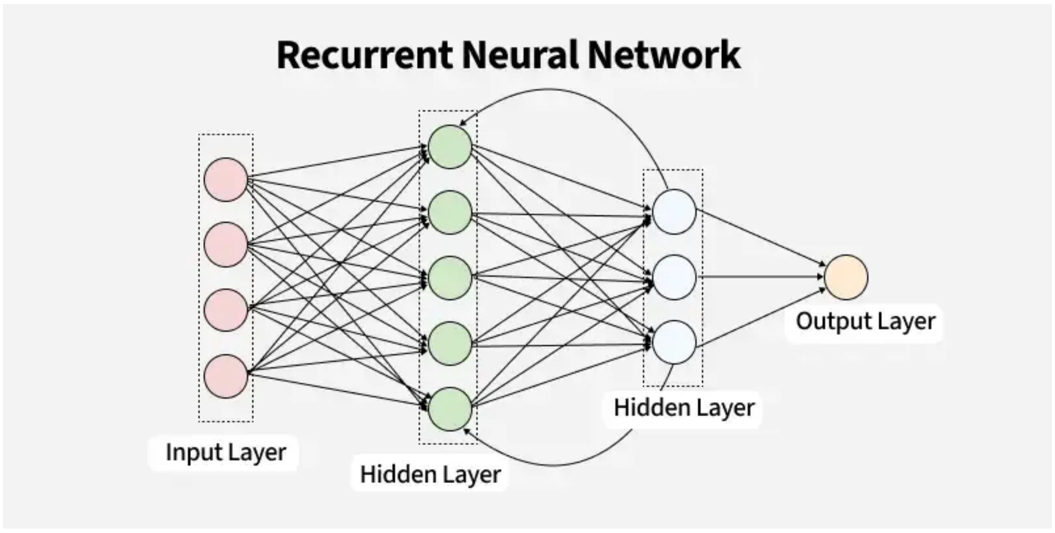
Recurrent Neural Networks (RNNs)

Architecture

RNNs are designed to process **sequential data** (e.g., time series, text, audio). Unlike feedforward neural networks, RNNs have **loops** that allow information to **persist over time**.

A recurrent neural network (RNN) is a type of artificial neural network designed to process sequential data by using a feedback loop to retain information from previous steps. This "memory" allows RNNs to understand context and make predictions based on the order of elements, making them ideal for tasks like natural language processing, speech recognition, and time series forecasting



Components:

- **Input sequence**: $x_1, x_2, \ldots, x_T$
- **Hidden state**: h_t captures past information at time t
- **Output**: y_t (optional per timestep or at end)

Update equations:
At each time step t:

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

$$y_t = W_{hy}h_t + b_y$$

- W_{xh}, W_{hh}, W_{hy}: weight matrices
- b_h, b_y: biases
- $\tanh$: non-linear activation function

This loop structure allows RNNs to **retain memory** over time by passing hidden state information from one step to the next.

Key features and concepts

- **Sequential data processing:**

  Unlike traditional networks that treat each input independently, RNNs are built for data where the order matters, such as text, speech, or time series data.

- **Memory and feedback loops:**

  An RNN takes the output from a previous step and feeds it back in as an input for the next step. This creates a hidden state that acts as a memory, carrying information across time.

- **Contextual understanding:**

  The ability to remember past information allows the network to understand the context of the data. For example, in language, it can use previous words to predict the next word in a sentence.

- **Applications:**
  RNNs are used in a wide range of applications, including machine translation (like Google Translate), speech recognition (like Siri), and time series forecasting for things like stock prices or weather.

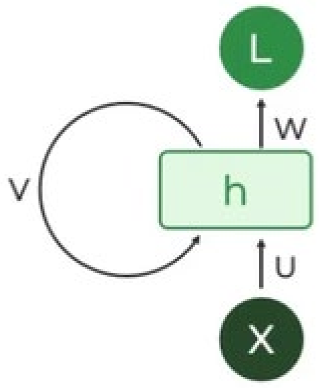
Lets understand RNN with a example:

Imagine reading a sentence and you try to predict the next word, you don't rely only on the current word but also remember the words that came before. RNNs work similarly by "remembering" past information and passing the output from one step as input to the next i.e it considers all the earlier words to choose the most likely next word. This memory of previous steps helps the network understand context and make better predictions.

Key Components of RNNs
There are mainly two components of RNNs that we will discuss.

1. Recurrent Neurons

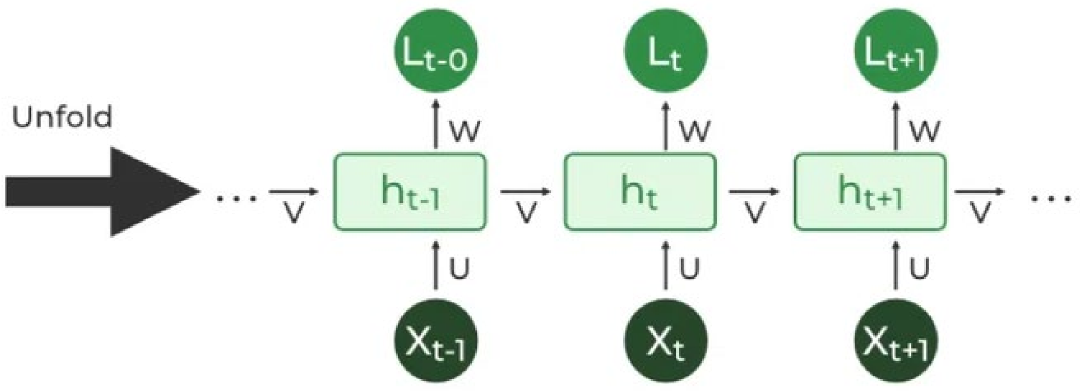
The fundamental processing unit in RNN is a Recurrent Unit**.** They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



Recurrent Neuron

2. RNN Unfolding

RNN unfolding or unrolling is the 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. This unrolling enables backpropagation through time (BPTT) a learning process where errors are propagated across time steps to adjust the network's weights enhancing the RNN's ability to learn dependencies within sequential data.



RNN Unfolding

Recurrent Neural Network Architecture

RNNs share similarities in input and output structures with other deep learning architectures but differ significantly in how information flows from input to output. Unlike traditional deep neural networks where each dense layer has distinct weight matrices. RNNs use shared weights across time steps, allowing them to remember information over sequences.

In RNNs the hidden state H_i is calculated for every input X_i to retain sequential dependencies. The computations follow these core formulas:

1. Hidden State Calculation:

$$h = \sigma(U \cdot X + W \cdot h_{t-1} + B)$$

Here:

- h represents the current hidden state.
- U and W are weight matrices.
- B is the bias.

