CO4

DEEP LEARNING

23AD2205A

Topic:

INTRODUCTION TO RECURRENT NEURAL NETWORKS (RNNS) & BACKPROPAGATION THROUGH TIME (BPTT)

Session - 20

AIM OF THE SESSION

To familiarize students with the   sequence prediction problems

INSTRUCTIONAL OBJECTIVES

This Session is designed to:

1. Discuss the Contractive Autoencoders and Variational autoencoder
2. Demonstrate the concept of Contractive Autoencoders and Variational autoencoder
   Discussion on Contractive Autoencoders and Variational autoencoder

LEARNING OUTCOMES

At the end of this session, you should be able to: concepts for real time applications

1. To build Contractive Autoencoders and Variational autoencoder
2. To apply different types of Contractive Autoencoders and Variational autoencoder

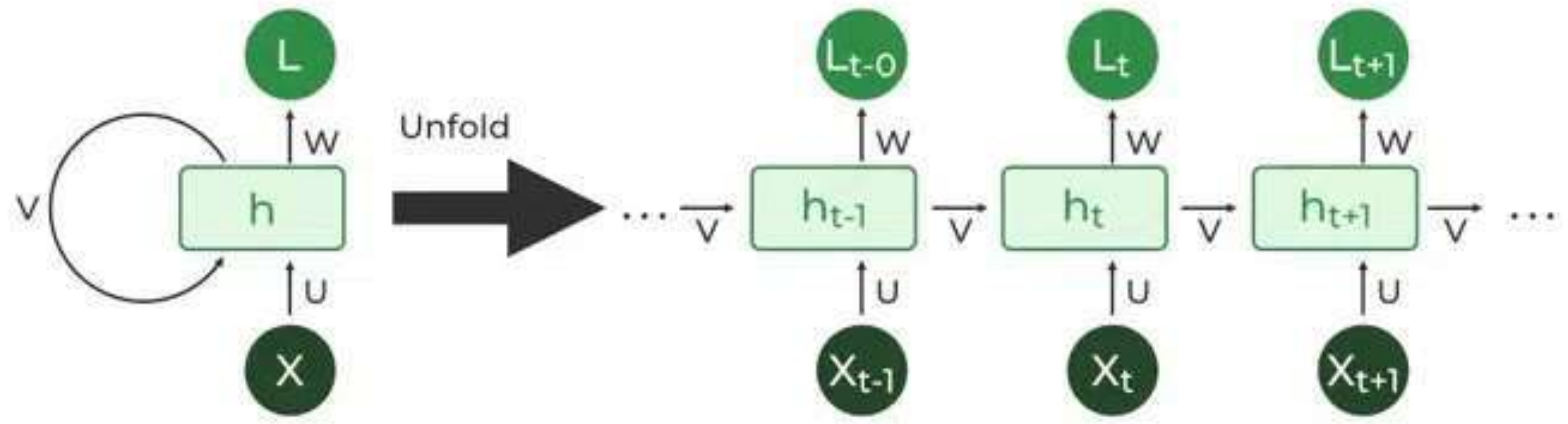
What are Recurrent Neural Networks (RNNs)?

- Traditional neural networks process inputs independently, limiting their ability to learn from sequential data.
- RNNs introduce loops, where the output from one step is fed back as input to the next, enabling memory.
- They are ideal for tasks where context from previous inputs is essential.

How RNNs Work

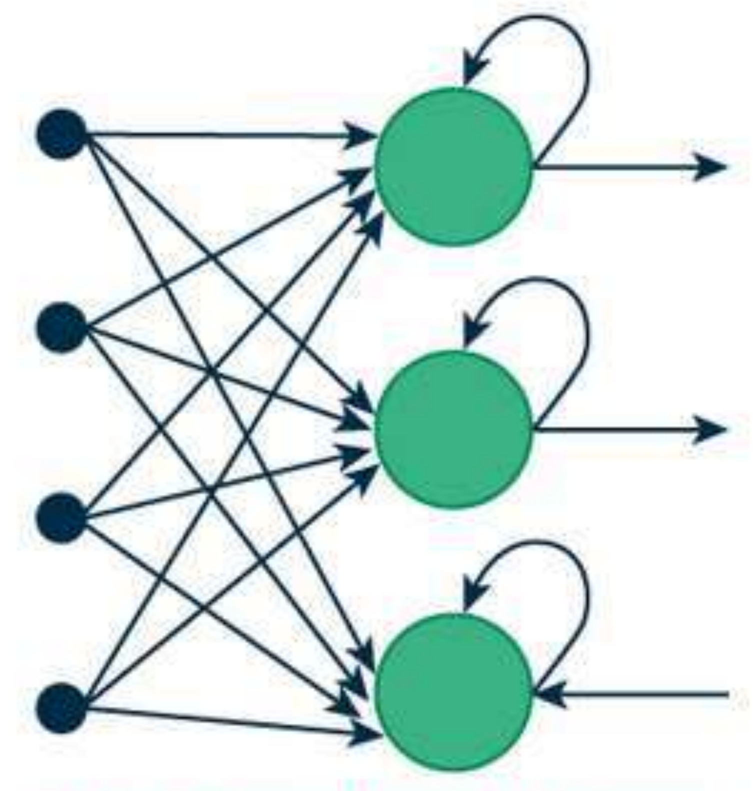
- RNNs apply the same network to each element in a sequence.
- They maintain a *hidden state* that retains information across time steps.
- Shared parameters across all steps reduce complexity and enable learning of temporal dependencies

