs.append(output)

    # For next step, use previous output (or ground truth
        in teacher forcing)
    first_input = output
```

- **Training Challenges**:

  - Exposure bias problem (difference between training and inference)
  - Need for teacher forcing or scheduled sampling

- **Example Applications**:
  - **Music Generation**: A starting chord generates a musical sequence
  - **Image Captioning**: An image (single input) generates a sequence of words
  - **Sequence Generation**: Initial seed generates a story or poem

- **Variations**:
  - Can use LSTM/GRU cells instead of basic RNN
  - Often combined with attention mechanisms

3. Many-to-One

- **Input-Output Relationship**: Sequence of inputs to single output
- **Mathematical Formulation**:

$$h_t = \text{RNN}(x_t, h_{t-1}) \quad \text{for } t = 1, \ldots, T$$
$$y = f(h_T)$$

  where f is typically a dense layer

- **Architecture Details**:

```
# Pseudocode for many-to-one with bidirectional option
h_forward = initial_state
h_backward = initial_state  # if bidirectional
all_outputs = []

# Forward pass
for input in input_sequence:
    h_forward = RNN_cell(input, h_forward)
    all_outputs.append(h_forward)

# For bidirectional, add reverse pass
if bidirectional:
    for input in reversed(input_sequence):
        h_backward = RNN_cell(input, h_backward)
        all_outputs.append(h_backward)

# Final prediction combines all hidden states
final_output = dense_layer(concatenate(all_outputs))
```

- **Output Aggregation Strategies**:
  - Use only final hidden state (h_T)

  - Average all hidden states
  - Weighted combination (attention)
  - Max pooling across time steps

- **Example Applications**:
  - **Sentiment Analysis**: Sequence of words → positive/negative score
  - **Time Series Classification**: Sensor readings → equipment failure prediction
  - **Video Classification**: Sequence of frames → action recognition

- **Practical Considerations**:
  - Handling variable-length sequences through padding/masking
  - Choice of final representation affects performance
  - Bidirectional often helps for text applications

4. Many-to-Many (Equal Length)

- Input: Sequence of inputs
- Output: Sequence of outputs (same length)
- Architecture:

```
# Pseudocode for many-to-many (NER)
h = initial_state
outputs = []
for input in input_sequence:
    output, h = RNN_cell(input, h)
    outputs.append(output)
```

- Example: Named Entity Recognition (our classroom example)

5. Many-to-Many (Different Length)

- Input: Complete input sequence
- Output: Generated output sequence
- Architecture (Encoder-Decoder):

```
# Pseudocode for encoder-decoder
# Encoder
h = initial_state
for input in input_sequence:
    h = RNN_cell(input, h)

# Decoder
outputs = []
first_output = RNN_cell(start_token, h)
```

```
outputs.append(first_output)
for i in range(max_output_length):
    next_output = RNN_cell(outputs[-1], h)
    outputs.append(next_output)
    if next_output == stop_token:
        break
```

- Example: Machine translation (English to French)

Implementing RNNs for Named Entity Recognition

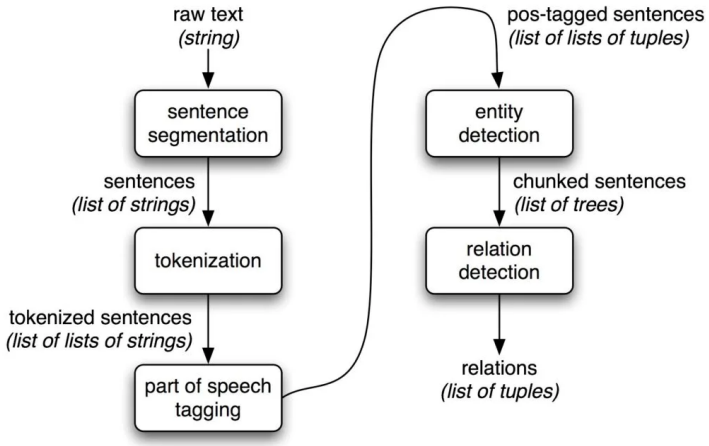


Figure 4: Named Entity Recognition example showing how RNNs can identify entities in text

Let's translate our classroom discussion into Python code using PyTorch to implement an RNN for named entity recognition:

```
import torch
import torch.nn as nn
import torch.optim as optim

class NER_RNN(nn.Module):
    def __init__(self, vocab_size, hidden_size, output_size):
        super(NER_RNN, self).__init__()
        self.hidden_size = hidden_size

        # Weight matrices
        self.W_xh = nn.Linear(vocab_size, hidden_size)  #
            W_xh
        self.W_hh = nn.Linear(hidden_size, hidden_size) #
            W_hh
        self.W_hy = nn.Linear(hidden_size, output_size) #
            W_hy
        self.tanh = nn.Tanh()
```

```
        self.sigmoid = nn.Sigmoid()

    def forward(self, inputs):
        # Initialize hidden state with zeros
        hidden = torch.zeros(1, self.hidden_size)

        outputs = []
        for i in range(inputs.size(0)):
            # Combine input and previous hidden state
            combined = self.W_xh(inputs[i]) + self.W_hh(
               hidden)
            hidden = self.tanh(combined)
            output = self.sigmoid(self.W_hy(hidden))
            outputs.append(output)

        return torch.stack(outputs)

# Example usage
vocab_size = 10000  # Size of our vocabulary
hidden_size = 100    # Number of hidden units
output_size = 1      # Binary classification (is named entity
    or not)

# Create model
model = NER_RNN(vocab_size, hidden_size, output_size)

# Example input (batch of 1 sequence with 3 time steps)
# "Virat Kohli plays" as one-hot encoded vectors
input_seq = torch.zeros(3, vocab_size)
input_seq[0][word_to_index["Virat"]] = 1  # "Virat"
input_seq[1][word_to_index["Kohli"]] = 1  # "Kohli"
input_seq[2][word_to_index["plays"]] = 1  # "plays"

# Forward pass
outputs = model(input_seq.unsqueeze(1))  # Add batch
   dimension
print(outputs)
```

Expected output might look like:

```
tensor([[[0.98]],  # "Virat" is a named entity
        [[0.97]],  # "Kohli" is a named entity
        [[0.02]]], # "plays" is not
       grad_fn=<StackBackward>)
```

This implements exactly the architecture we discussed in class:

1. At each time step, it combines:
    - Current input through W_xh

- Previous hidden state through W_hh

2. Applies tanh activation for the new hidden state
3. Computes output probability using sigmoid
4. Hidden state flows to next time step

Challenges and Limitations of Basic RNNs

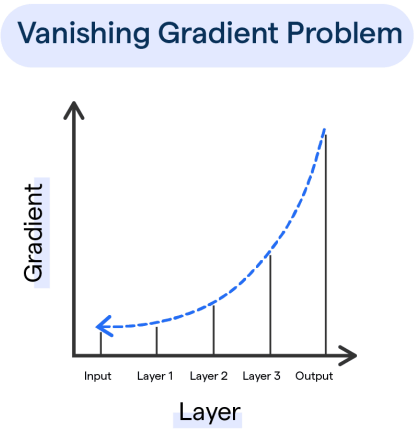


Figure 5: The vanishing gradients problem in RNNs - gradients become extremely small as they propagate back through time

While RNNs revolutionized sequence modeling, our classroom discussion hinted at several challenges:

1. **Vanishing Gradients**: During backpropagation through time (BPTT), gradients can become extremely small as they're multiplied through many time steps. This makes learning long-range dependencies nearly impossible.
2. **Exploding Gradients**: Conversely, gradients can grow exponentially, causing numerical instability.
3. **Computational Complexity**: Processing long sequences sequentially limits parallelization.
4. **Memory Limitations**: Basic RNNs struggle to maintain information over very long sequences.

These limitations are like trying to remember the first ball of a cricket match when analyzing the 300th ball - human memory fades over time, and so does RNN memory.

Advanced RNN Variants

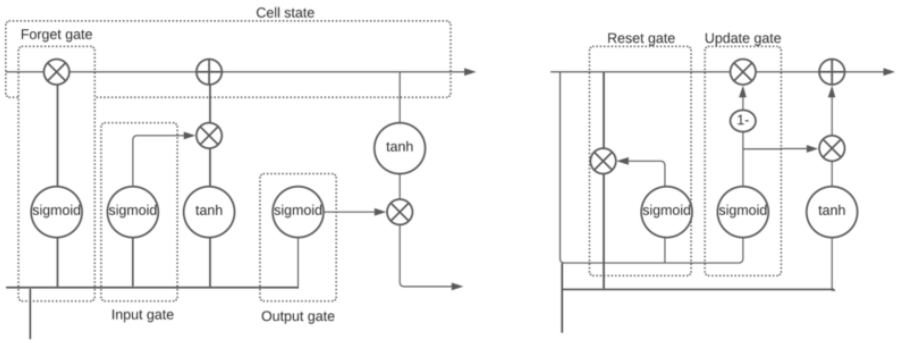


Figure 6: Comparison of LSTM (left) and GRU (right) architectures that solve the vanishing gradients problem

To address these limitations, more sophisticated architectures were developed:

Long Short-Term Memory (LSTM)

LSTMs introduce three gates to control information flow:

1. **Forget Gate**: Decides what to discard from cell state
2. **Input Gate**: Decides what new information to store
3. **Output Gate**: Decides what to output based on cell state

