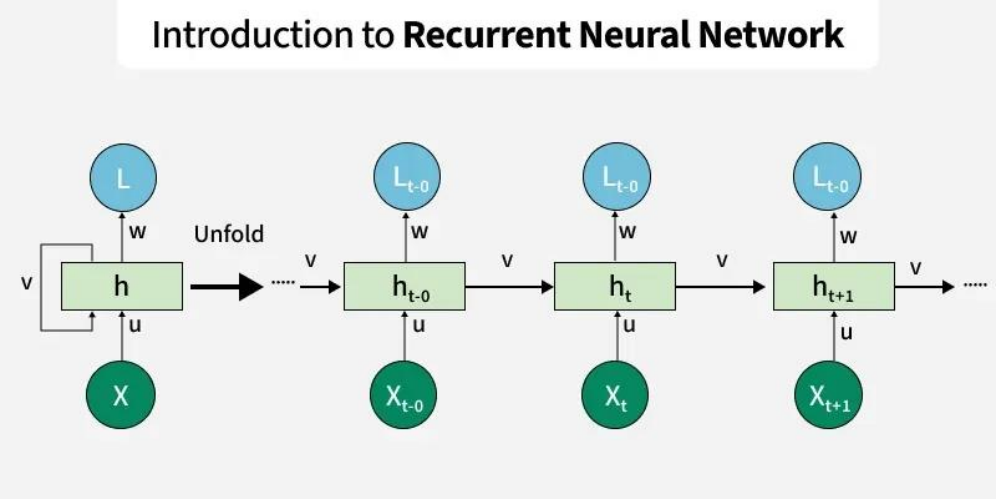
RECURRENT NEURAL NETWORKS

Recurrent Neural Networks (RNNs) are a class of neural networks designed to process sequential data by retaining information from previous steps. They are especially effective for tasks where context and order matter.

- Designed for sequential and temporal data
- Maintains memory of past inputs
- Widely used in NLP, forecasting and speech tasks



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 i.e it considers all the earlier words to choose the most likely next word.

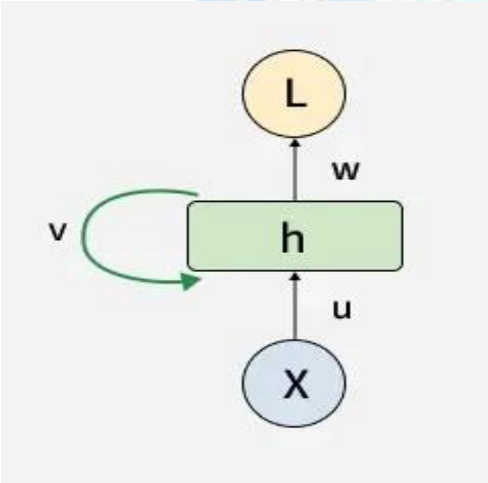
This memory of previous steps helps the network understand context and make better predictions.

Key Components of RNNs

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

1. Recurrent Neurons

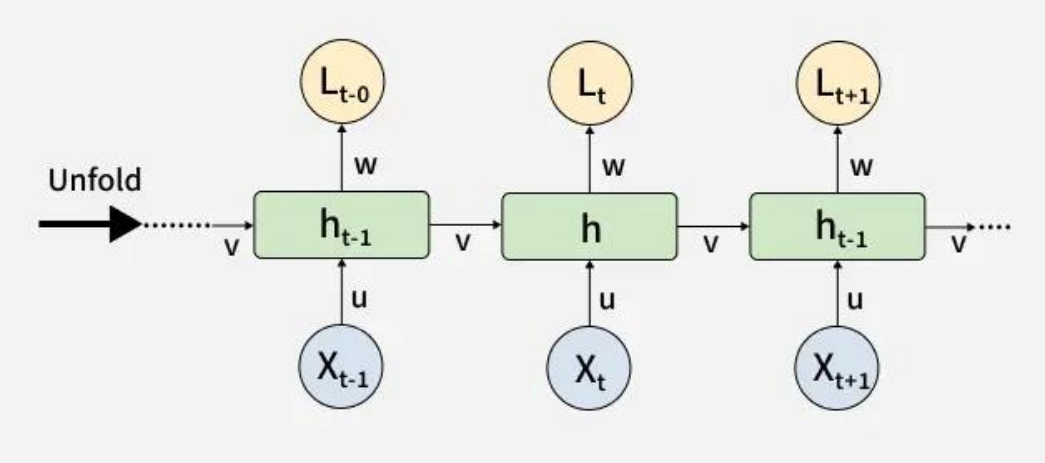
The fundamental processing unit in RNN is a Recurrent Unit**.** They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



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.



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:

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

Here:

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

2. Output Calculation:

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

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.

