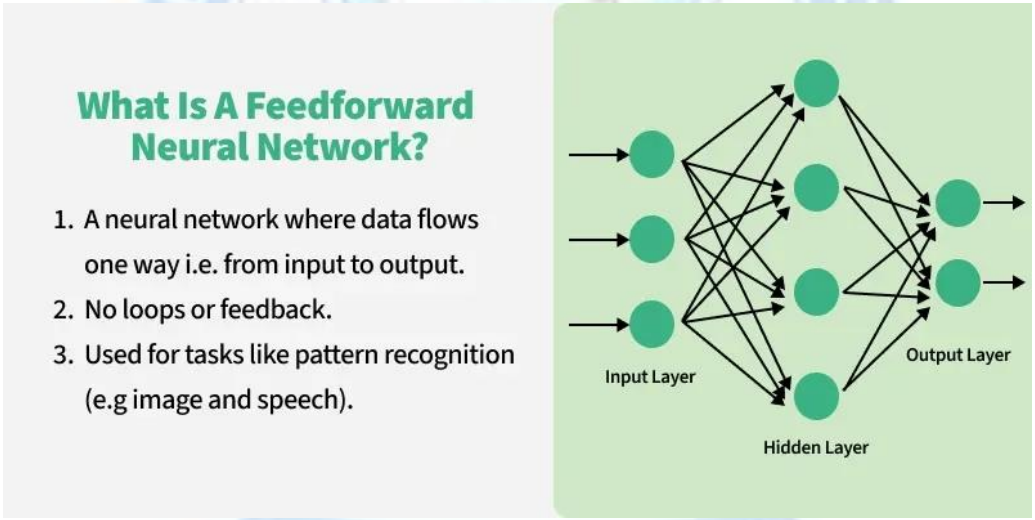
3.3 Feedforward Neural Network

Feedforward Neural Network (FNN) is a type of artificial neural network in which information flows in a single direction i.e from the input layer through hidden layers to the output layer without loops or feedback. It is mainly used for pattern recognition tasks like image and speech classification.

For example in a credit scoring system, banks use an FNN which analyze users financial profiles such as income, credit history and spending habits to determine their creditworthiness.

Each piece of information flows through the network's layers where various calculations are made to produce a final score.

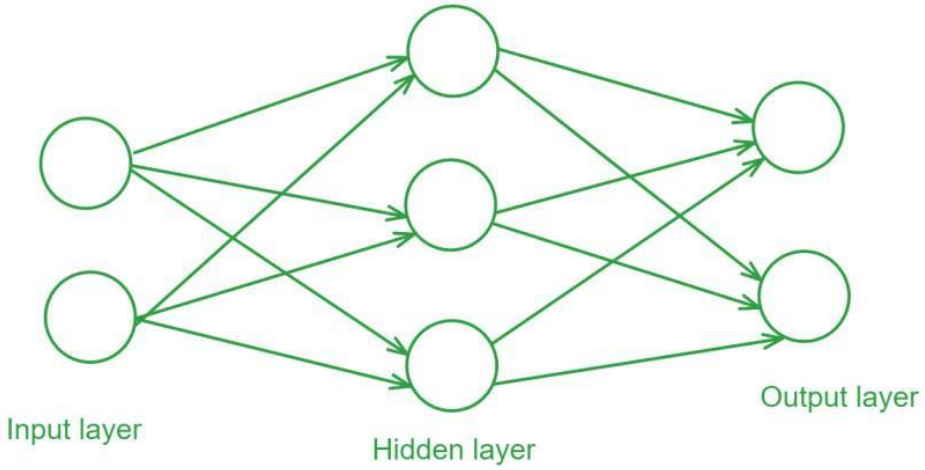


Structure of a Feedforward Neural Network

Feedforward Neural Networks have a structured layered design where data flows sequentially through each layer.