```
class LSTM_NER(nn.Module):
    def __init__(self, vocab_size, hidden_size, output_size):
        super().__init__()
        self.lstm = nn.LSTM(vocab_size, hidden_size,
           batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.lstm(x)  # out contains all hidden
           states
        return torch.sigmoid(self.fc(out))
```

Gated Recurrent Units (GRU)

GRUs simplify LSTMs with two gates:

1. **Reset Gate**: Determines how to combine new input with previous memory
2. **Update Gate**: Decides what proportion of previous memory to keep

```
class GRU_NER(nn.Module):
    def __init__(self, vocab_size, hidden_size, output_size):
        super().__init__()
        self.gru = nn.GRU(vocab_size, hidden_size,
           batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.gru(x)
        return torch.sigmoid(self.fc(out))
```

Practical Considerations in Training RNNs

From our classroom discussion, several practical aspects emerged:

1. **Initialization**: Initial hidden state (h_0) is typically initialized to zeros or small random values.
2. **Teacher Forcing**: When training sequence generation models, we sometimes feed the correct previous output rather than the model's prediction to stabilize training.
3. **Sequence Padding**: For batch processing, we often pad sequences to equal length and use masking to ignore padding.
4. **Bidirectional RNNs**: Process sequences both forward and backward to capture context from both directions.