Recurrent Neural Network

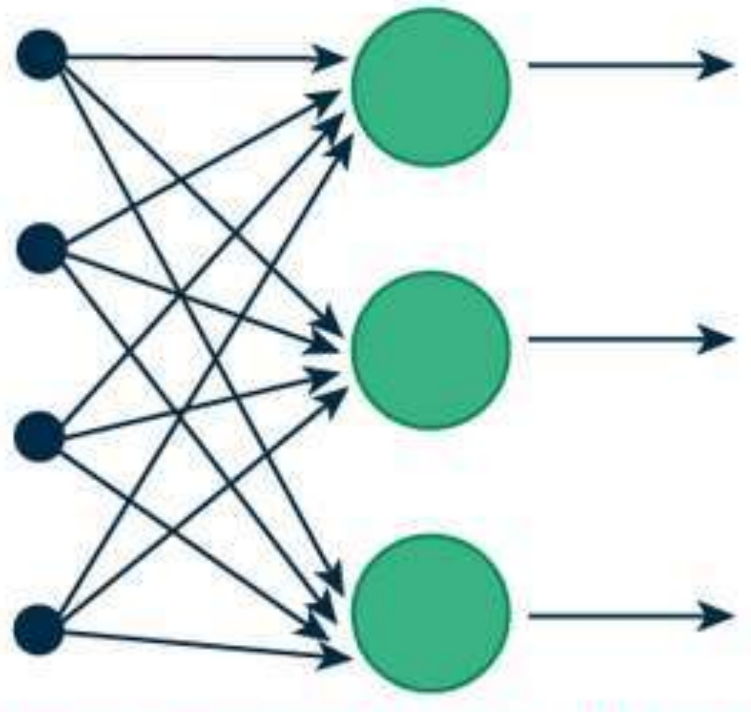


RNN vs. Feedforward Neural Networks

- **FNNs:** Process data in one direction, without memory of past inputs.
- **RNNs:** Utilize feedback loops to remember previous inputs.
- RNNs excel in sequential tasks where context matters.



(a) Recurrent Neural Network



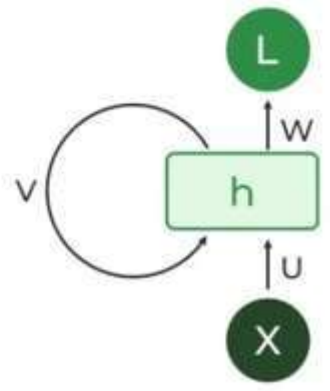
(b) Feed-Forward Neural Network

Key Components of RNNs

1. **Recurrent Neurons:** Maintain hidden state, allowing memory across time.
2. **RNN Unfolding:** Expands the recurrent structure over time steps for training.

