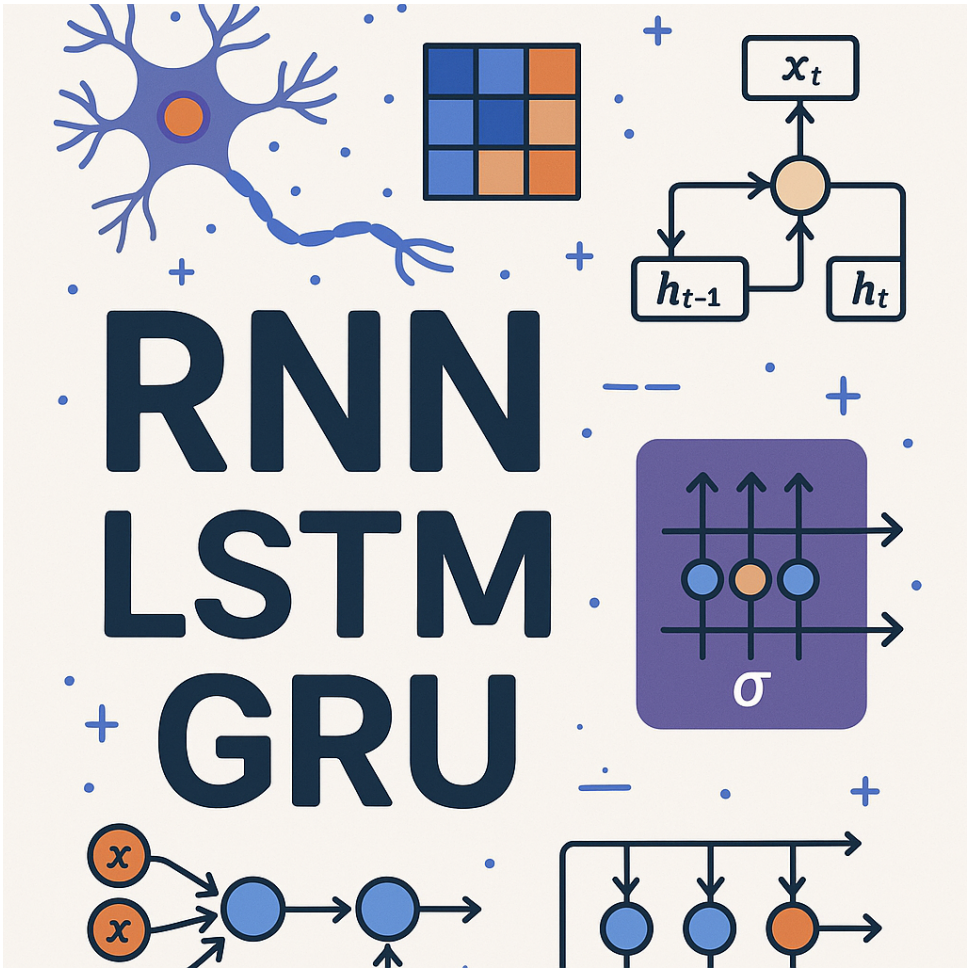
Comprehensive Machine Learning Notes

Minor in AI

May 5, 2025



The Need for Sequence Modeling - A Hospital Story

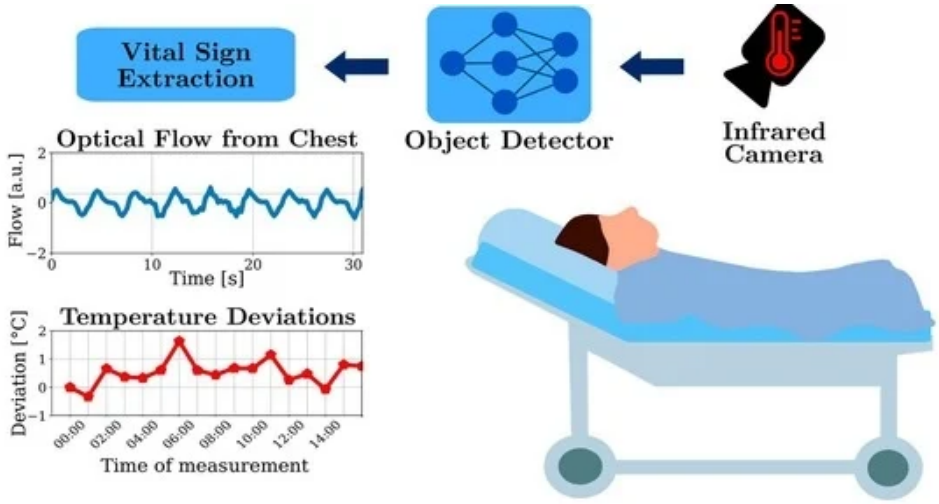


Figure 1: ICU patient monitoring demonstrates the importance of sequential data analysis