1. **Input Layer:** The input layer consists of neurons that receive the input data. Each neuron in the input layer represents a feature of the input data.
2. **Hidden Layers:** One or more hidden layers are placed between the input and output layers. These layers are responsible for learning the complex patterns in the data. Each neuron in a hidden layer applies a weighted sum of inputs followed by a non-linear activation function**.**
3. **Output Layer:** The output layer provides the final output of the network. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.

Each connection between neurons in these layers has an associated weight that is adjusted during the training process to minimize the error in predictions.



Activation Functions

Activation functions introduce non-linearity into the network enabling it to learn and model complex data patterns.

Common activation functions include:

- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Tanh**: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- **ReLU**: $\text{ReLU}(x) = \max(0, x)$

Training a Feedforward Neural Network

Training a Feedforward Neural Network involves adjusting the weights of the neurons to minimize the error between the predicted output and the actual output. This process is typically performed using backpropagation and gradient descent.

1. **Forward Propagation:** During forward propagation the input data passes through the network and the output is calculated.
2. **Loss Calculation**: The loss (or error) is calculated using a loss function such as Mean Squared Error (MSE) for regression tasks or Cross-Entropy Loss for classification **tasks.**
3. **Backpropagation:** In backpropagation the error is propagated back through the network to update the weights. The gradient of the loss function with respect to each weight is calculated and the weights are adjusted using gradient descent.

Applications of Feedforward Neural Networks

Feedforward neural networks are used in a variety of machine learning tasks including:

- Pattern recognition
- Classification tasks
- Regression analysis
- Image recognition
- Time series prediction

Despite their simplicity, feedforward neural networks can model complex relationships in data and have been the foundation for more complex neural network architectures.

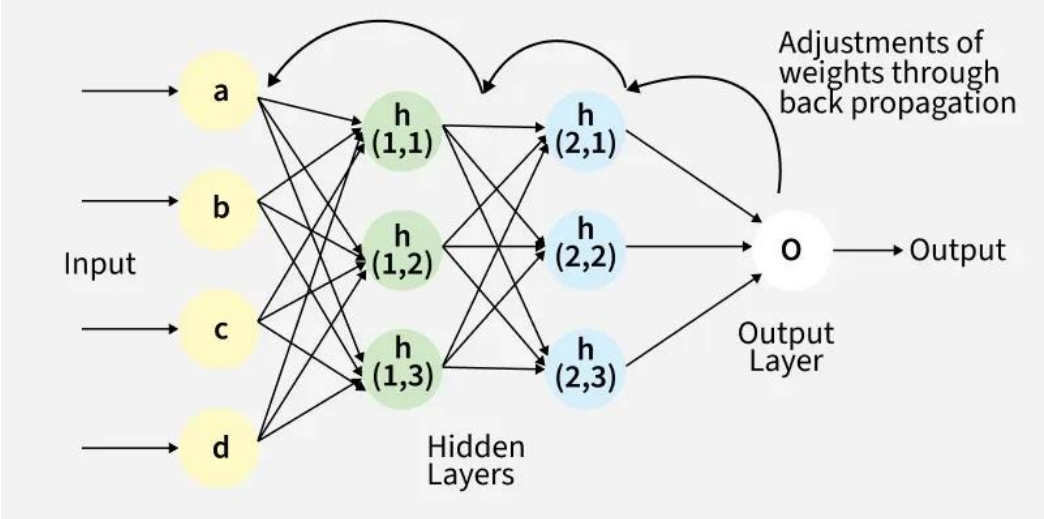
Challenges and Limitations

While feedforward neural networks are powerful, they come with their own set of challenges and limitations. One of the main challenges is the choice of the number of hidden layers and the number of neurons in each layer, which can significantly affect the performance of the network. Overfitting is another common issue where the network learns the training data too well, including the noise, and performs poorly on new, unseen data.

Backpropagation in Neural Network

Backpropagation is an algorithm that trains neural networks by reducing prediction error. It works by propagating errors backward, computing gradients using the chain rule, and updating weights and biases to improve performance.