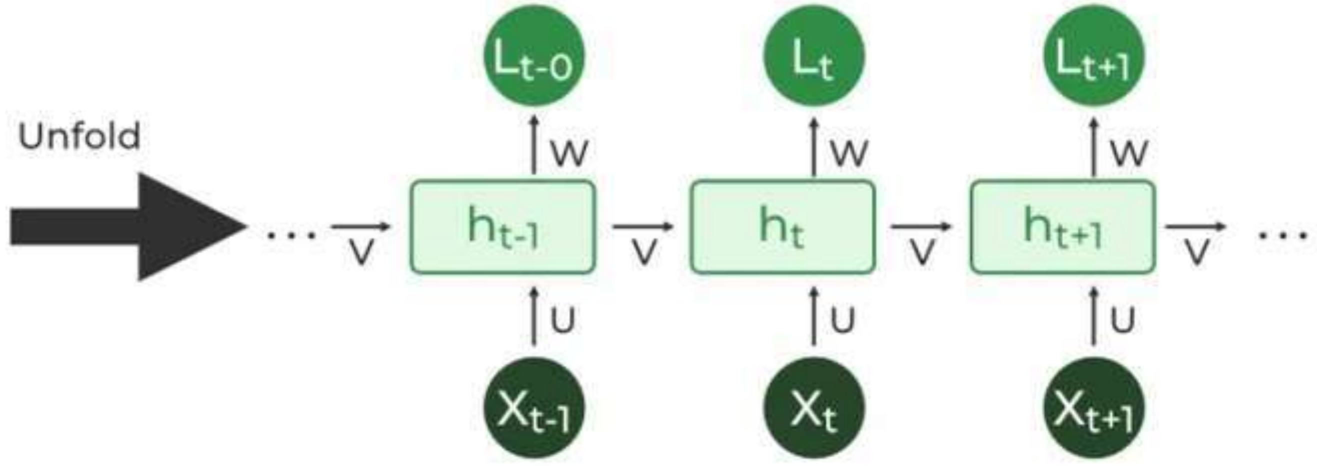
Recurrent Neurons

- Recurrent units hold a hidden state that maintains information about previous inputs in a sequence.
- Recurrent units can “remember” information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



RNN Unfolding:

- process of expanding the recurrent structure over time steps.
- During unfolding, each step of the sequence is represented as a separate layer in a series, illustrating how information flows across each time step.

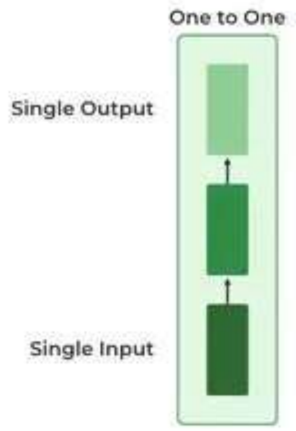


Types of RNNs

- **One-to-One RNN:** Single input to single output (e.g., image classification).
- **One-to-Many RNN:** Single input to multiple outputs (e.g., image captioning).
- **Many-to-One RNN:** Multiple inputs to a single output (e.g., sentiment analysis).
- **Many-to-Many RNN:** Sequence input to sequence output (e.g., language translation).

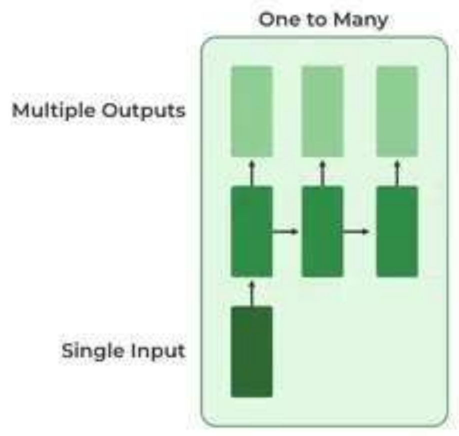
One-to-One RNN

- behaves as the **Vanilla Neural Network**, is the simplest type of neural network architecture.
- In this setup, there is a single input and a single output.
- Commonly used for straightforward classification tasks where input data points do not depend on previous elements.



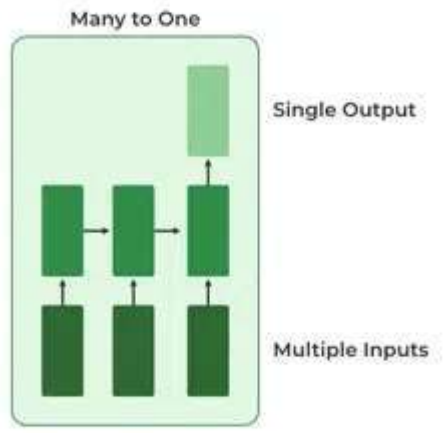
One-to-Many RNN:

- In a **One-to-Many RNN**, the network processes a single input to produce multiple outputs over time.
- This setup is beneficial when a single input element should generate a sequence of predictions.



Many-to-One RNN:

- receives a sequence of inputs and generates a single output.
- This type is useful when the overall context of the input sequence is needed to make one prediction.
- In sentiment analysis, the model receives a sequence of words (like a sentence) and produces a single output, which is the sentiment of the sentence (positive, negative, or neutral).



Variants of Recurrent Neural Networks (RNNs)

1. Vanilla RNN

- This simplest form of RNN consists of a single hidden layer, where weights are shared across time steps.
- Vanilla RNNs are suitable for learning short-term dependencies but are limited by the vanishing gradient problem, which hampers long-sequence learning.

2. Bidirectional RNNs

- Bidirectional RNNs process inputs in both forward and backward directions, capturing both past and future context for each time step.
- This architecture is ideal for tasks where the entire sequence is available, such as named entity recognition and question answering.