Imagine you're a doctor monitoring a patient in the ICU. The patient's vital signs - heart rate, blood pressure, oxygen levels - stream in every second. Yesterday's readings affect today's treatment plan. A single high blood pressure reading might be noise, but a rising trend over hours signals danger. This is sequence data where order matters and history is crucial.

Understanding Sequential Data Challenges

Traditional neural networks would treat each second's readings as independent, losing all temporal relationships. This is like a doctor who only looks at the current blood pressure without considering whether it's been rising or falling. Two key problems emerge:

- **Variable-Length Sequences**: Patient monitoring might last 8 hours or 8 days. Fixed-input networks can't handle this variability.
- **Temporal Dependencies**: A dropping oxygen level becomes meaningful when we know it followed a respiratory event 30 minutes prior.

The professor illustrated this with named entity recognition (NER) - identifying proper nouns in text. Consider:

- "Apple released a new phone" vs "She ate an apple"
- The word "Apple" means different things based on its sequence context

Recurrent Neural Networks - Biological Memory in Artificial Systems

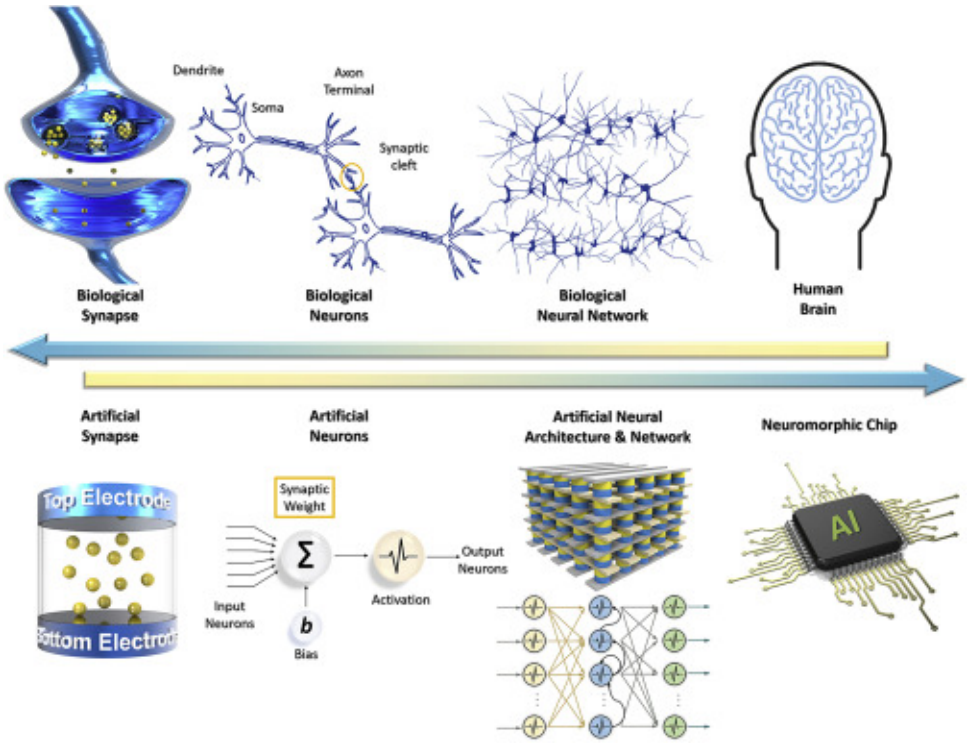


Figure 2: The human brain's memory system inspired RNN architecture

The Cognitive Science Behind RNNs: Recurrent Neural Networks represent one of the most biologically plausible architectures in deep learning, directly inspired by how human memory functions. When neuroscientists study working memory, they observe that our brains maintain contextual information through recurrent connections between neurons - precisely the mechanism RNNs emulate. Consider language comprehension: As you read this sentence, your brain automatically maintains activation from previous words to understand the current word's meaning. This persistent context is what allows you to resolve ambiguous references like pronouns ("it", "they") whose meaning depends entirely on prior context. Traditional feedforward networks fail catastrophically at such tasks because they process each input in isolation, while RNNs maintain this crucial temporal context through their hidden states.