2. Output Calculation:

$$Y = O(V \cdot h + C)$$

2. Output Calculation:

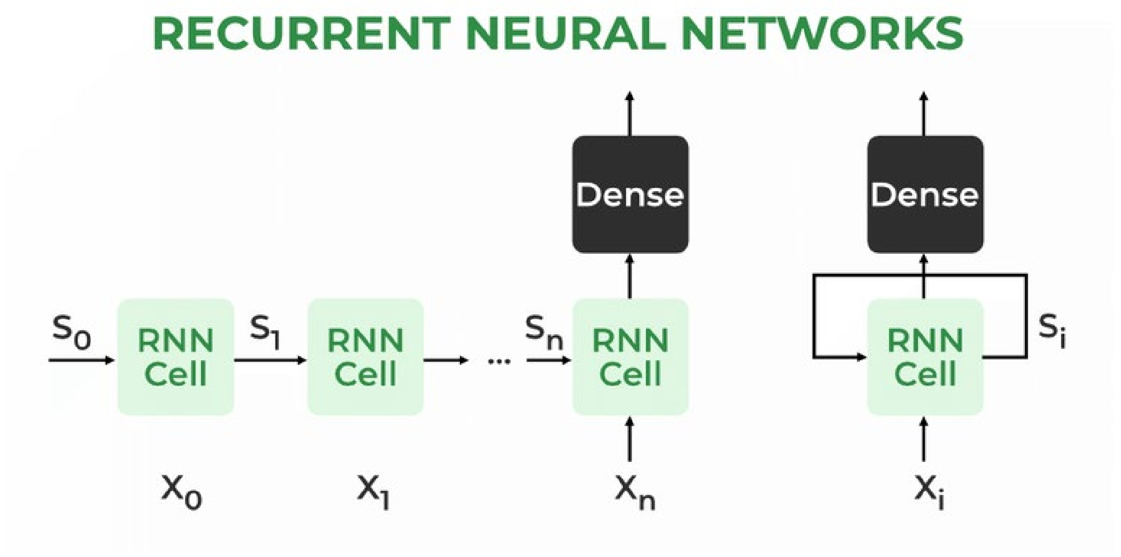
$$Y = O(V \cdot h + C)$$

The output Y is calculated by applying O an activation function to the weighted hidden state where V and C represent weights and bias.

3. Overall Function:

$$Y = f(X, h, W, U, V, B, C)$$

This function defines the entire RNN operation where the state matrix S holds each element s_i representing the network's state at each time step i.



Recurrent Neural Architecture

How does RNN work?

At each time step RNNs process units with a fixed activation function. These units have an internal hidden state that acts as memory that retains information from previous time steps. This memory allows the network to store past knowledge and adapt based on new inputs.

Updating the Hidden State in RNNs

The current hidden state h_t depends on the previous state h_{t-1} and the current input x_t 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:

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

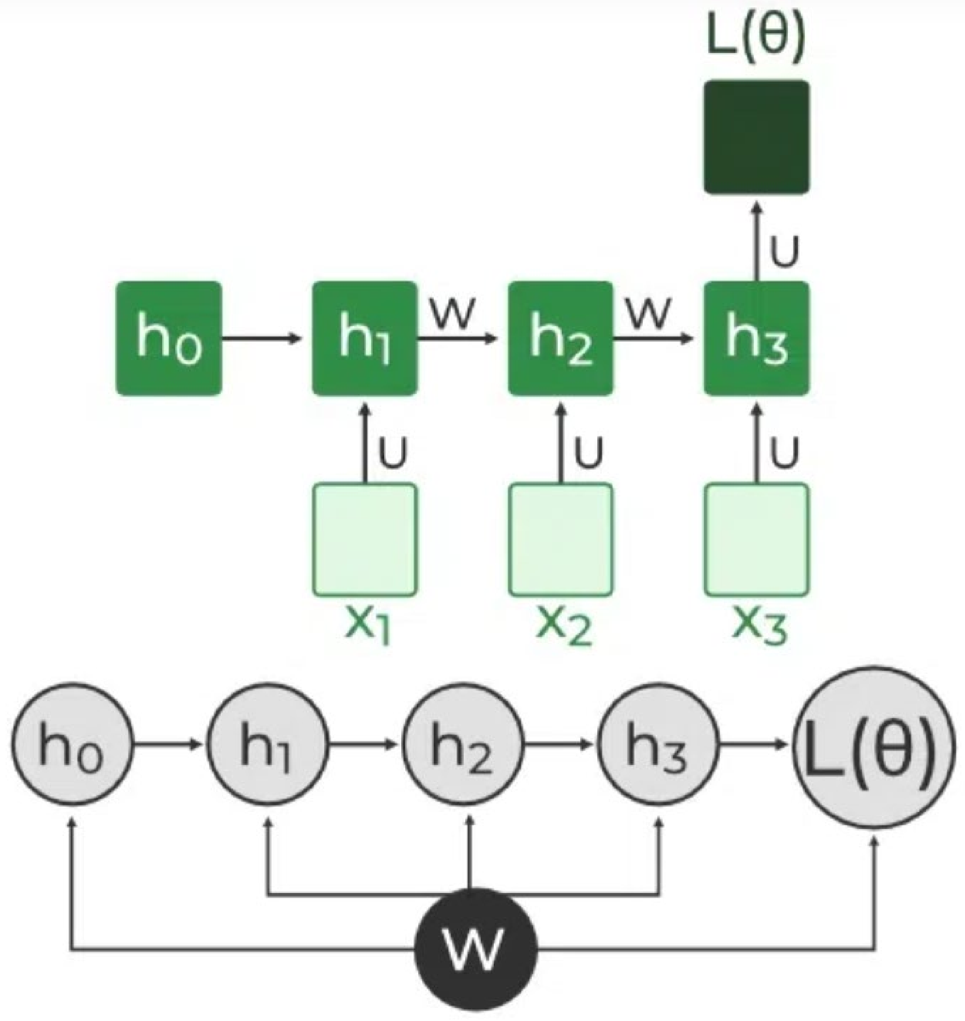
where y_t is the output and W_{hy} is the weight at the output layer.