- Computes gradients of the loss function with respect to each weight using the chain rule
- Updates weights and biases efficiently to reduce error
- Scales well to deep and complex neural networks
- Enables automatic learning and continuous improvement across training iterations



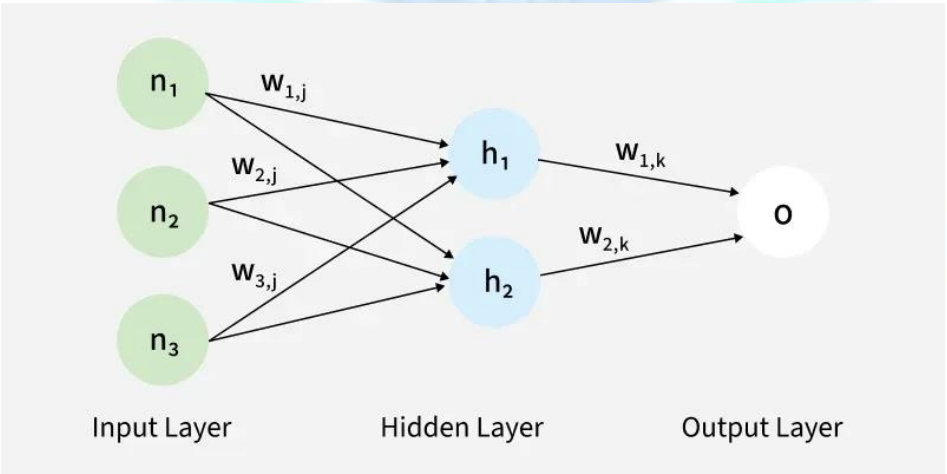
Working of Back Propagation Algorithm

The Back Propagation algorithm involves two main steps:

1. Forward Pass Work

In forward pass, input data moves through the network to generate an output.

- Input data is passed to the input layer and forwarded to hidden layers
- Each neuron computes a weighted sum and adds a bias
- Activation functions (like ReLU) are applied to produce outputs
- Outputs from one layer become inputs to the next layer
- Final layer uses functions like softmax to produce predictions (e.g., probabilities for classification)



2. Backward Pass

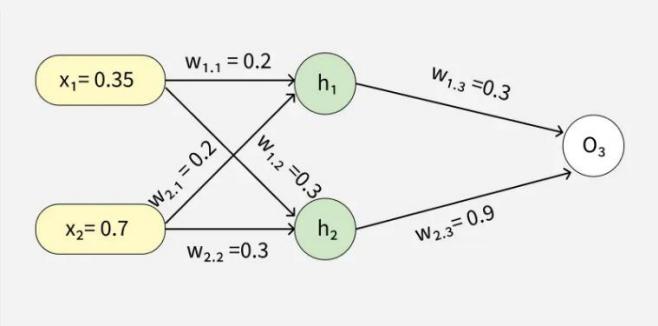
In this step, the error between predicted and actual output is propagated backward to update weights and biases. Error is calculated (e.g., using Mean Squared Error)

$$\mathbf{MSE = (Predicted\ Output - Actual\ Output)^2}$$

- Gradients are computed using the chain rule
- Weights and biases are updated to reduce the error
- Process continues layer by layer from output to input
- Derivatives of activation functions are used to compute gradients effectively