Working of RNN

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:

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

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:

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

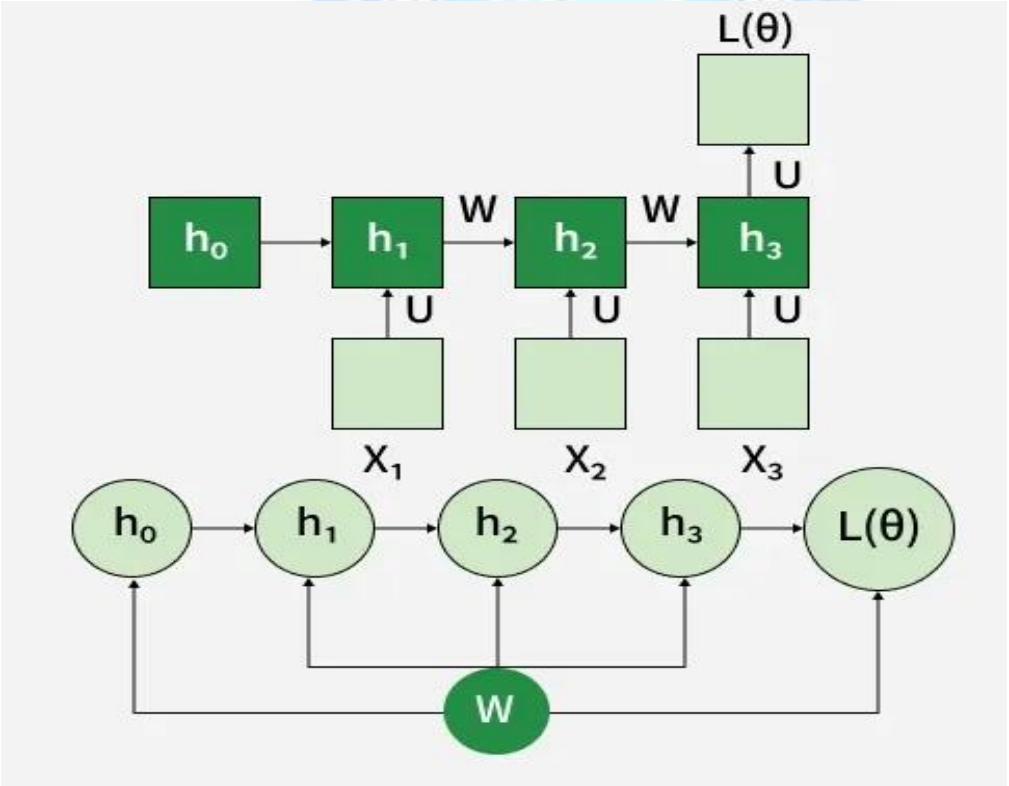
where y_t is the output and W_{hy} is the weight at the output layer.

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



Backpropagation Through Time (BPTT) in RNNs

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:

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

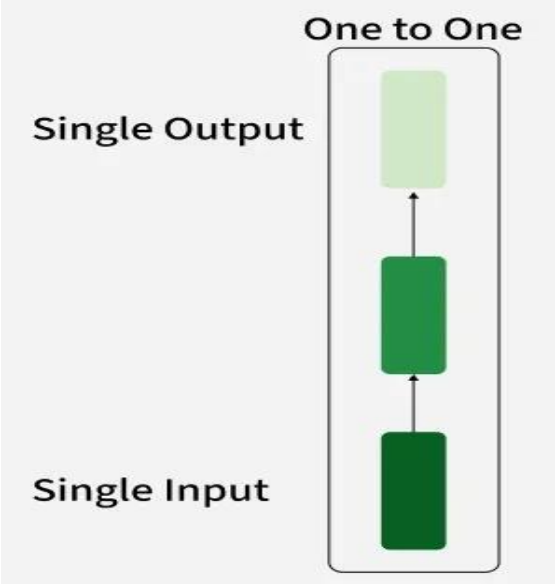
This iterative process is the essence of backpropagation through time.

Types Of Recurrent Neural Networks

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

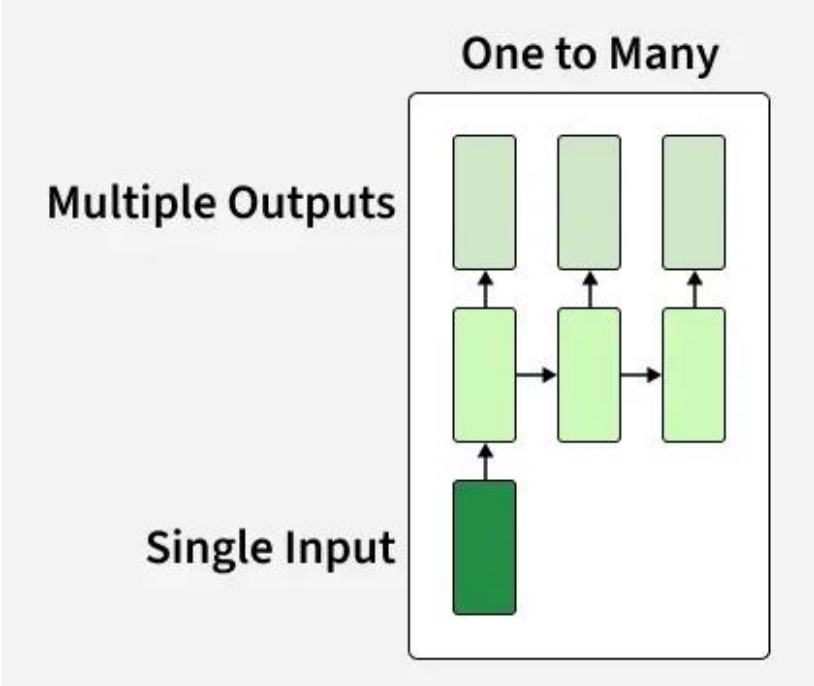
1. One-to-One RNN

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.