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

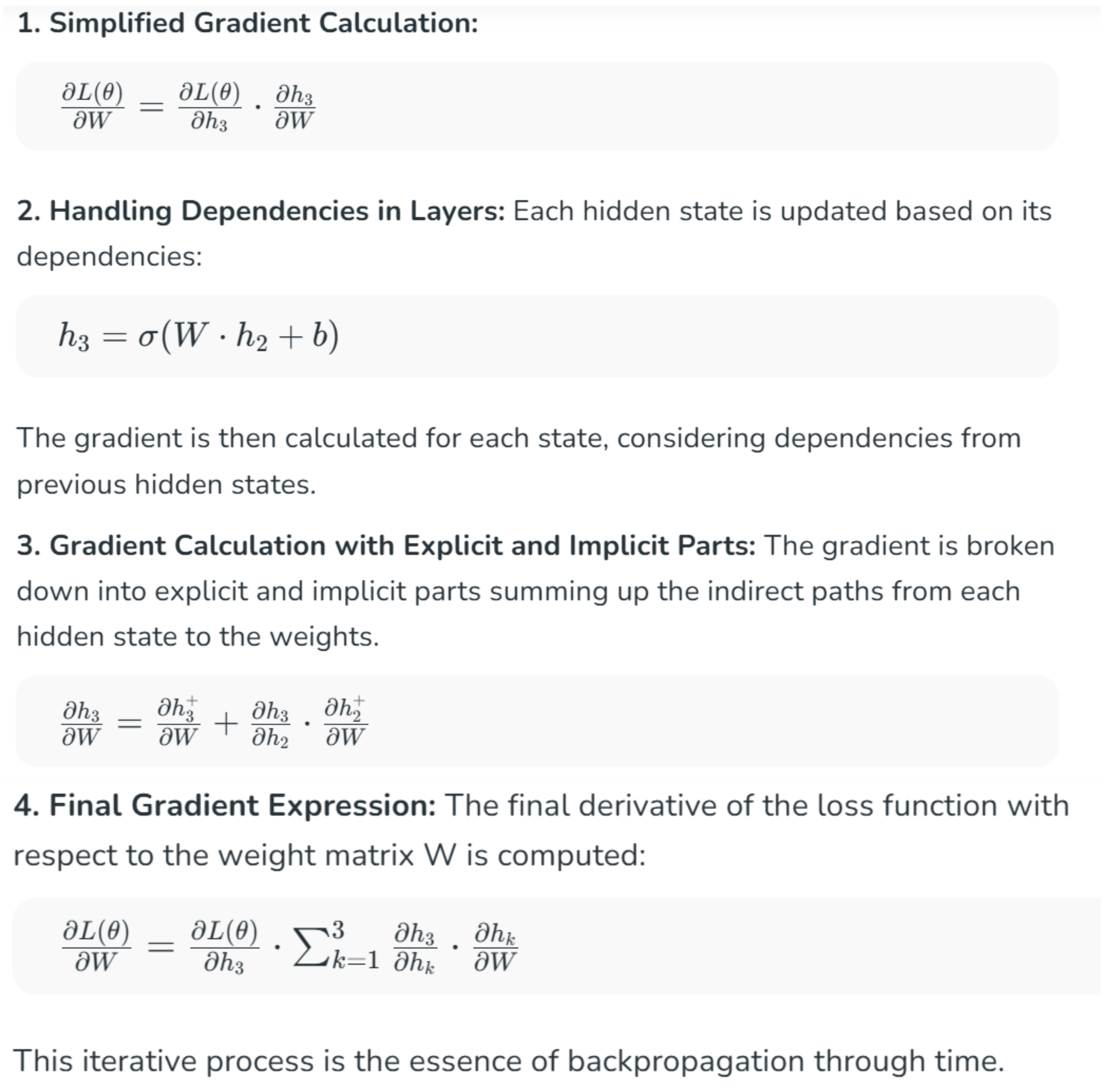
Since RNNs process sequential data Backpropagation Through Time (BPTT) is used to update the network's parameters. The loss function L(θ) depends on the final hidden state h_3 and each hidden state relies on preceding ones forming a sequential dependency chain:

h_3 depends on depends on h_2, h_2 depends on $h_1, \ldots, h_1$ depends on h_0.



Backpropagation Through Time (BPTT) In RNN

In BPTT, gradients are backpropagated through each time step. This is essential for updating network parameters based on temporal dependencies.



1. Simplified Gradient Calculation:

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

2. Handling Dependencies in Layers: Each hidden state is updated based on its dependencies:

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

The gradient is then calculated for each state, considering dependencies from previous hidden states.

3. Gradient Calculation with Explicit and Implicit Parts: The gradient is broken down into explicit and implicit parts summing up the indirect paths from each hidden state to the weights.

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

4. Final Gradient Expression: The final derivative of the loss function with respect to the weight matrix W is computed:

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

This iterative process is the essence of backpropagation through time.

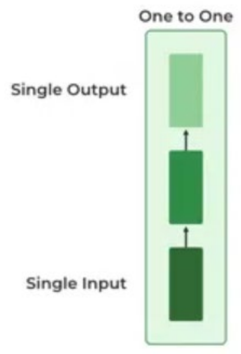
Types Of Recurrent Neural Networks

There are four types of RNNs based on the number of inputs and outputs in the network:

1. One-to-One RNN

This is the simplest type of neural network architecture where there is a single input and a single output. It is used for straightforward classification tasks such as binary classification where no sequential data is involved.

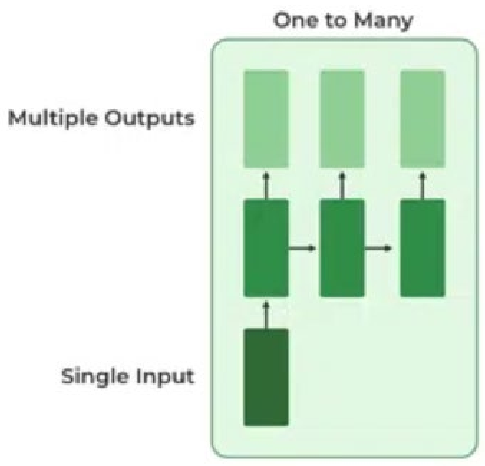
One to One RNN



One to One RNN

2. One-to-Many RNN

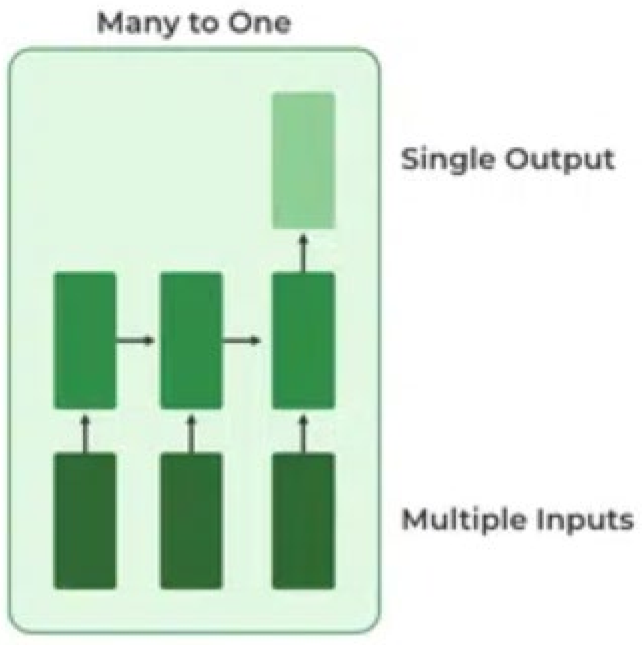
In a One-to-Many RNN the network processes a single input to produce multiple outputs over time. This is useful in tasks where one input triggers a sequence of predictions (outputs). For example in image captioning a single image can be used as input to generate a sequence of words as a caption.
One to Many RNN



One to Many RNN

3. Many-to-One RNN

The 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 like positive, negative or neutral.

