Understanding RNNs and Vanishing Gradient

Minor in AI - IIT ROPAR

12th March, 2025

The Story of Rahul – The News Analyst

Rahul is a **news analyst** working for a media company. Every day, he reads **hundreds of news articles** and needs to **summarize them for the company's daily bulletin**.

One day, his boss gives him a **tough assignment**: *"Rahul, I want you to prepare a summary of the entire week's news. The summary should highlight the key events and trends we have observed."*

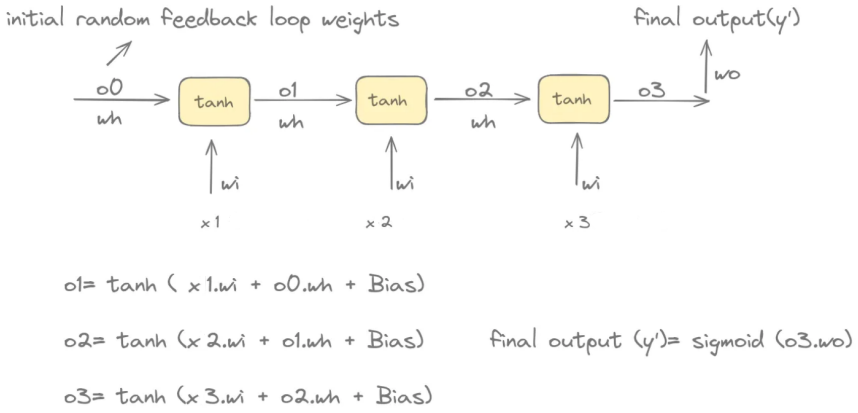
Rahul starts working but faces a major problem:

- He remembers **yesterday's news very clearly**.
- He can recall **the news from two days ago** with some effort.
- But **the news from five days ago?** The details are **hazy**.
- And **the news from the start of the week?** Almost **completely forgotten!**

His brain is **prioritizing recent news**, but **forgetting past events**, even though they are **crucial for understanding long-term trends**. This is analogous to what happens in **Recurrent Neural Networks (RNNs)**.

How RNNs Process Sequences

A Recurrent Neural Network (RNN) is a type of neural network designed to handle **sequential data**. Unlike traditional feedforward networks, RNNs maintain a **hidden state** that helps retain information from previous time steps.



Mathematical Formulation

At each time step t, the RNN computes the hidden state using:

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

where:

- x_t = input at time step t
- h_{t-1} = hidden state from the previous step
- W_h, W_x = weight matrices
- b_h = bias term
- tanh = activation function

The final output is computed as:

$$y_t = W_y h_t + b_y \tag{2}$$

where y_t is the predicted output at time step t.

The Vanishing Gradient Problem

Just like Rahul struggles to remember older news, RNNs face a **memory problem**. When training an RNN using **Backpropagation Through Time (BPTT)**, we propagate errors backward through time. However:

- The model updates earlier layers using **gradients**.
- If the sequence is **too long**, these gradients become **extremely small**.
- Small gradients mean the network **stops learning from earlier words**.
- The **oldest information is effectively lost**.

Mathematically, this happens because:

- At each step, the gradient is **multiplied by the derivative of the activation function** (usually a value between 0 and 1).
- Multiplying small numbers repeatedly leads to an **exponential decrease in gradient values**.
- This means that after many steps, the gradients **become so small** that the model **cannot learn anything from earlier inputs**.

Solutions to the Vanishing Gradient Problem

To address this problem, advanced RNN architectures introduce mechanisms to retain information over long sequences.

Gated Recurrent Units (GRU)

GRUs introduce **gates** that **control what information is remembered or forgotten**. This helps in **preserving long-term dependencies** without vanishing gradients.

Long Short-Term Memory (LSTM)

LSTMs take this further by **storing information in a memory cell** and using:

- **Forget Gate** – Decides what to erase.
- **Input Gate** – Decides what new information to store.
- **Output Gate** – Decides what to send as output.