Anatomy of an RNN Cell

The fundamental building block of all RNN architectures is the recurrent cell, which contains three core components working in concert:

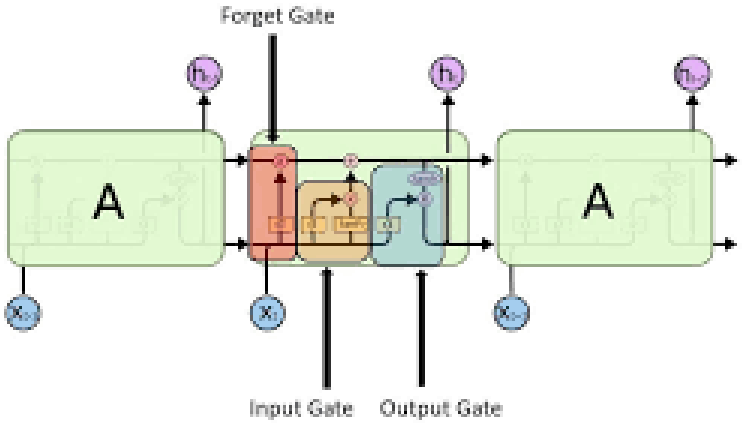


Figure 3: Detailed breakdown of an RNN cell's internal components

1. **Input Processing System**: The current input x_t enters through a dense neural layer that transforms it into a intermediate representation. This transformation uses the weight matrix W_{xh} which determines how input features influence the hidden state. In natural language processing, where inputs are often one-hot encoded words, this layer effectively learns word embeddings on the fly.

2. **Memory Maintenance System**: The hidden state h_{t-1} from the previous time step passes through another dense layer parameterized by W_{hh}. This weight matrix controls how much historical information carries forward versus how much gets discarded. The values in W_{hh} essentially determine the "memory decay rate" of the network.

3. **State Update Mechanism**: The transformed input and previous hidden state combine through addition before passing through a nonlinear activation (typically tanh). This operation can be understood as:

$$\text{New Memory} = \tanh(\text{Input Contribution} + \text{Memory Retention})$$

The tanh function serves two critical purposes: it squashes values to the [-1,1] range preventing explosive growth, and its nonlinearity allows the network to learn complex temporal dynamics.

The Mathematics of Memory Retention

The hidden state update equation $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$ deserves deeper examination. Let's analyze its behavior under different conditions:

1. **Input-Dominated Regime**: When $\|W_{xh}x_t\| \gg \|W_{hh}h_{t-1}\|$, the current input overwhelms the memory, making the network behavior similar to a feedforward network. This occurs when processing highly informative tokens that reset the context.

2. **Memory-Dominated Regime**: When $\|W_{hh}h_{t-1}\| \gg \|W_{xh}x_t\|$, the network ignores current inputs in favor of maintained context. This manifests when processing function words like "the" that don't change meaning.

3. **Balanced Regime**: In the optimal operating zone, the network performs delicate interpolation between new inputs and existing context. The tanh nonlinearity ensures this balance remains stable over many time steps.

The output computation $y_t = \sigma(W_{hy}h_t + b_y)$ typically uses sigmoid (for binary classification) or softmax (for multi-class) to produce predictions. Crucially, the output at time t depends only on the current hidden state h_t, which itself contains all relevant history.

Unfolding Through Time: Computational Perspective

When implementing RNNs, we conceptually "unroll" the network through time during both forward and backward passes:

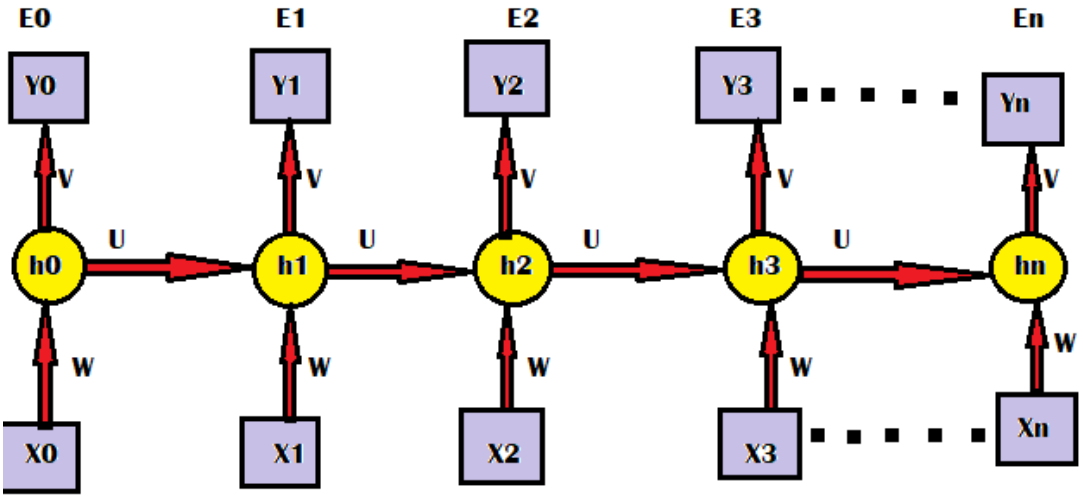


Figure 4: Complete unrolling of an RNN processing a 5-word sentence

Consider processing the sentence "The cat sat on the mat" with word-level RNN:

1. **Initialization**: We begin with h_0 typically initialized to zeros, representing no prior context. Some implementations use learned initial states.

2. **First Word ("The")**: - Input x_1 = one-hot vector for "the" - $h_1 = \tanh(W_{xh}x_1 + W_{hh}h_0 + b_h)$ - y_1 predicts whether "the" is part of a named entity

3. **Second Word ("cat")**: - Now h_1 carries information about "the" - The computation $h_2 = \tanh(W_{xh}x_2 + W_{hh}h_1 + b_h)$ combines "cat" with context "the" - This allows the network to learn phrases like "the cat" as units

4. **Subsequent Words**: The pattern continues, with each hidden state accumulating relevant context. By "sat", the network may recognize this as a verb following a noun phrase.

The key advantage emerges in variable-length sequences - whether the input is 5 words or 50, the same weights W_{xh}, W_{hh}, W_{hy} are reused at each step. This parameter sharing makes RNNs fundamentally different from simply stacking feedforward networks.

Practical Implementation Considerations

Implementing RNNs efficiently requires addressing several technical challenges:

1. **Batch Processing**: Modern implementations process multiple sequences simultaneously in batches. This requires handling sequences of different lengths through padding and masking.

2. **Teacher Forcing**: During training, we often feed the ground truth previous output rather than the network's own prediction to stabilize training.

3. **Sequence Padding**: Shorter sequences in a batch are padded with zeros, requiring careful masking during loss computation to ignore padding.

Listing 1: Complete RNN Implementation with Batching

```
import torch
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size,
           hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size,
           output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = torch.tanh(self.i2h(combined))
        output = self.softmax(self.i2o(combined))
        return output, hidden

    def initHidden(self, batch_size):
        return torch.zeros(batch_size, self.hidden_size)

# Example usage
rnn = RNN(input_size=100, hidden_size=128, output_size=10)
hidden = rnn.initHidden(batch_size=32)
inputs = torch.randn(32, 10, 100)  # (batch, seq_len,
   input_size)

# Process sequence step-by-step
outputs = []
for i in range(inputs.size(1)):
    output, hidden = rnn(inputs[:, i, :], hidden)
    outputs.append(output)
```

The Challenge of Long-Term Dependencies

While RNNs theoretically can maintain information indefinitely, practical training runs into the famous vanishing/exploding gradient problem. Consider what happens when we try to learn a simple sequence copying task:

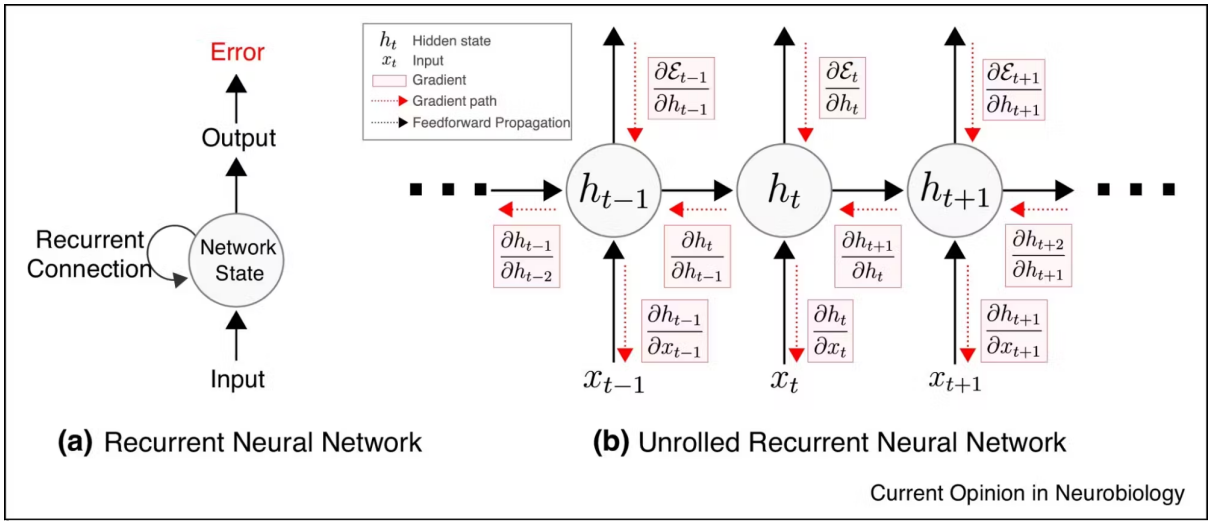


Figure 5: Gradient flow analysis in a 10-step unrolled RNN

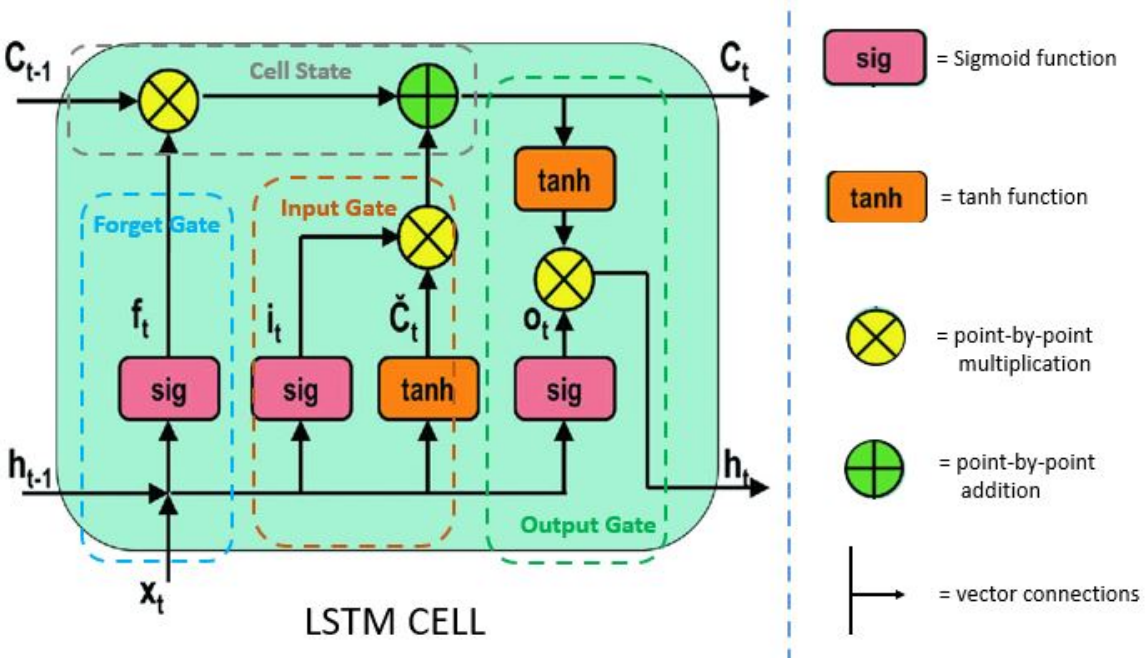
The gradient of the loss with respect to early hidden states involves repeated multiplication of W_{hh}:

$$\frac{\partial h_{10}}{\partial h_1} = \prod_{k=1}^{9} \frac{\partial h_{k+1}}{\partial h_k} = \prod_{k=1}^{9} \text{diag}(\tanh'(W_{hh}h_k + W_{xh}x_{k+1} + b_h))W_{hh}$$