Example

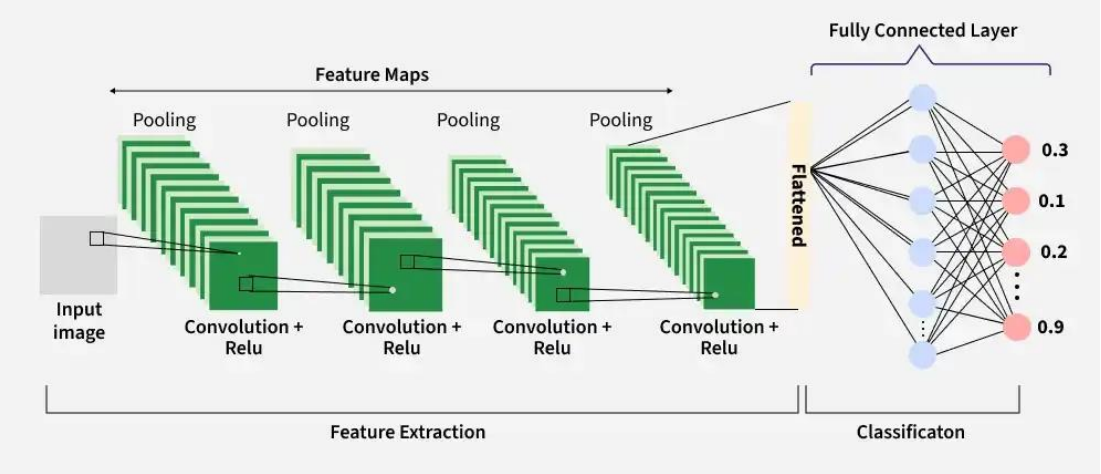
Let's walk through an example of Back Propagation in machine learning. Assume the neurons use the sigmoid activation function for the forward and backward pass. The target output is 0.5 and the learning rate is 1.



Convolution Neural Network

Convolutional Neural Networks (CNNs), are neural network architectures inspired by the human visual system, designed to process image data by capturing spatial relationships between pixels.

- Learn hierarchical features from simple edges to complex objects
- Capture spatial patterns using convolution operations
- Detect objects regardless of their position in the image
- Reduce computation by focusing on local regions instead of the entire image



Key Components of CNN

A complete Convolution Neural Networks architecture is also known as covnets. A covnets is a sequence of layers and every layer transforms one volume to another through a differentiable function. Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

1. Input Layer

The input layer receives the raw image data and passes it to the network for processing. In CNNs, input is typically a 3D volume (width × height × depth).

- Stores pixel values of the image (e.g., 32 × 32 × 3 for RGB images).
- Preserves the spatial structure of the image for further feature extraction.

2. Convolutional Layer

The Convolutional Layer is responsible for extracting important features from the input data. It applies a set of learnable filters (kernels) that slide over the image and compute the dot product between the filter weights and corresponding image patches, producing feature maps.

- Uses small filters (e.g., 2×2, 3×3, 5×5) to scan the input image.
- Generates feature maps that capture patterns such as edges, textures and shapes.

Example: Using 12 filters results in an output volume of 32 × 32 × 12.

3. Activation Layer

The Activation Layer introduces non-linearity into the network by applying an element-wise activation function to the output of the convolution layer. This enables the model to learn complex patterns beyond linear relationships.

- Common activation functions include ReLU, Tanh and Leaky ReLU.
- Applied element-wise to the feature maps.

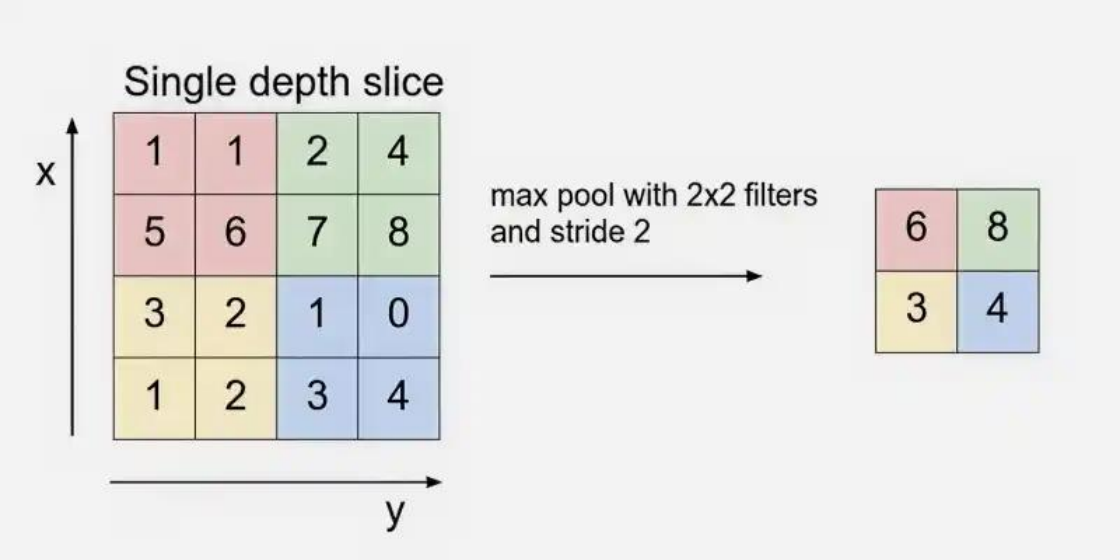
- The output dimensions remain unchanged (e.g., 32 × 32 × 12).