3. Long Short-Term Memory Networks (LSTMs)

introduce a memory mechanism to overcome the vanishing gradient problem. Each LSTM cell has three gates:

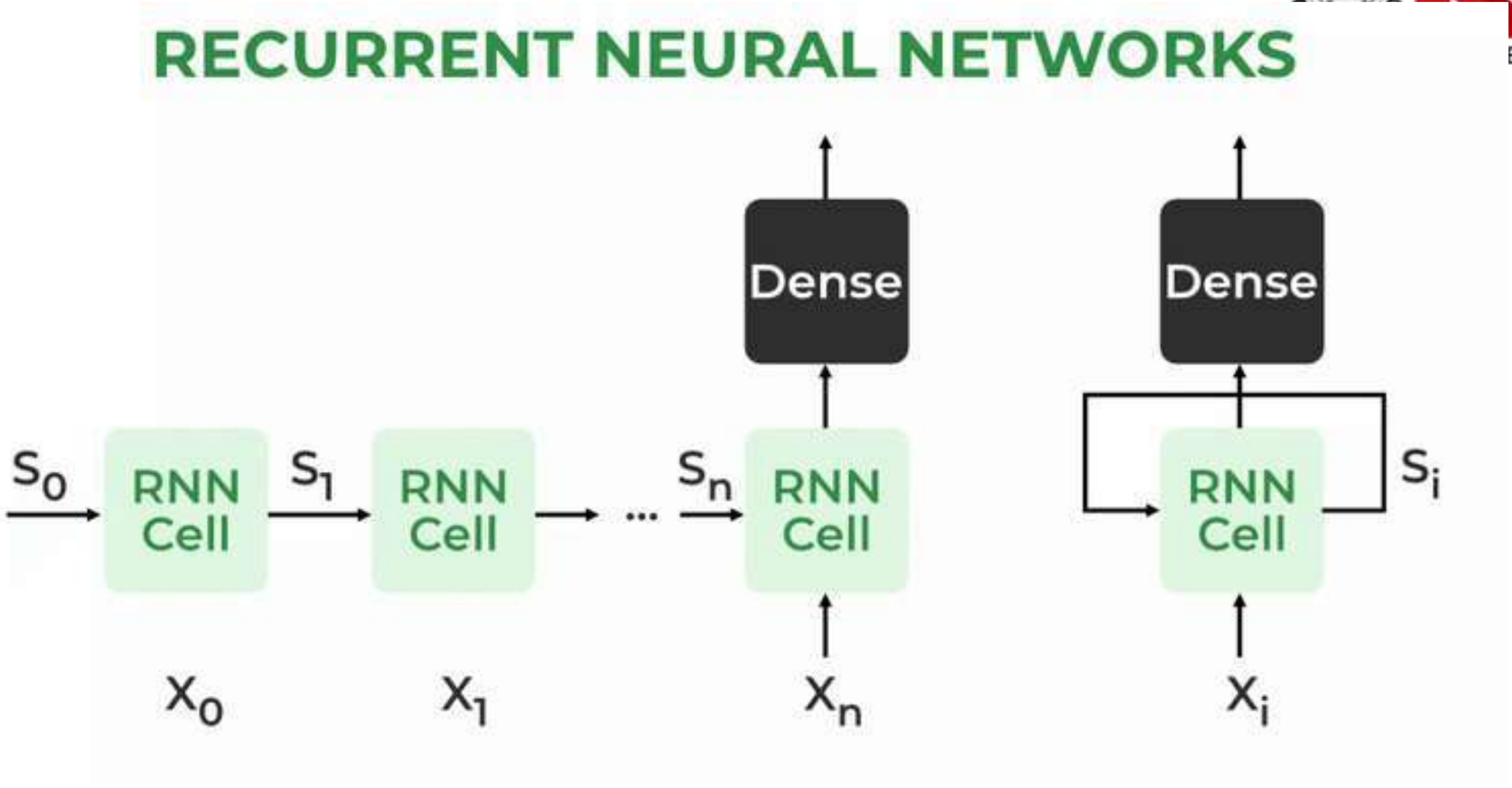
•**Input Gate**: Controls how much new information should be added to the cell state.

•**Forget Gate**: Decides what past information should be discarded.

•**Output Gate**: Regulates what information should be output at the current step. This selective memory enables LSTMs to handle long-term dependencies, making them ideal for tasks where earlier context is critical.

4. Gated Recurrent Units (GRUs)

- simplify LSTMs by combining the input and forget gates into a single update gate and streamlining the output mechanism.
- This design is computationally efficient, often performing similarly to LSTMs, and is useful in tasks where simplicity and faster training are beneficial.