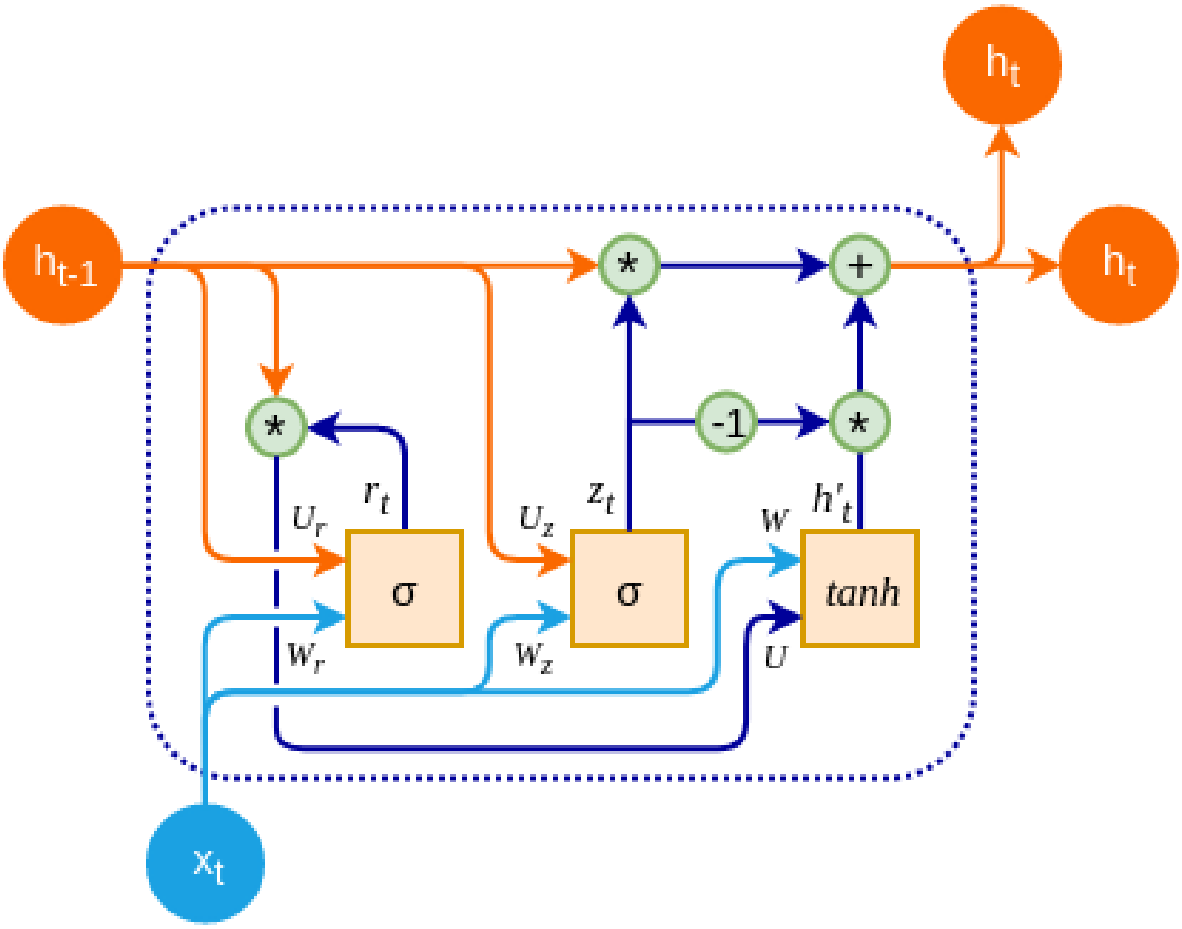
If the largest eigenvalue of W_{hh} is less than 1, this product shrinks exponentially (vanishing gradients). If greater than 1, it grows exponentially (exploding gradients). In practice, this makes basic RNNs unable to learn dependencies spanning more than 10-20 time steps.

Modern Solutions: LSTM and GRU Architectures

The limitations of basic RNNs led to two improved architectures that dominate practical applications:



(a) LSTM cell with three gating mechanisms



(b) GRU cell with two gating mechanisms

Figure 6: Detailed views of gated RNN architectures

Long Short-Term Memory (LSTM) Networks

The Long Short-Term Memory (LSTM) network was specifically designed to address the vanishing gradient problem in traditional RNNs. Its architecture introduces several sophisticated gating mechanisms that carefully regulate information flow:

Core Components of LSTM

1. **Cell State** (C_t): The central innovation of LSTMs is the cell state - a horizontal conveyor belt running through the entire sequence. Mathematically, it evolves as:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

where $\odot$ denotes element-wise multiplication. This linear operation allows gradients to flow unchanged when the forget gate $f_t \approx 1$, enabling long-term memory.