4. Pooling Layer

The Pooling Layer is used to reduce the spatial dimensions of the feature maps, making computation faster, reducing memory usage and helping to prevent overfitting. It is typically inserted between convolutional layers in a CNN.

- Common types include Max Pooling and Average Pooling.
- Reduces width and height while keeping depth unchanged.

Example: Using 2 × 2 max pooling with stride 2 reduces the volume from 32 × 32 × 12 to 16 × 16 × 12.



5. Flattening

Flattening converts the multi-dimensional feature maps into a one-dimensional vector after convolution and pooling. This vector is then passed to the fully connected layer for classification or regression.

Example: Flattening 16 × 16 × 12 results in a vector of size 3072 (16 × 16 × 12).

6. Fully Connected Layer

The fully connected (dense) layer performs high-level reasoning using extracted features and produces the final classification scores.

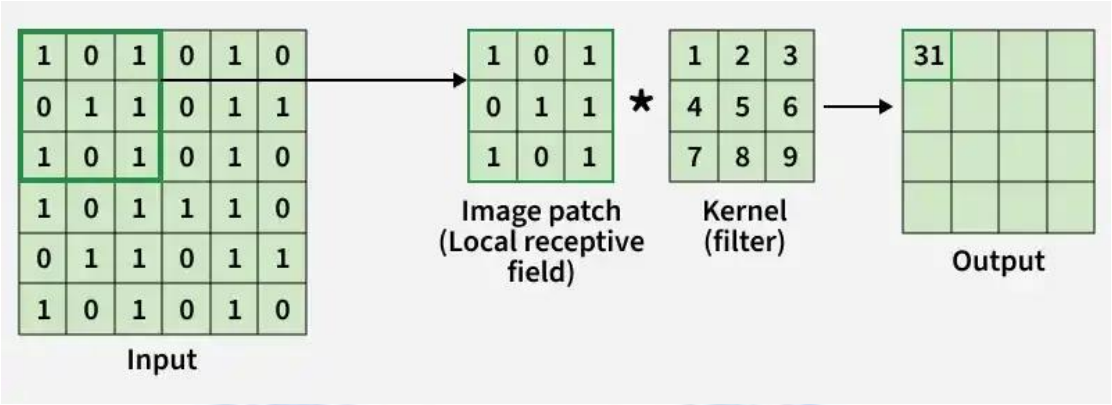
Example: The 3072-length vector is connected to neurons for classification

7. Output Layer

The output layer converts final scores into probabilities using activation functions like Sigmoid (binary classification) or Softmax (multi-class classification).

Example: For 10 classes, Softmax produces 10 probability values each representing the likelihood of a class.

