Backpropagation in RNN

🔷 Summary (For quick orientation)

- **Backpropagation** is how the RNN learns by adjusting weights.
- In RNNs, we use a special version called **Backpropagation Through Time (BPTT)**.
- This is because RNNs process sequences **step-by-step**, and errors must flow **back through each time step**.
- It helps update weights so the network performs better in the next round.

🔷 What is Normal Backpropagation?

Let's understand what backpropagation is in regular neural networks first:

- You give input → it flows forward through the layers → you get output.
- You calculate how wrong the output is (loss).
- Then, using **gradient descent**, you adjust the weights **backwards** to reduce the loss.
- This is done using **chain rule** from calculus.

🔷 The Challenge with RNNs

In an RNN:

- The same **weights are reused** at every time step.
- It maintains a **hidden state**, which gets passed from one time step to the next.
- So errors must be backpropagated not only **across layers** but also **across time**.

This is why we need a special version of backpropagation:

👉 **Backpropagation Through Time (BPTT)**

🔷 BPTT: Backpropagation Through Time

Imagine we have an RNN processing this sequence:

Input sequence: x_0, x_1, x_2

Time steps: 0, 1, 2

🧠 Step 1: Forward Pass

At each time step:

- Take the current input x_t
- Combine it with the previous hidden state h_{t-1}
- Produce a new hidden state h_t
- Compute output $\hat{y}_t$ from h_t

Repeat this for all time steps. Save each hidden state and output.

$x_0 \rightarrow h_0 \rightarrow \hat{y}_0$

$x_1 \rightarrow h_1 \rightarrow \hat{y}_1$

$x_2 \rightarrow h_2 \rightarrow \hat{y}_2$

🧠 Step 2: Compute Loss

At each step, compare the predicted output $\hat{y}_t$ with the actual output y_t.

Calculate a **loss** (e.g., cross-entropy or MSE).

Then compute total loss:

$\text{Total Loss} = L_0 + L_1 + L_2$

🧠 Step 3: Backward Pass — Through Time!

Now the key idea:

To update the RNN, we need to:

1. **Go backward in time**: start from the last time step and go back to the beginning.
2. At each step, compute **gradients of the loss** with respect to:
   - Output weights (e.g., from h_t to $\hat{y}_t$)
   - Hidden state weights (e.g., from h_{t-1} to h_t)
   - Input weights (e.g., from x_t to h_t)

We apply the **chain rule**, like this:

$$
\begin{aligned}
\partial Loss/\partial W = \partial Loss/\partial \hat{y}_2 * \partial \hat{y}_2/\partial h_2 * \partial h_2/\partial W \\
+ \partial h_2/\partial h_1 * \partial h_1/\partial W \\
+ \partial h_1/\partial h_0 * \partial h_0/\partial W
\end{aligned}
$$

This shows:

- A single loss at time t affects **many previous time steps**.
- Gradients are computed by **unrolling the network** in time.

🔷 Why Is This Called "Unrolling"?

Think of the RNN loop like a **spring**.

To compute gradients, we **unroll** it into a straight line:

$x_0 \rightarrow h_0 \rightarrow \hat{y}_0$

$x_1 \rightarrow h_1 \rightarrow \hat{y}_1$

$x_2 \rightarrow h_2 \rightarrow \hat{y}_2$

- Now it looks like a **feedforward neural network**, but with **shared weights** and **connected hidden states**.
- Then we do **normal backpropagation** on this unrolled version.

🔷 Problems in BPTT

🔶 1. Vanishing Gradient

- Gradients become **very small** as they flow back through many time steps.
- Early time steps get **almost no learning** signal.
- This is why basic RNNs struggle with **long sequences**.

🔶 2. Exploding Gradient

- Sometimes gradients become **too large** and cause instability.

Solutions:

- Use **gradient clipping**: force gradients to stay in a safe range.
- Use **LSTM/GRU** cells which have gates to control the flow of gradients.

🔷 What Gets Updated in BPTT?

There are typically three weight matrices in a vanilla RNN:

- `W_xh`: input → hidden state
- `W_hh`: hidden state → next hidden state (recurrent weight)
- `W_hy`: hidden state → output

All of them are updated during BPTT, using the total accumulated gradients over time.

🔷 Truncated BPTT

When sequences are long, it's expensive to backprop through the **entire** sequence.

So we can **truncate** it:

- Process 100 steps forward
- Backpropagate through only last 10 steps

This is called **Truncated BPTT**, and it helps save memory and time.