```
# Bidirectional LSTM example
bidir_lstm = nn.LSTM(input_size=100, hidden_size=50,
                     num_layers=1, bidirectional=True)
```

Applications Beyond Named Entity Recognition

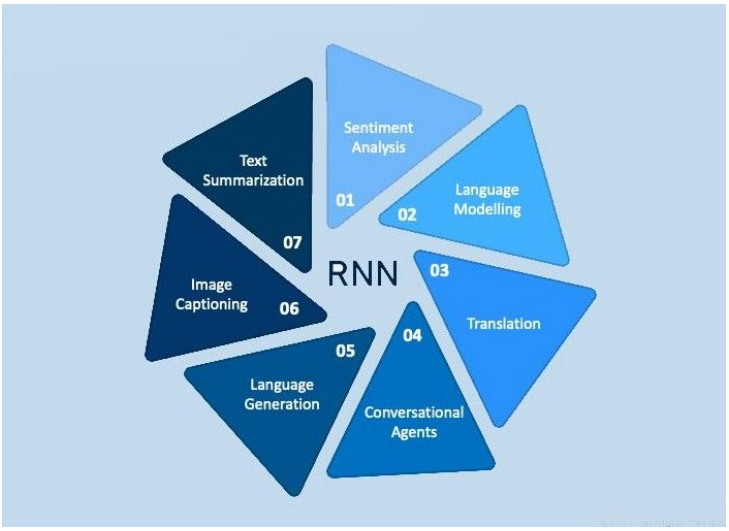


Figure 7: Various applications of RNNs in different domains

While we focused on NER in class, RNNs power numerous applications:

1. **Machine Translation**: Encoder-decoder architectures translate between languages
2. **Speech Recognition**: Converting audio waveforms to text
3. **Time Series Forecasting**: Predicting stock prices or weather patterns
4. **Video Analysis**: Understanding temporal sequences of frames
5. **Music Generation**: Creating sequential musical patterns

Current State and Evolution

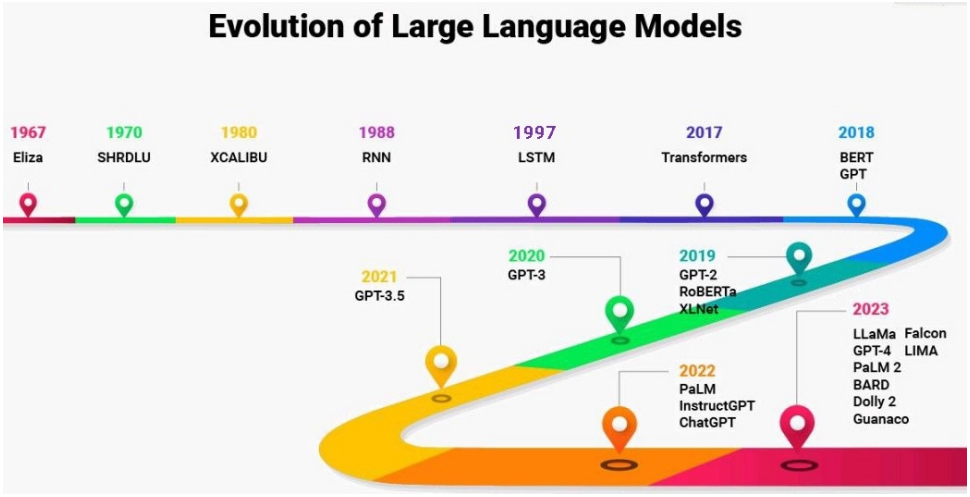


Figure 8: Evolution of sequence models from RNNs to Transformers

While RNNs were groundbreaking, our classroom discussion came at an interesting time in AI evolution. Today, Transformer architectures (like BERT, GPT) have largely surpassed RNNs for many tasks due to their parallel processing and superior handling of long-range dependencies. However, understanding RNNs remains crucial because:

1. They establish fundamental concepts of sequence modeling
2. Some applications still benefit from their sequential nature
3. They're computationally lighter than transformers for some tasks
4. Hybrid models often combine RNN and attention mechanisms

Summery

Our classroom journey through RNNs mirrored the historical development of sequence modeling in AI. We began with the limitations of traditional approaches, discovered how RNNs solve these problems through recurrent connections, implemented a basic RNN for named entity recognition, and discussed both the power and limitations of these architectures.

Just as cricket has evolved from test matches to T20s while maintaining its core principles, sequence modeling has evolved from RNNs to Transformers while preserving the fundamental need to handle sequential data effectively. The concepts learned in this session - recurrent connections, weight sharing across time steps, and sequential processing - remain relevant even as architectures become more sophisticated.

Key Takeaways

- **Sequential Processing**: RNNs are designed to handle sequential data by maintaining a hidden state that serves as memory of previous inputs, making them ideal for tasks like text processing, time series analysis, and speech recognition.
- **Architecture Variations**: Different RNN architectures exist for different tasks:
  - One-to-Many (e.g., image captioning)
  - Many-to-One (e.g., sentiment analysis)
  - Many-to-Many (e.g., machine translation, NER)
- **Mathematical Foundation**: The core RNN equations:

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

  enable information flow across time steps through weight matrices W_{hh}, W_{xh}, and W_{hy}.
- **Implementation Challenges**: Basic RNNs suffer from:
  - Vanishing/exploding gradients

  - Difficulty learning long-range dependencies
  - Sequential processing limitations

- **Advanced Variants**: LSTM and GRU architectures solve many RNN limitations through gating mechanisms that better control information flow.

- **Practical Considerations**: Effective RNN training requires attention to:
  - Proper initialization
  - Teacher forcing
  - Sequence padding and masking
  - Bidirectional processing when appropriate

- **Evolution**: While Transformers have surpassed RNNs for many tasks, understanding RNN fundamentals remains crucial for:
  - Grasping sequence modeling concepts
  - Working with legacy systems
  - Applications where RNNs remain competitive

- **Implementation**: The PyTorch example demonstrated how to build an RNN for Named Entity Recognition, highlighting the practical application of the theoretical concepts.