How Convolutional Layers Work



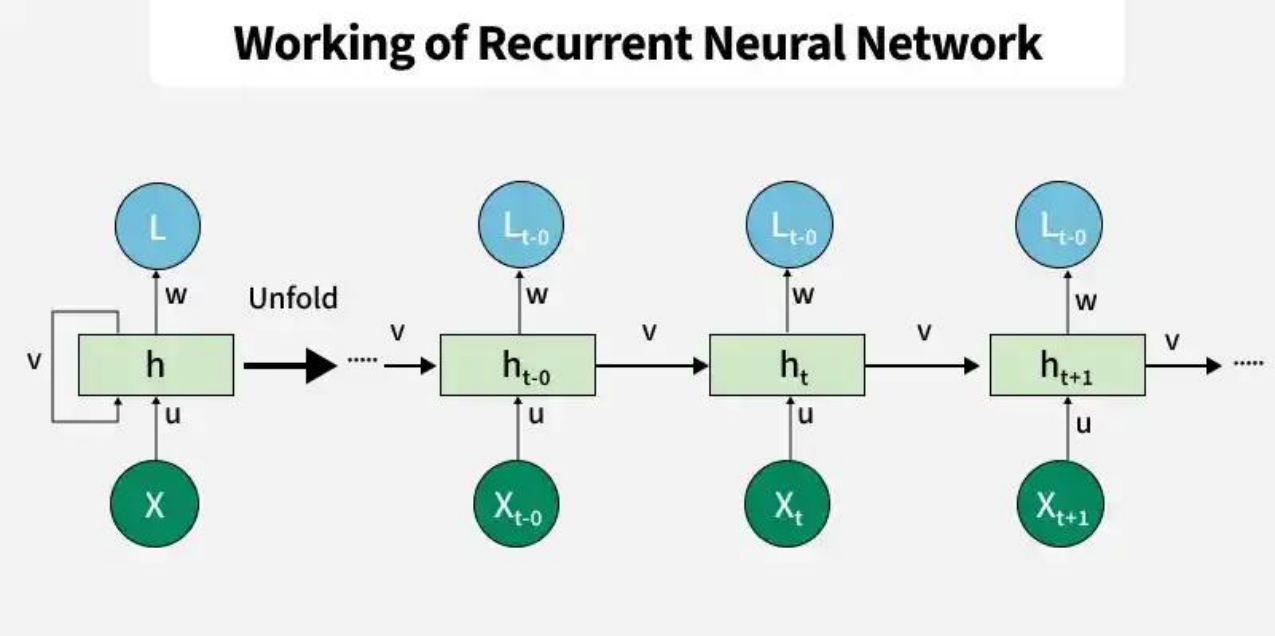
Convolution Operation

- A small matrix called a filter (kernel) slides over the input image to extract important features.
- At each position, the filter performs element-wise multiplication with the image patch.
- The multiplied values are summed together to produce a single output value.
- This operation is repeated across the entire image using a defined stride.
- The result is a new matrix called a feature map, which highlights detected patterns.
- Multiple filters are applied to capture different features such as edges, textures and shapes.
- The process preserves spatial relationships while reducing the number of learnable parameters compared to fully connected layers.
- Padding can be used to control output size and prevent loss of border information.

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



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.

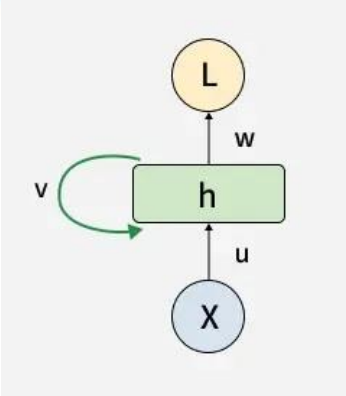
This memory of previous steps helps the network understand context and make better predictions.