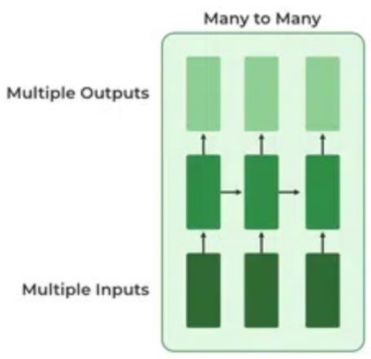


Many to One RNN

Many to One RNN

4. Many-to-Many RNN

The Many-to-Many RNN type processes a sequence of inputs and generates a sequence of outputs. In language translation task a sequence of words in one language is given as input and a corresponding sequence in another language is generated as output.



Many to Many RNN

Variants of Recurrent Neural Networks (RNNs)

There are several variations of RNNs, each designed to address specific challenges or optimize for certain tasks:

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)

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.

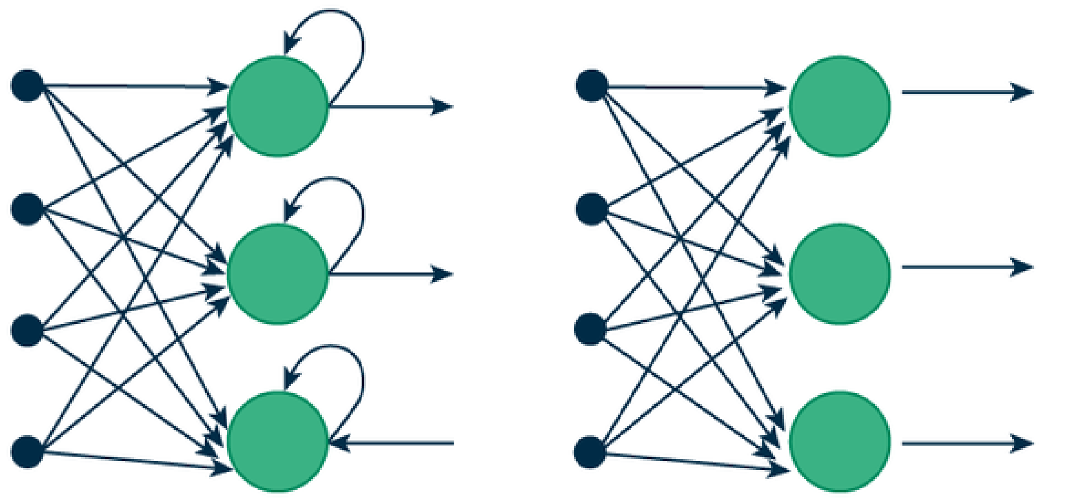
4. Gated Recurrent Units (GRUs)

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.

How RNN Differs from Feedforward Neural Networks?

Feedforward Neural Networks (FNNs) process data in one direction from input to output without retaining information from previous inputs. This makes them suitable for tasks with independent inputs like image classification. However FNNs struggle with sequential data since they lack memory.

Recurrent Neural Networks (RNNs) solve this by incorporating loops that allow information from previous steps to be fed back into the network. This feedback enables RNNs to remember prior inputs making them ideal for tasks where context is important.



ERxample:

Implementing a Text Generator Using Recurrent Neural Networks (RNNs)

In this section, we create a character-based text generator using Recurrent Neural Network (RNN) in TensorFlow and Keras. We'll implement an RNN that learns patterns from a text sequence to generate new text character-by-character.

1. Importing Necessary Libraries

We start by importing essential libraries for data handling and building the neural network.

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import SimpleRNN, Dense
```

2. Defining the Input Text and Prepare Character Set

We define the input text and identify unique characters in the text which we'll encode for our model.

```
text = "This is GeeksforGeeks a software training institute"

chars = sorted(list(set(text)))

char_to_index = {char: i for i, char in enumerate(chars)}

index_to_char = {i: char for i, char in enumerate(chars)}
```

3. Creating Sequences and Labels

To train the RNN, we need sequences of fixed length (seq_length) and the character following each sequence as the label.

```
seq_length = 3

sequences = []
```

```
labels = []

for i in range(len(text) - seq_length):
    seq = text[i:i + seq_length]
    label = text[i + seq_length]
    sequences.append([char_to_index[char] for char in seq])
    labels.append(char_to_index[label])

X = np.array(sequences)
y = np.array(labels)
```

4. Converting Sequences and Labels to One-Hot Encoding

For training we convert X and y into one-hot encoded tensors.

```
X_one_hot = tf.one_hot(X, len(chars))
y_one_hot = tf.one_hot(y, len(chars))
```

5. Building the RNN Model

We create a simple RNN model with a hidden layer of 50 units and a Dense output layer with softmax activation.

```
model = Sequential()
model.add(SimpleRNN(50, input_shape=(seq_length, len(chars)), activation='relu'))
model.add(Dense(len(chars), activation='softmax'))
```

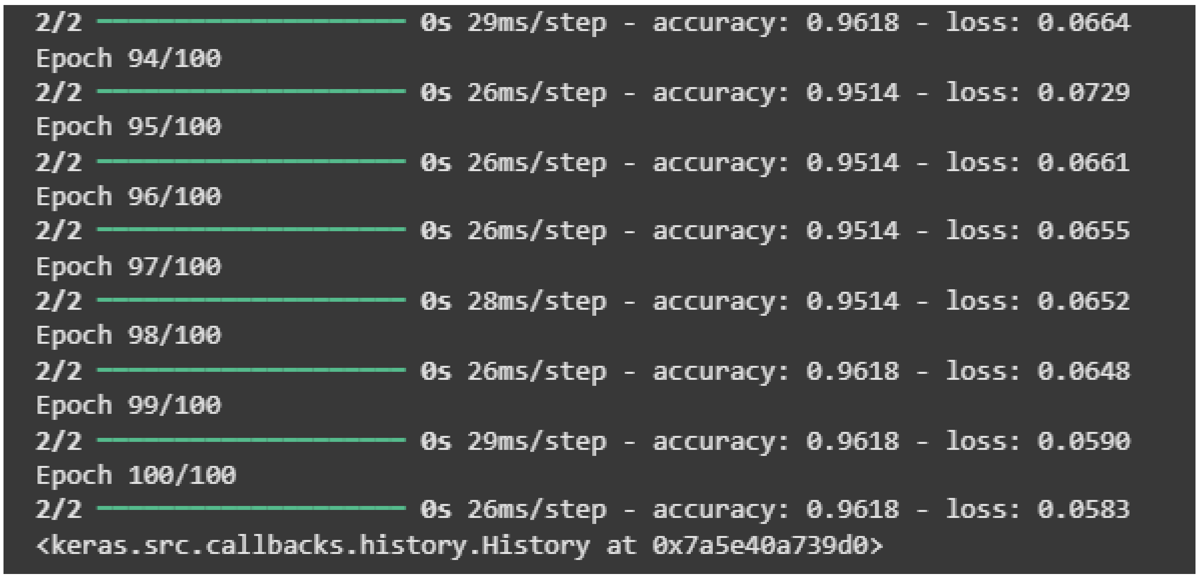
6. Compiling and Training the Model

We compile the model using the categorical_crossentropy loss and train it for 100 epochs.

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model.fit(X_one_hot, y_one_hot, epochs=100)

Output:



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.9618 - loss: 0.0664
Epoch 94/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9514 - loss: 0.0729
Epoch 95/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9514 - loss: 0.0661
Epoch 96/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9514 - loss: 0.0655
Epoch 97/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.9514 - loss: 0.0652
Epoch 98/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9618 - loss: 0.0648
Epoch 99/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.9618 - loss: 0.0590
Epoch 100/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9618 - loss: 0.0583
<keras.src.callbacks.history.History at 0x7a5e40a739d0>
```