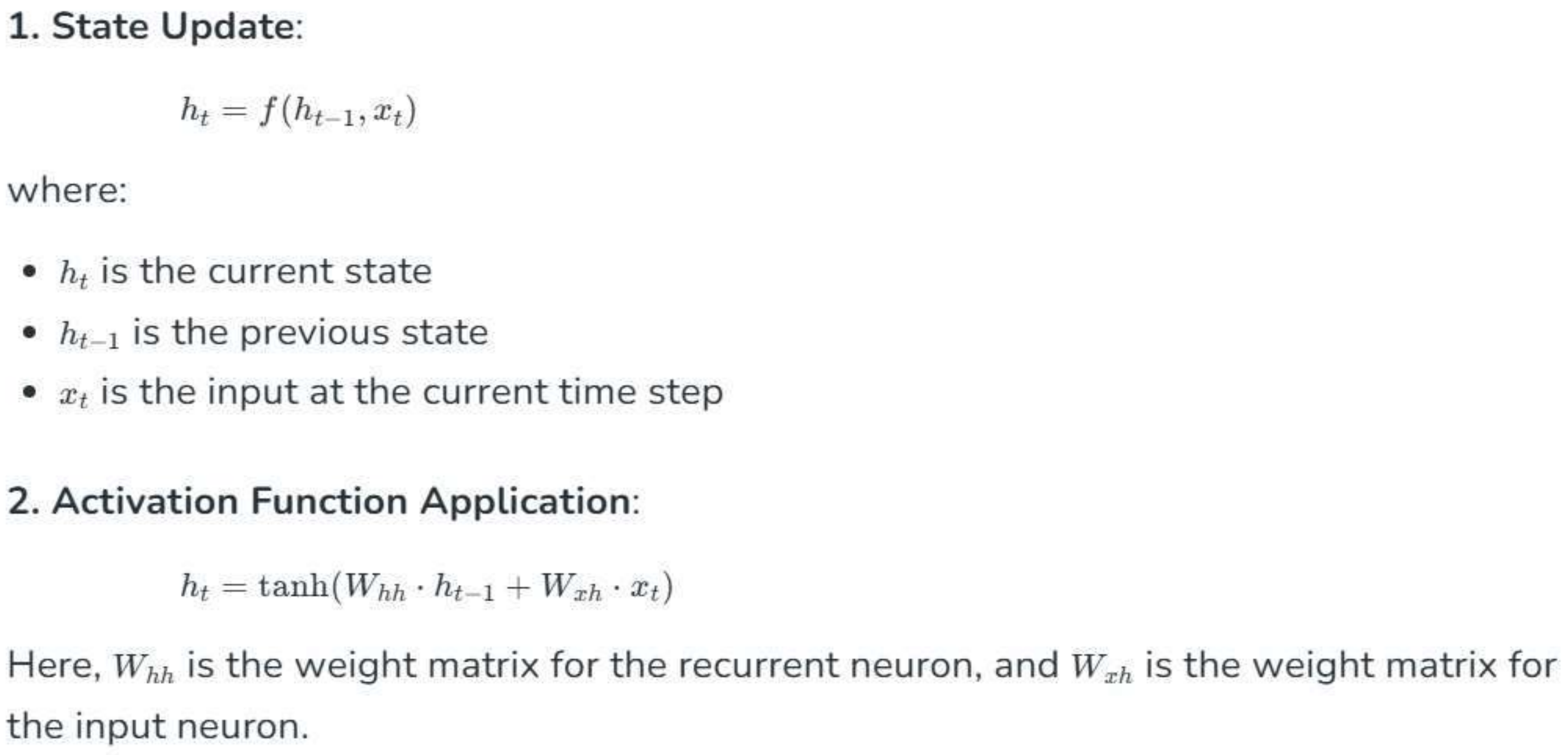

RECURRENT NEURAL NETWORKS
Dense
Dense
S_0
RNN Cell
S_1
RNN Cell
...
S_n
RNN Cell
RNN Cell
S_i
X_0
X_1
X_n
X_i

How does RNN work?

- In a RNN, each time step consists of units with a fixed activation function.
- Each unit contains an internal hidden state, which acts as memory by retaining information from previous time steps, thus allowing the network to store past knowledge.
- The hidden state h_{t-1} is updated at each time step to reflect new input, adapting the network’s understanding of previous inputs.

Updating the Hidden State in RNNs

- The current hidden state *ht* depends on the previous state *ht*−1 and the current input *xt*, and is calculated using the following relations:



1. State Update:

$$h_t = f(h_{t-1}, x_t)$$

where:

- h_t is the current state
- h_{t-1} is the previous state
- x_t is the input at the current time step

2. Activation Function Application:

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

Here, W_{hh} is the weight matrix for the recurrent neuron, and W_{xh} is the weight matrix for the input neuron.