Key Components of RNNs

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

1. Recurrent Neurons

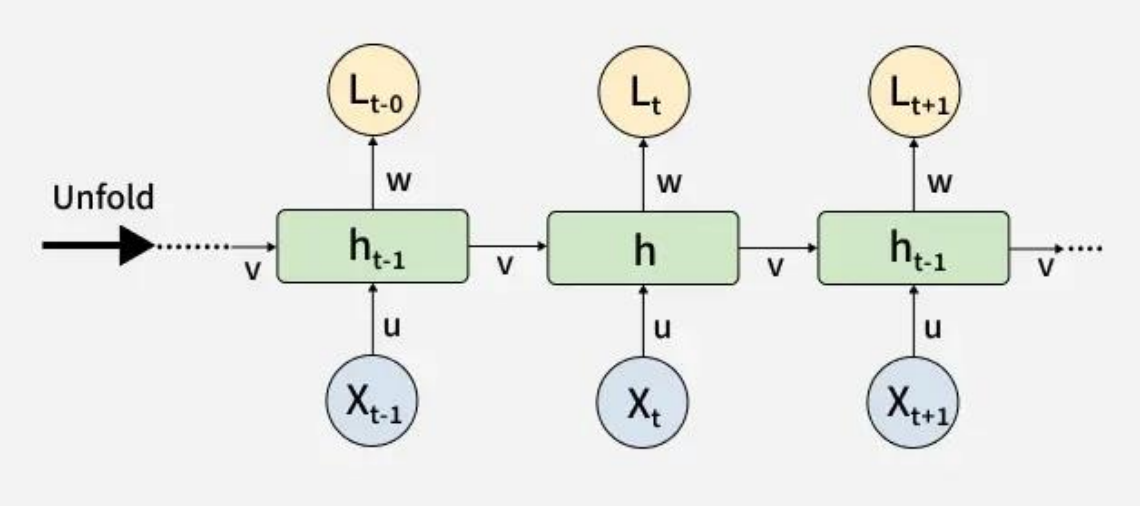
The fundamental processing unit in RNN is a Recurrent Unit**.** They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



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.
- σ represents tanh activation function
- 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:

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

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:

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

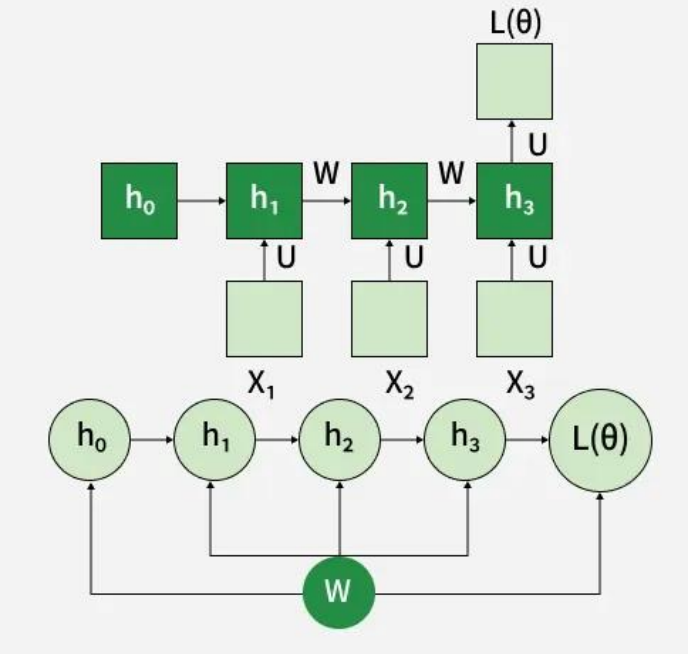
where y_t is the output and W_{hy} is the weight at the output layer.

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



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:

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

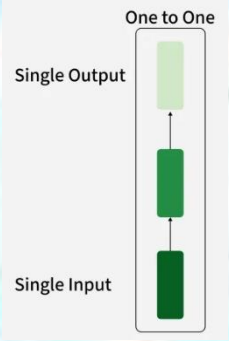
This iterative process is the essence of backpropagation through time.