2. One-to-Many RNN

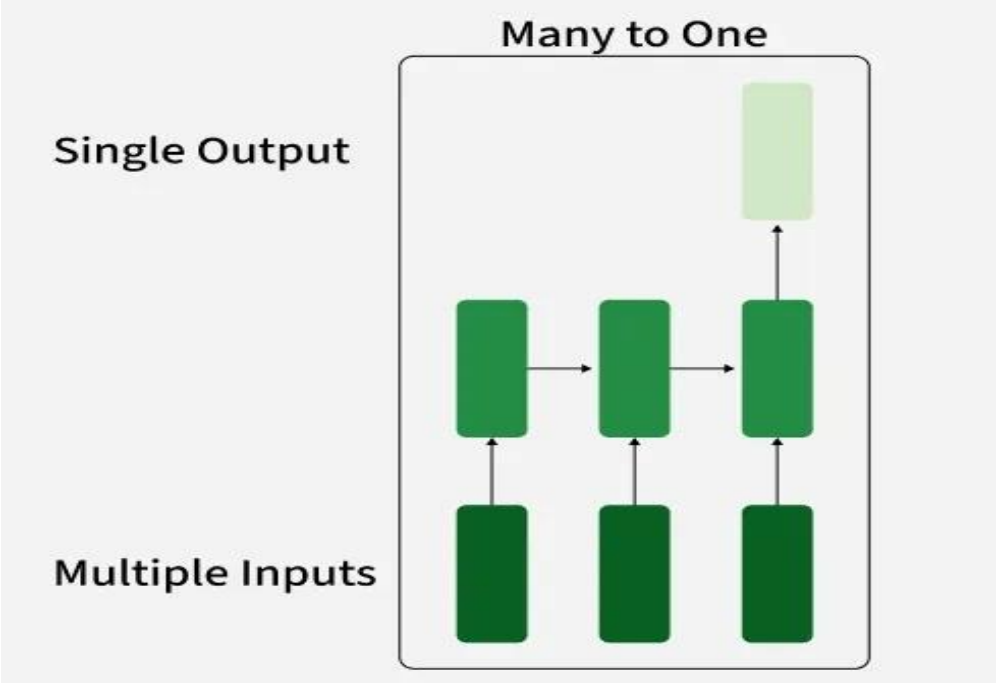
In a One-to-Many RNN the network processes a single input to produce multiple outputs over time. This is useful in tasks where one input triggers a sequence of predictions (outputs). For example in image captioning a single image can be used as input to generate a sequence of words as a caption.



3. Many-to-One RNN

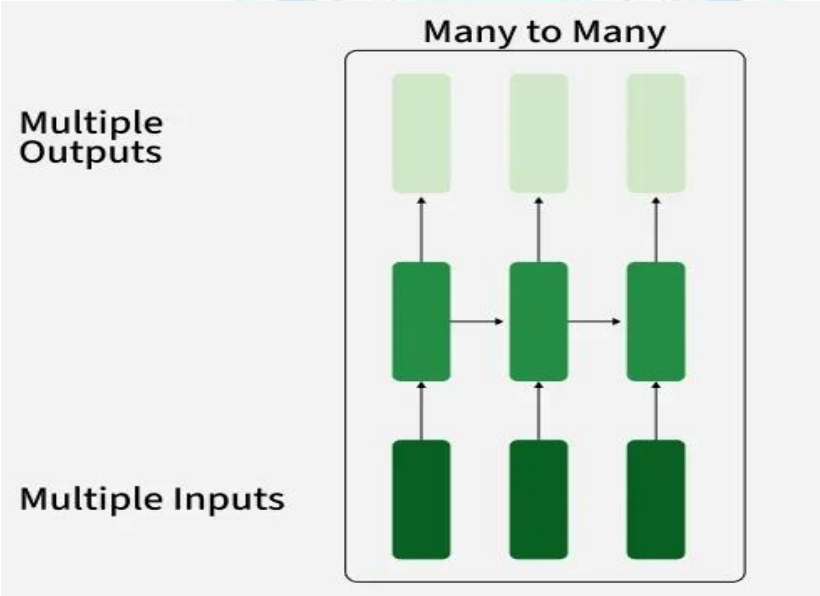
The Many-to-One RNN receives a sequence of inputs and generates a single output. This type is useful when the overall context of the input sequence is needed to make

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.



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.