2. **Forget Gate** (f_t): Determines what information to discard from the cell state. Computed as:

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

where σ is the sigmoid function. Values near 1 indicate "keep this information" while values near 0 mean "discard this". The forget gate learns to reset memory when the context changes.

3. **Input Gate** (i_t): Controls how much new information gets stored in the cell state:

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

It works in tandem with the candidate cell state $\tilde{C}_t$:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

4. **Output Gate** (o_t): Regulates what parts of the cell state are exposed as the hidden state:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \odot \tanh(C_t)$$

Listing 2: LSTM Implementation in PyTorch

```
import torch
import torch.nn as nn

# Single LSTM cell implementation
lstm_cell = nn.LSTMCell(input_size=100, hidden_size=64)

# Initialize states
hx = torch.zeros(32, 64)  # Hidden state
cx = torch.zeros(32, 64)  # Cell state

# Process sequence
for i in range(seq_len):
    hx, cx = lstm_cell(inputs[:,i,:], (hx, cx))
```

Why LSTMs Work

The LSTM's effectiveness comes from several architectural features:

- **Additive State Updates**: The cell state update $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$ is additive rather than multiplicative, preventing gradient vanishing
- **Gated Protection**: The forget gate protects the cell state from noisy updates, allowing selective memory retention
- **Nonlinear Transformations**: Multiple nonlinearities (sigmoid and tanh) allow complex learning of what to remember/forget
- **Decoupled Memory**: The cell state C_t and hidden state h_t serve different purposes - long-term storage vs working memory

Gated Recurrent Units (GRUs)

The Gated Recurrent Unit (GRU) is a streamlined variant of the LSTM that maintains similar performance with fewer parameters:

Core Components of GRU

1. **Update Gate (z_t)**: Decides how much of the previous state to keep:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

This combines the LSTM's input and forget gates into a single mechanism.

2. **Reset Gate (r_t)**: Controls how much past information affects the candidate state:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

3. **Candidate Activation ($\tilde{h}_t$)**: Proposes a new state considering the reset gate:

$$\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t] + b)$$

4. **Final State Update**: Interpolates between old and new states:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Listing 3: GRU Implementation in PyTorch

```
import torch
import torch.nn as nn

# Single GRU cell implementation
gru_cell = nn.GRUCell(input_size=100, hidden_size=64)

# Initialize state
hx = torch.zeros(32, 64)  # Hidden state

# Process sequence
for i in range(seq_len):
    hx = gru_cell(inputs[:,i,:], hx)
```

GRU vs LSTM: Key Differences

Feature	LSTM	GRU
Gates	3 (input, forget, output)	2 (update, reset)
Memory State	Separate cell state (C_t)	Unified hidden state (h_t)
Parameters	More (4 sets of weights)	Fewer (3 sets of weights)
Performance	Better for long sequences	Comparable for most tasks
Training Speed	Slower	Faster

Table 1: Comparison between LSTM and GRU architectures

Choosing Between LSTM and GRU

The choice between LSTM and GRU depends on several factors:

- **Sequence Length**: LSTMs generally perform better for very long sequences (100+ steps)
- **Dataset Size**: GRUs train faster and perform well with smaller datasets
- **Computational Resources**: GRUs require about 25% fewer parameters
- **Task Requirements**: LSTMs may be better for precise timing/position tasks
- **Regularization Needs**: GRUs often require less dropout regularization

Recent research suggests that proper hyperparameter tuning often matters more than the choice between LSTM and GRU. The reduced computational cost of GRUs makes them particularly attractive for:

- Real-time applications
- Mobile and embedded systems
- Scenarios requiring frequent retraining

Both architectures remain superior to vanilla RNNs for most sequence modeling tasks, with the relative performance depending heavily on the specific application domain and implementation details.