Types Of Recurrent Neural Networks

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

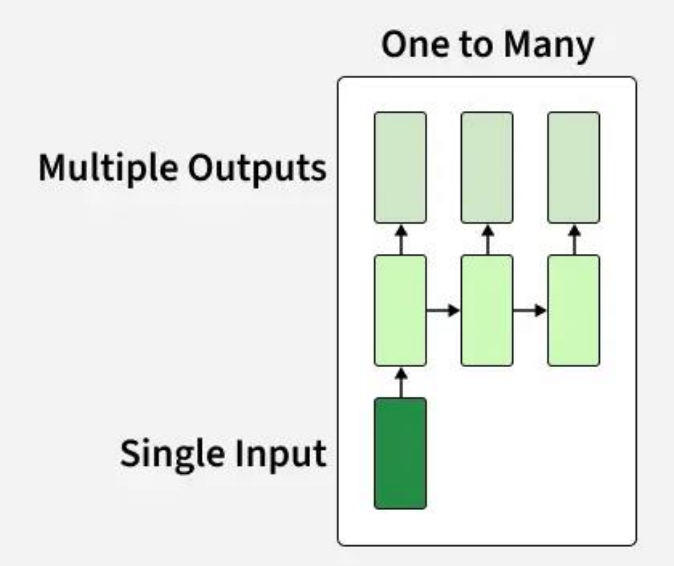
1. One-to-One RNN

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.



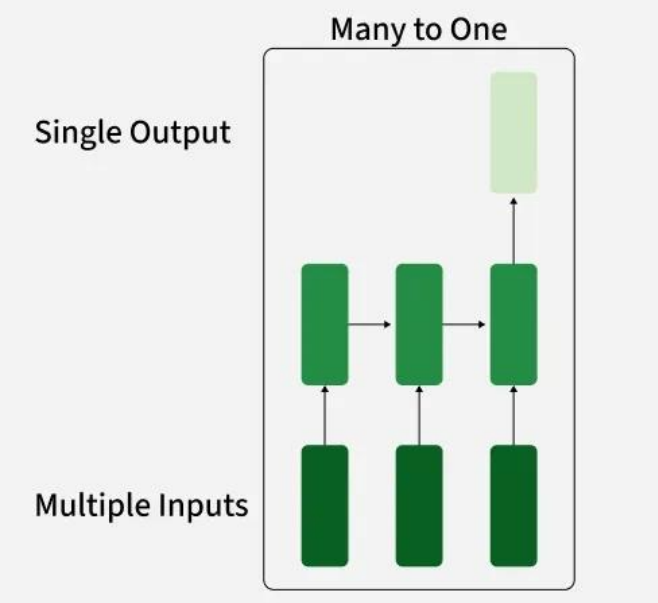
2. One-to-Many RNN

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.



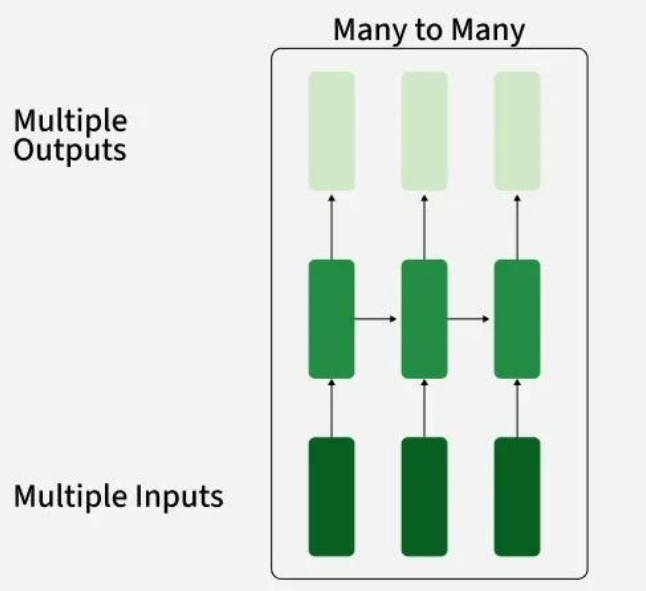
3. Many-to-One RNN

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.



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.

RNN vs Feedforward Neural Networks

Feedforward Neural Networks (FNNs)

- Process data in a single direction, from input to output
- Do not store information from previous inputs
- Suitable for tasks with independent data, such as image classification
- Perform poorly on sequential data due to the absence of memory

Recurrent Neural Networks (RNNs)

- Include feedback loops that pass information from previous steps
- Maintain memory of past inputs through hidden states
- Designed for sequential and time-dependent data
- Effective for tasks where context matters, such as text and time-series analysis