3. Output Calculation:

$$y_t = W_{hy} \cdot h_t$$

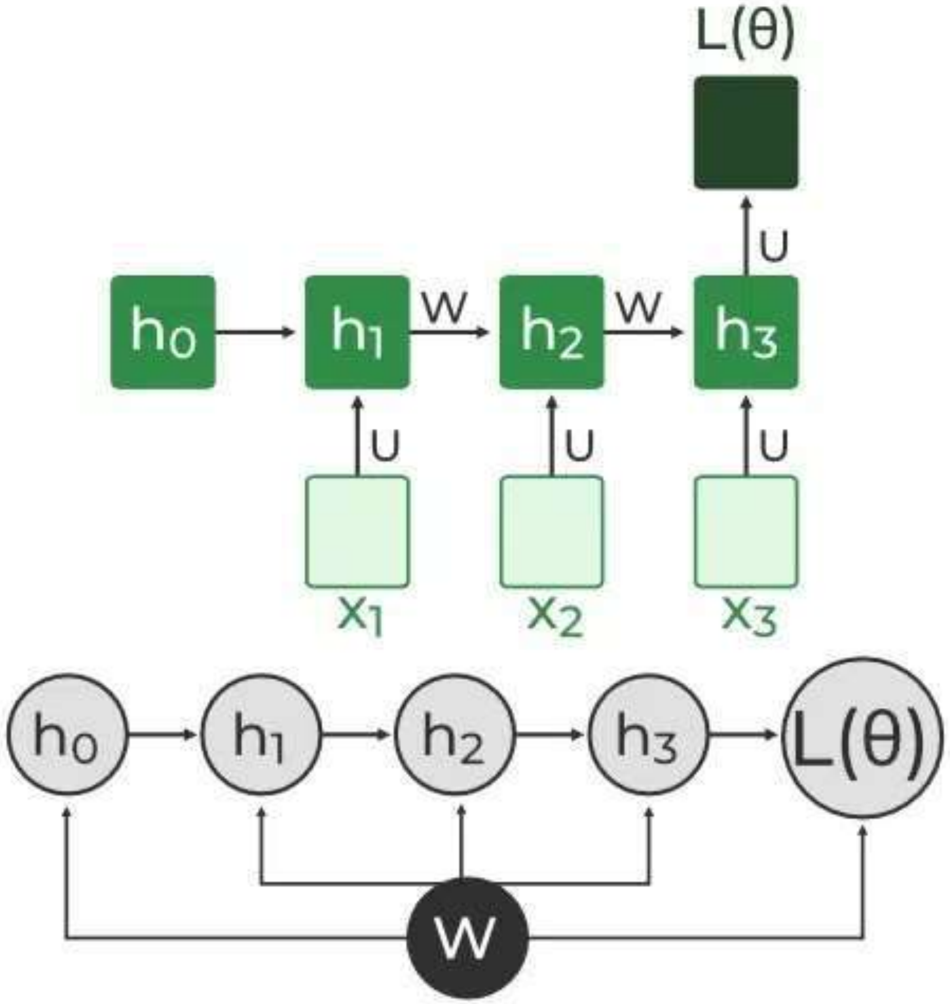
where y_t is the output and W_{hy} is the weight at the output layer.

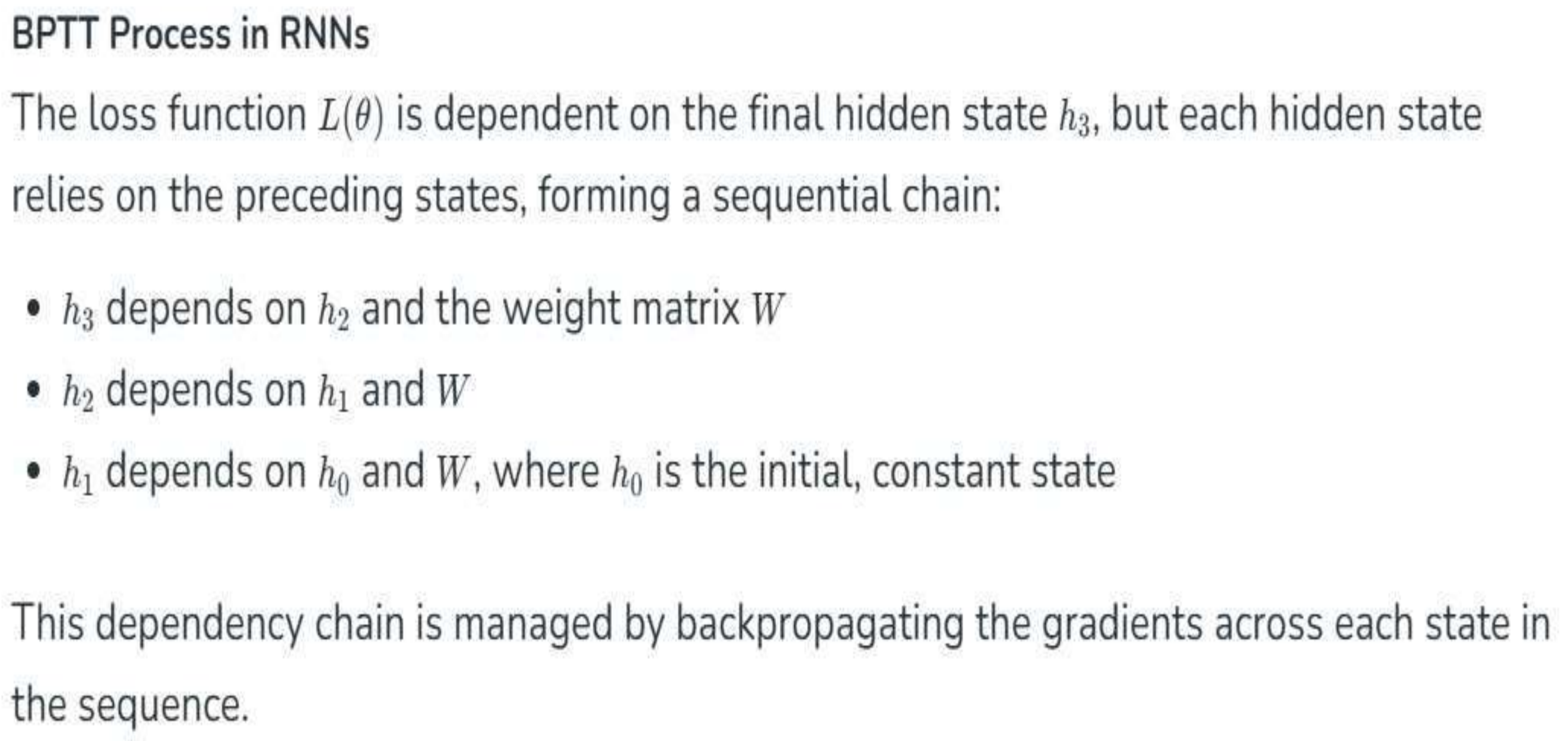
These parameters are updated using backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as ***backpropagation through time***.