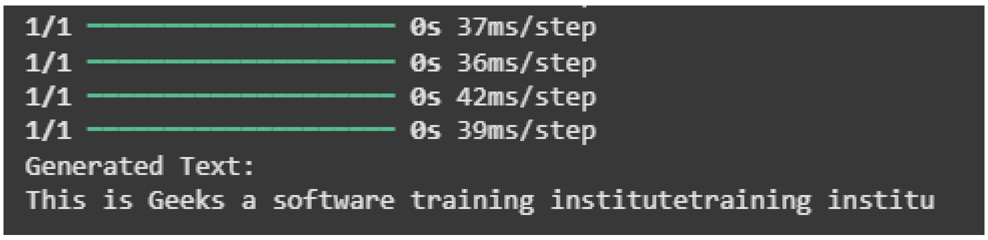
Training the RNN model

7. Generating New Text Using the Trained Model

After training we use a starting sequence to generate new text character by character.

```
start_seq = "This is G"

generated_text = start_seq


for i in range(50):

    x = np.array([[char_to_index[char] for char in generated_text[-seq_length:]]])

    x_one_hot = tf.one_hot(x, len(chars))

    prediction = model.predict(x_one_hot)

    next_index = np.argmax(prediction)

    next_char = index_to_char[next_index]
    generated_text += next_char


print("Generated Text:")

print(generated_text)
```

Output:



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
Generated Text:
This is Geeks a software training institutetraining institu
```

Predicting the next word

Advantages of RNN

- **Sequential Memory**: RNNs retain information from previous inputs making them ideal for time-series predictions where past data is crucial.
- **Enhanced Pixel Neighborhoods**: RNNs can be combined with convolutional layers to capture extended pixel neighborhoods improving performance in image and video data processing.

Limitations of RNN

While RNNs excel at handling sequential data they face two main training challenges i.e vanishing gradient and exploding gradient problem:

1. **Vanishing Gradient**: During backpropagation gradients diminish as they pass through each time step leading to minimal weight updates. This limits the RNN's ability to learn long-term dependencies which is crucial for tasks like language translation.
2. **Exploding Gradient**: Sometimes gradients grow uncontrollably causing excessively large weight updates that de-stabilize training.

These challenges can hinder the performance of standard RNNs on complex, long-sequence tasks.

Applications of RNN

RNNs are used in various applications where data is sequential or time-based:

- **Time-Series Prediction**: RNNs excel in forecasting tasks, such as stock market predictions and weather forecasting.
- **Natural Language Processing (NLP)**: RNNs are fundamental in NLP tasks like language modeling, sentiment analysis and machine translation.
- **Speech Recognition**: RNNs capture temporal patterns in speech data, aiding in speech-to-text and other audio-related applications.
- **Image and Video Processing**: When combined with convolutional layers, RNNs help analyze video sequences, facial expressions and gesture recognition.

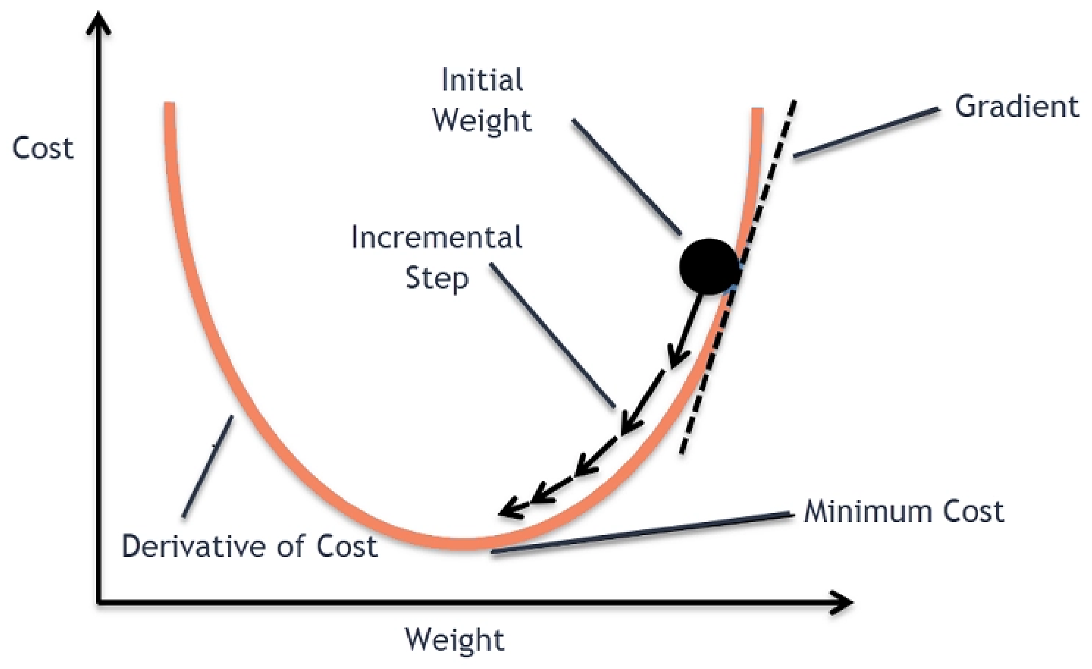
Challenges and advancements

- **Gradient problems:**

  A common challenge when training RNNs on long sequences is the vanishing gradient problem, where the network struggles to learn long-term dependencies, and the exploding gradient problem, where gradients become excessively large.
- **Advanced architectures:**
  To address these issues, more advanced architectures have been developed, including:
- **Long Short-Term Memory (LSTM) networks:** A type of RNN that uses gating mechanisms to better control the flow of information and remember long-term dependencies.
- **Gated Recurrent Units (GRU):** A simplified version of LSTM with similar performance.
- **Bidirectional LSTM (BiLSTM):** Processes sequences in both forward and backward directions to capture more context.

Gradient-based Learning:

The main aim of gradient-based learning is to minimize the loss function. A gradient is simply the slope of a function. Gradient learning aims to travel down the slope of the loss function until we reach local minima.



Backpropagation is a common method to calculate the derivatives of a neural network. It calculates the gradient of the loss function with respect to the neural network's weights. Backpropagation calculates the gradients and then feeds them back to the first layer of the network. This allows us to calculate the gradients of each layer more precisely.
We use the chain rule of calculus to calculate the gradient at a particular layer. All the gradients of the following layers are combined to calculate the gradient of a layer.

Problem with backpropagation

Backpropagation allows us to effectively train a neural network. However, it is seen that if we use a large number of layers in our network. The performance of our network decreases absurdly, and sometimes, it becomes impossible to train. This is due to vanishing or exploding gradients. It occurs due to the repeated calculation of gradients using the chain rule

Vanishing gradients

Vanishing gradient arises when we repeatedly multiply the gradient with a number smaller than one. Due to this, the gradient becomes smaller and smaller in each step and becomes infinitesimally small. Due to the small gradient, the step size approaches zero, and as a result, we can never reach the optimum position of our loss function.

Exploding gradients

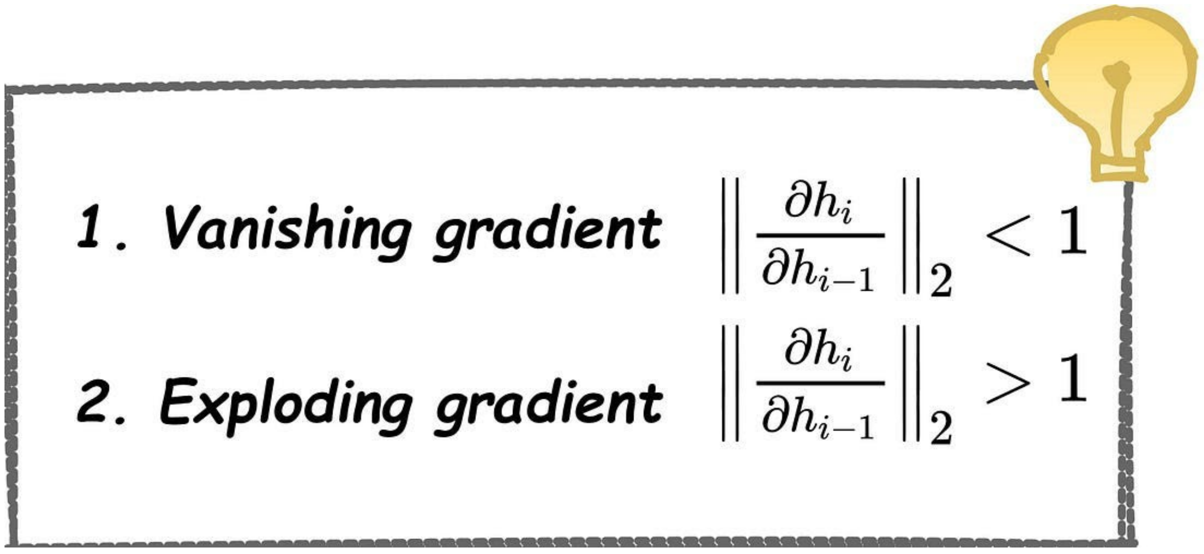
On the other hand, if we keep on multiplying the gradient with a number larger than one. The gradient keeps increasing and can become so large that it makes our network unstable. Sometimes, the parameters can become so large that they result in NaN values. NaN stands for not a number. It represents undefined values.

Vanishing Gradient Problem

Recurrent neural networks (RNNs) are a powerful type of artificial neural network (ANN) that can process sequential data, such as text, speech, or video. However, they also suffer from some common challenges, such as the vanishing and exploding gradient problems. In this article, you will learn what these problems are, why they occur, and how to deal with them using some popular techniques.

What is Vanishing and Exploding?

In the context of neural networks, the term "vanishing" usually refers to the phenomenon where the gradients of the loss function with respect to the network weights become very small as they propagate backward through the network layers during training. This can happen when the network is very deep and has many layers, especially when using activation functions that tend to produce small gradients (e.g., sigmoid or tanh).



1. Vanishing gradient $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 < 1$

2. Exploding gradient $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 > 1$

Let us take a quick look at gradient-based learning. The main aim of gradient-based learning is to minimize the loss function. A gradient is simply the slope of a function. Gradient learning aims to travel down the slope of the loss function until we reach local minima.

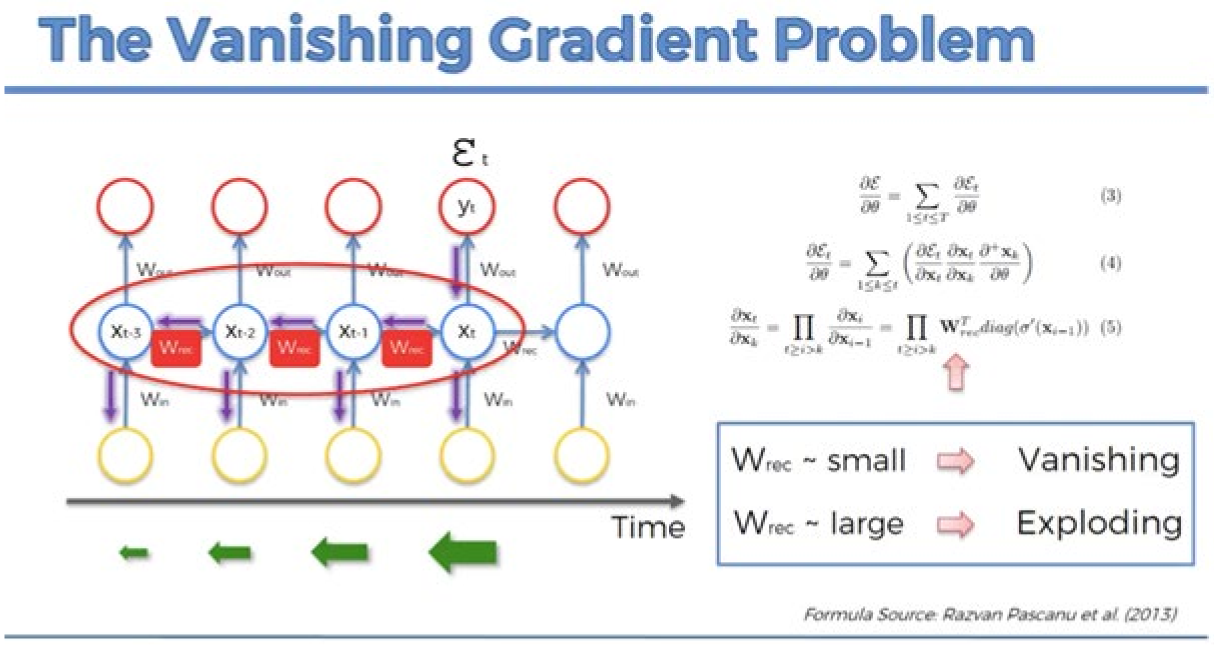
When the gradients are too small, the optimizer has difficulty updating the weights, and the network may not learn properly. This can result in slow convergence, poor performance, or even complete failure to learn.