Attention Mechanisms

Attention mechanisms dynamically focus on relevant words in a sentence. Instead of relying on **fixed-length memory**, they **scan all words** and pick the **most important ones**.

Python Implementation of RNN Forward Propagation

We now implement the forward propagation of an RNN using Python. The goal is to process sequential input data and predict outputs over multiple time steps.

Code Implementation

```
import numpy as np

np.random.seed(182)

# Initializing input data
x_t = np.random.randn(5000,5,10) # Shape: (vocab size, batch size, time steps)

a_prev = np.random.randn(231,5) # Initial hidden state (hidden state size, batch size)

# Defining parameters
parameters = {}
parameters["w1"] = np.random.randn(231,5000) # Weight matrix for input
parameters["w2"] = np.random.randn(231,231) # Weight matrix for hidden state
parameters["b1"] = np.random.randn(231,1)   # Bias for hidden state
parameters["b2"] = np.random.randn(5000,1)  # Bias for output
parameters["w3"] = np.random.randn(5000,231) # Weight matrix for output

def softmax(x):
    return np.exp(x)/np.sum(np.exp(x), axis=0)

# Forward propagation for a single time step
def rnn_cell_forward(xt, a_prev, parameters):
    w1 = parameters["w1"]
```

```
    w2 = parameters["w2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]
    w3 = parameters["w3"]

    # Compute the next hidden state
    a_next = np.tanh(np.dot(w1, xt) + np.dot(w2, a_prev) + b1)

    # Compute output prediction
    yt_pred = softmax(np.dot(w3, a_next) + b2)

    return a_next, yt_pred

# Forward propagation through all time steps
def run_forward(x, a0, parameters):
    n_x, m, t_x = x.shape  # Extract input dimensions
    n_y, n_a = parameters["w3"].shape  # Extract output and hidden state dimensions

    a_next = a0

    for i in range(t_x):
        xt = x[:,:,i]
        a_next, yt_pred = rnn_cell_forward(xt, a_next, parameters)

    return a_next, yt_pred

# Running the RNN
a_tmp, y_pred = run_forward(x_t, a_prev, parameters)
```

Step-by-Step Explanation

- Initialization:
  - x_t represents the input data with dimensions (vocab size, batch size, time steps).
  - a_{prev} is the initial hidden state.
  - The parameters dictionary stores weights and biases used in forward propagation.
- Softmax Function
  - Converts raw output scores into probabilities to ensure numerical stability and interpretability.
- rnn_cell_forward Function
  - Computes the next hidden state a_t using weight matrices and biases.
  - Applies the 'tanh' activation function to regulate values within a range.
  - Predicts the output probability using softmax.
- run_forward Function
  - Iterates over all time steps to process the entire input sequence.
  - Calls 'rnn_cell_forward' at each time step to update the hidden state.

Key Takeaways

- **RNNs are designed for sequential data** such as time-series forecasting, speech recognition, and natural language processing. Unlike feedforward networks, they maintain a **hidden state** to capture temporal dependencies.
- **Mathematically, RNNs update their hidden state** using the equation:

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

and compute output as:

$$y_t = W_y h_t + b_y$$

where W_h, W_x, W_y are weight matrices, and b_h, b_y are biases.

- **The Vanishing Gradient Problem hinders learning from long sequences** because gradients shrink exponentially during backpropagation. This prevents the model from retaining long-term dependencies, making it ineffective for complex tasks.
- **Solutions such as GRU, LSTM, and Attention Mechanisms** address vanishing gradients. GRUs introduce gates to regulate memory, LSTMs add a dedicated memory cell, and Attention allows dynamic focus on relevant words.
- **The Python implementation of RNN forward propagation demonstrates how sequences are processed** through matrix operations. The core steps involve computing hidden states, applying activation functions, and generating predictions using softmax.
- **RNNs, despite their limitations, form the foundation for advanced models like Transformers** that leverage self-attention for better long-range dependency handling.