Backpropagation Through Time (BPTT) in RNNs

- In a Recurrent Neural Network (RNN), data flows sequentially, where each time step’s output depends on the previous time step.
- This ordered data structure necessitates applying backpropagation across all hidden states, or time steps, in sequence.
- This unique approach is called **Backpropagation Through Time (BPTT)**, essential for updating network parameters that rely on temporal dependencies.

Backpropagation Through Time (BPTT) In RNN



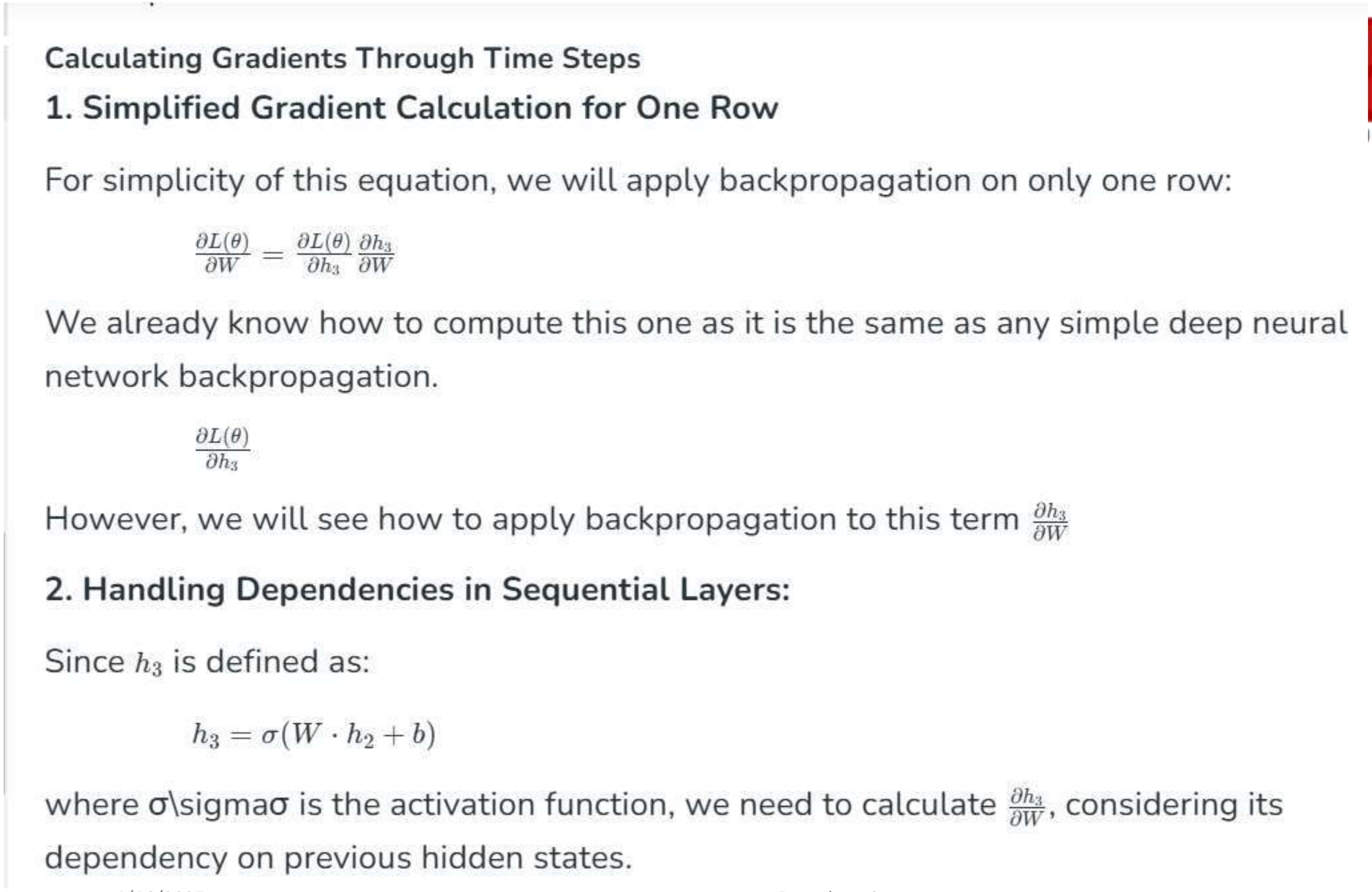


BPTT Process in RNNs

The loss function $L(\theta)$ is dependent on the final hidden state h_3, but each hidden state relies on the preceding states, forming a sequential chain:

- h_3 depends on h_2 and the weight matrix W
- h_2 depends on h_1 and W
- h_1 depends on h_0 and W, where h_0 is the initial, constant state

This dependency chain is managed by backpropagating the gradients across each state in the sequence.



Calculating Gradients Through Time Steps

1. Simplified Gradient Calculation for One Row

For simplicity of this equation, we will apply backpropagation on only one row:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \frac{\partial h_3}{\partial W}$$

We already know how to compute this one as it is the same as any simple deep neural network backpropagation.

$$\frac{\partial L(\theta)}{\partial h_3}$$

However, we will see how to apply backpropagation to this term $\frac{\partial h_3}{\partial W}$

2. Handling Dependencies in Sequential Layers:

Since h_3 is defined as:

$$h_3 = \sigma(W \cdot h_2 + b)$$

where σ\sigmaσ is the activation function, we need to calculate $\frac{\partial h_3}{\partial W}$, considering its dependency on previous hidden states.

3. Gradient Calculation with Explicit and Implicit Parts:

The total derivative of h_3 with respect to WWW is broken into:

- **Explicit:** $\frac{\partial h_3+}{\partial W}$, treating all other inputs as constants
- **Implicit:** Summing over all indirect paths from h_3 to W

Thus, we calculate:

$$\begin{aligned}\frac{\partial h_3}{\partial W} &= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W}\\ &= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\left[\frac{\partial h_2^+}{\partial W} + \frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W}\right]\\ &= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\left[\frac{\partial h_1^+}{\partial W}\right]\end{aligned}$$