In the context of neural networks, the term "exploding" usually refers to the phenomenon where the gradients of the loss function with respect to the network weights become very large as they propagate backward through the network layers during training. This can happen when the network is very deep and has many layers, especially when using activation functions that tend to produce large gradients (e.g., ReLU with a large learning rate).

When the gradients become too large, the optimizer may take large steps in weight updates, which can result in unstable and unpredictable behavior during training. This can cause the network to diverge, making it impossible to train.

Why it occurs?

1. Activation function: When using activation functions that tend to produce small gradients (e.g., sigmoid or tanh), the gradients can become very small outside of their active regions. This can cause the gradient signal to decrease as it propagates backward through the network layers, leading to vanishing gradients.
2. Depth of the network: As the number of layers in a neural network increases, the gradients can become smaller and smaller as they propagate backward through the network layers. This is because the gradient signal must pass through many layers, and each layer may cause the gradients to become smaller.


The Vanishing Gradient Problem
$\mathcal{E}_t$
y_t
W_{out}
X_{t-3}
X_{t-2}
X_{t-1}
X_t
W_{rec}
W_{in}
Time
$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \le t \le T} \frac{\partial \mathcal{E}_t}{\partial \theta}$ (3)
$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right)$ (4)
$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \ge i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1}))$ (5)
W_{rec} ~ small ⇨ Vanishing
W_{rec} ~ large ⇨ Exploding
Formula Source: Razvan Pascanu et al. (2013)

Vanishing & Exploding

How to overcome?

1. Weight initialization: Proper initialization of network weights can help prevent vanishing gradients. One popular method is to initialize the weights using Xavier initialization, which sets the weights such that the variance of the activations is preserved across layers.
2. Activation functions: Using activation functions that produce larger gradients outside of their active regions can help prevent vanishing gradients. Rectified Linear Units (ReLU) and its

variants are commonly used for this purpose, as they produce a constant gradient for positive inputs.

3. Normalization techniques: Batch normalization and layer normalization can help prevent vanishing gradients by normalizing the inputs to each layer.
4. Skip connections: Skip connections allow the gradient to bypass some layers in the network, which can help prevent vanishing gradients. This technique is commonly used in deep residual networks.

Formula:-
dE/dw = dE/dy * dy/dz * dz/dx * dx/dw

where:

- E is the error function
- w is the weight parameter
- y is the output of the layer
- z is the input to the activation function
- x is the weighted input to the layer

The vanishing gradient problem occurs when the derivative of the activation function (dy/dz) approaches zero, which can cause the derivative of the weighted input (dz/dx) and the derivative of the weight parameter (dx/dw) to also approach zero. This means that the derivative of the error function with respect to the weight parameter (dE/dw) becomes very small, making it difficult to update the weights during training.

How can we prevent or mitigate the vanishing and exploding gradient problems?

To prevent or mitigate the vanishing and exploding gradient problems in RNNs, there are several techniques that can be utilized. Gradient clipping is a simple technique that involves setting a threshold for the maximum or minimum value of the gradients. If the gradients exceed or fall below this value, they are clipped or rescaled. Weight initialization is another technique which involves choosing appropriate values for the initial weights and biases of the network. Long short-term memory (LSTM) or gated recurrent unit (GRU) cells are special types of RNN cells with internal mechanisms to regulate the flow of information and gradients. They use gates which learn to open or close depending on input and output, thus preventing irrelevant information from accumulating or important information from fading in the cell state. Additionally, these gates can control how much gradients are affected by previous inputs and outputs, thereby preventing them from vanishing or exploding.

How can we evaluate the effectiveness of these techniques?

When training RNNs using **Backpropagation Through Time (BPTT)**, gradients are propagated backward through many time steps. During this process:

- **Gradients can shrink (vanish)** or
- **Grow uncontrollably (explode)**

This is especially problematic with **long sequences**, leading to:

- **Vanishing gradients** → weights receive minimal updates → network **fails to learn long-term dependencies**
- **Exploding gradients** → numerical instability → gradients become too large
-

The **Vanishing Gradient Problem** happens when **training deep neural networks or RNNs**, especially on **long sequences**.

During **backpropagation**, gradients (used to update weights) **shrink exponentially** as they are passed backward through the layers (or time steps in RNNs). Eventually, these gradients become so **small (close to zero)** that the network **stops learning** — especially in early layers.

🔄 How It Happens (in RNNs)

RNNs use the **same weights repeatedly** across time steps. When you compute gradients to train them (via **Backpropagation Through Time – BPTT**), the gradient at each step involves **multiplying many small derivatives**.

$$\text{Gradient} \propto \prod_{t=1}^{T} \frac{\partial h_t}{\partial h_{t-1}}$$

each of these partial derivatives is a number less than 1 (like 0.5), multiplying many of them gives a **very**
nall number:

$$0.5 \times 0.5 \times 0.5 \times ... = \text{almost zero}$$

s a result:

- Gradients **vanish** (become very small)
- **Early layers (or early time steps)** receive **almost no updates**
- The network **cannot learn long-term dependencies**

Solutions to Vanishing Gradients

1. **Use LSTM or GRU**:
   - These are special RNNs designed to remember long-term information by **controlling the flow of gradients**.
   - They use **gates** to keep gradients from vanishing.
2. **Gradient Clipping**:

    - Prevents gradients from becoming too large or too small by capping their values.
3. **Better Activation Functions**:
    - Functions like **ReLU** (instead of sigmoid or tanh) can help reduce vanishing gradients in some cases.
4. **Batch Normalization**:
    - Normalizes outputs layer-by-layer to stabilize learning.
5. **Proper Weight Initialization**:
    - Techniques like **Xavier** or **He initialization** help gradients flow better during training.

LSTM (Long Short-Term Memory :

What is LSTM?

LSTM is a type of **Recurrent Neural Network (RNN)** designed to **remember information for long periods**, and to **solve the vanishing gradient problem** that standard RNNs struggle with.

LSTM Architecture Components

LSTM introduces **gates** and a **cell state**:

Component	Function
Cell state CtC_tCt	Memory of the network — flows unchanged unless gates change it
Hidden state hth_tht	Output of the LSTM at time ttt
Forget gate ftf_tft	Decides what to forget from previous memory
Input gate iti_tit	Decides what new information to store
Output gate oto_tot	Decides what part of memory to output

How LSTM Works?

In an LSTM cell, information flows through the forget, input, and output gates, each contributing to the decision-making process. This gating mechanism enables LSTMs to selectively update, retain, or discard informatin, ensuring robust handling of sequential data.

Long Short-Term Memory (LSTM) networks address the limitations of traditional RNNs by maintaining long-term dependencies through a unique gating mechanism. Here’s a step-by-step explanation of the LSTM workflow:

1. Input Processing

Each LSTM cell receives input data (x_t) at a specific time step along with the previous hidden state (h_t-1) and cell state (C_t-1). This combination allows the network to consider both current and historical information.

2. Forget Gate

The forget gate determines which parts of the previous cell state (C_t-1) should be discarded. It uses a sigmoid function to produce values between 0 and 1, where 0 means "forget this information" and 1 means "retain completely."

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

3. Input Gate

The input gate updates the cell state with new information. It comprises two parts:

- **Candidate State: A tanh layer generates potential updates (C̃_t).**
- **Update Decision: A sigmoid layer decides the importance of the candidate state.**

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

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

4. Cell State Update

The forget gate and input gate combine to update the cell state (C_t):

$$C_t = f_t * C_{t-1} + i_t * C_t$$

5. Output Gate

The output gate decides which parts of the cell state should influence the hidden state (h_t). It applies a sigmoid function to select the relevant parts, followed by a tanh transformation to produce the final hidden state:

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

$$h_t = o_t * \tanh(C_t)$$

Visualizing Information Flow

Optimizers

Optimizers adjust neural network weights to minimize the **loss function** during training. They use gradients to decide **how much** and **in what direction** to update weights.

🔹 Stochastic Gradient Descent (SGD)

- **Basic** optimization algorithm
- Update rule:

$$\theta = \theta - \eta \nabla_\theta L(\theta)$$

where:

- θ: weights
- η: learning rate
- $\nabla_\theta L(\theta)$: gradient of loss wrt weights

Pros:

- Simple and efficient for small datasets

Cons:

- Sensitive to learning rate
- Slow convergence
- Gets stuck in local minima or saddle points

Adam (Adaptive Moment Estimation)

Combines ideas from:

- **Momentum** (smooths updates)
- **RMSProp** (adapts learning rates)

Adam **Updated Steps**

1. Gradient of the Loss:

$$\nabla_\theta L(\theta)$$

- This is the **gradient** (partial derivative) of the **loss function** L with respect to the model parameters θ.
- It tells how much the weights should change to reduce the loss.

2. First Moment Estimate (Mean):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta L(\theta)$$

- m_t: moving average of gradients (like momentum)
- β_1: decay rate for the moving average (commonly 0.9)
- This smooths the gradient and reduces noise in updates.

3. Second Moment Estimate (Variance):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta L(\theta))^2$$

- v_t: moving average of **squared gradients**
- β_2: decay rate for this average (commonly 0.999)
- It tracks how large the gradients are — helps with adaptive learning rates.

4. Bias Correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- $\hat{m}_t$, $\hat{v}_t$: **bias-corrected** versions of the moving averages.
- Without this correction, early iterations would be biased toward zero because m_0 and v_0 start at 0.

5. Parameter Update Rule:

$$\theta = \theta - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

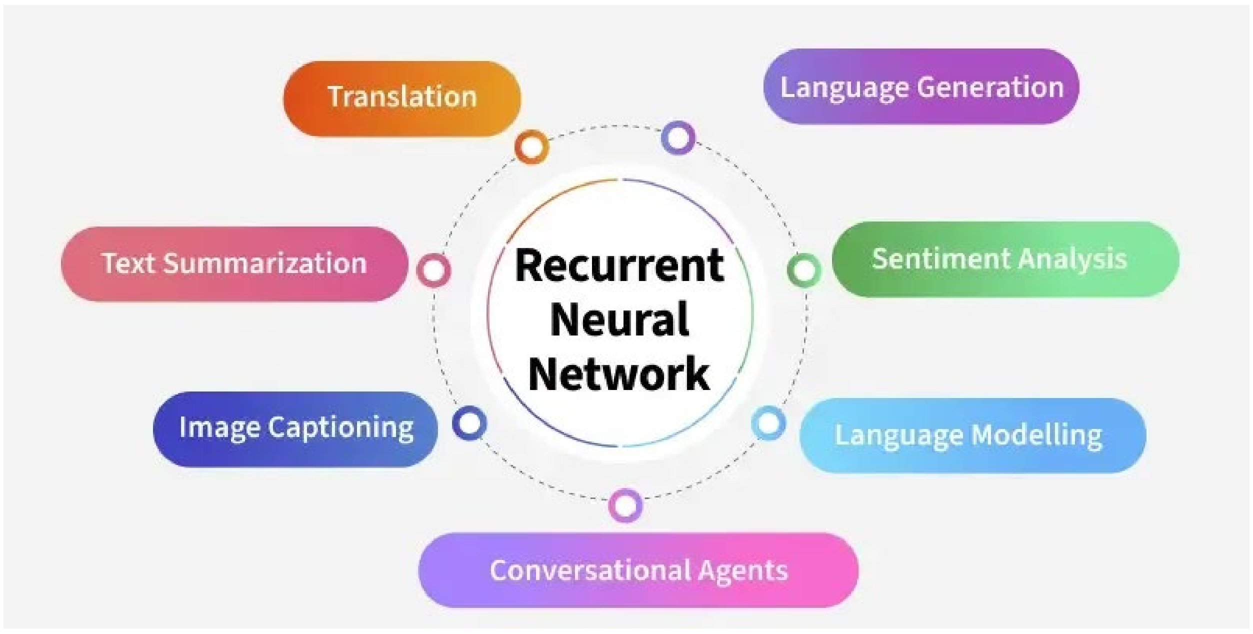
- θ: model weights (parameters)
- η: learning rate (step size)
- ϵ: small constant (e.g., 10^{-8}) to prevent division by zero
- The term $\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$ ensures **adaptive step sizes** for each parameter.

Pros:

- Fast convergence
- Adapts learning rate per parameter
- Works well with noisy data or sparse gradients

Cons:

- May not generalize as well as SGD in some cases



Summary Table

Component	Description
RNN	Processes sequential data with memory of past inputs
Vanishing Gradient	Gradients shrink during backpropagation, preventing learning
SGD	Simple optimizer, fixed learning rate
Adam	Advanced optimizer, adapts learning rate with momentum and variance