Bidirectional RNNs for Contextual Understanding

Standard RNNs process sequences strictly left-to-right, limiting their ability to incorporate future context. Bidirectional RNNs solve this by running two separate RNNs:

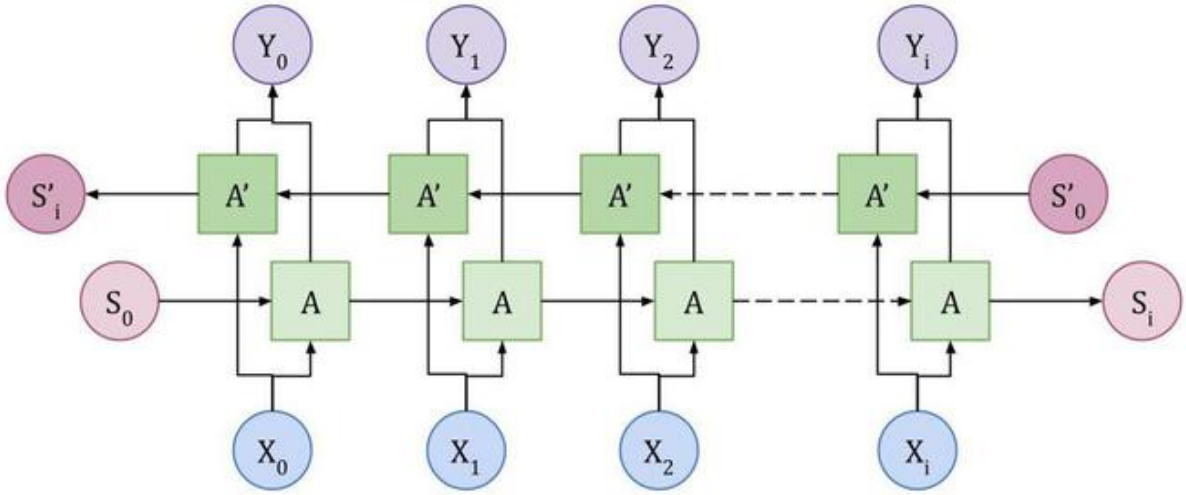


Figure 7: Bidirectional LSTM processing sequence in both directions

1. Forward RNN processes the sequence from x_1 to x_T 2. Backward RNN processes from x_T to x_1 3. Hidden states are concatenated at each time step

This allows predictions at time t to leverage both past and future context, crucial for tasks like:

- Named Entity Recognition ("Apple" in tech vs fruit contexts)
- Speech recognition (phoneme classification)
- Protein structure prediction

The PyTorch implementation is straightforward:

Listing 4: Bidirectional RNN in PyTorch

```
rnn = nn.RNN(input_size=100, hidden_size=64, num_layers=2,
   bidirectional=True)
output, hidden = rnn(input_sequence)
# output shape: (seq_len, batch, 2*hidden_size)
```

Practical Applications and Success Stories

RNNs and their variants have powered numerous real-world applications:

1. **Machine Translation**: Google's first neural translation systems used stacked LSTMs to achieve human-level quality on many language pairs.

2. **Speech Recognition**: Apple's Siri and Amazon's Alexa employed bidirectional LSTMs for acoustic modeling.

3. **Medical Time Series**: Predicting patient outcomes from ICU sensor data using RNNs has demonstrated superior performance to traditional methods.

4. **Financial Forecasting**: Modeling high-frequency trading data and detecting anomalous patterns.

TF-IDF Enhancements

1. **Sublinear TF Scaling**: Use $\log(1 + TF)$ to prevent bias toward long documents
2. **Smoothing**: Add 1 to numerator and denominator in IDF to handle unseen terms
3. **Normalization**: Scale vectors to unit length for cosine similarity

Listing 5: Enhanced TF-IDF with Scikit-Learn

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

docs = [
    "The quick brown fox jumps over the lazy dog",
    "Never jump over lazy dogs quickly",
    "Quick foxes are better than lazy dogs"
]

vectorizer = TfidfVectorizer(
    norm='l2',           # Euclidean normalization
    sublinear_tf=True,   # Use 1 + log(tf)
    smooth_idf=True      # Add 1 to all counts
)

tfidf = vectorizer.fit_transform(docs)

pd.DataFrame(
    tfidf.toarray(),
    columns=vectorizer.get_feature_names_out()
)
```

This shows how:

- "quick" gets higher weight than "the" (stop words)
- Each document vector has unit length (norm='l2')
- Values are smoothed for better generalization

These techniques form the foundation for modern NLP systems while remaining interpretable - a crucial requirement in many real-world applications.