4. Short-Circuiting Paths for Simplification:

For simplicity, we reduce the paths by short-circuiting, yielding:

$$\frac{\partial h_3}{\partial W} = \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2^+}{\partial W} + \frac{\partial h_3}{\partial h_1}\frac{\partial h_1^+}{\partial W}$$

5. Final Gradient Expression:

The total derivative of the loss function with respect to W then becomes:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \frac{\partial h_3}{\partial W}$$

Where:

$$\frac{\partial h_3}{\partial W} = \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

and the final expression becomes:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

This algorithm is called backpropagation through time (BPTT) as we backpropagate over all previous time steps.

Issues of Standard RNNs

Vanishing Gradient:

- Text generation, machine translation, and stock market prediction are examples of time-dependent and sequential data problems modeled with RNNs.
- Training RNNs is difficult due to the vanishing gradient problem, where gradients decay over time.

Exploding Gradient:

- Occurs when the gradient grows exponentially during training.
- Large error gradients cause very large updates to the neural network model weights, leading to instability.

TERMINAL QUESTIONS

Describe how the hidden state in Recurrent Neural Networks (RNNs) contributes to handling sequential data.

Explain why RNNs are more effective than feedforward neural networks for tasks like language modeling and time-series prediction.

Illustrate how different types of RNNs (One to Many, Many to One, Many to Many) can be applied to specific real-world problems.

Analyze the impact of using a Recurrent Unit instead of a traditional neuron in processing sequential inputs.

Compare the advantages and limitations of standard RNNs with Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU).

REFERENCES FOR FURTHER LEARNING OF THE SESSION

Books:

1 Ian Goodfellow and Yoshua Bengio and Aaron Courville (2016) Deep Learning Book
2.Deep Learning Book. eep Learning with Python, Francois Chollet , Manning publications, 2018