**Recurrent Neural Network**
RNN stands for Recurrent Neural Network. It is a type of artificial neural network designed for sequential data processing. RNNs are particularly useful for tasks where the input data's order or context is crucial, such as in natural language processing (NLP), speech recognition, and time series analysis.

The key feature of RNNs is their ability to maintain a hidden state that captures information about previous inputs in the sequence. This hidden state allows RNNs to retain memory of past information and use it to influence the processing of current inputs.

Despite their ability to capture sequential dependencies, traditional RNNs have some limitations, such as difficulties in learning long-term dependencies due to issues like vanishing gradients. To address these limitations, variations of RNNs have been developed, including Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), which have mechanisms to better handle long-range dependencies.

A recurrent neural network (RNN) is a kind of artificial neural network mainly used in speech recognition and natural language processing (NLP). RNN is used in deep learning and in the development of models that imitate the activity of neurons in the human brain.

Recurrent Networks are designed to recognize patterns in sequences of data, such as text, genomes, handwriting, the spoken word, and numerical time series data emanating from sensors, stock markets, and government agencies.

A recurrent neural network looks similar to a traditional neural network except that a memory-state is added to the neurons. The computation is to include a simple memory.

The recurrent neural network is a type of deep learning-oriented algorithm, which follows a sequential approach. In neural networks, we always assume that each input and output is dependent on all other layers. These types of neural networks are called recurrent because they sequentially perform mathematical computations.

In summary, RNNs are neural network architectures designed for sequential data processing, and they have applications in various fields where understanding the temporal or sequential aspects of data is crucial.

**Application of RNN**
RNN has multiple uses when it comes to predicting the future. In the financial industry, RNN can help predict stock prices or the sign of the stock market direction (i.e., positive or negative).

RNN is used for an autonomous car as it can avoid a car accident by anticipating the route of the vehicle.

RNN is widely used in image captioning, text analysis, machine translation, and sentiment analysis. For example, one should use a movie review to understand the feeling the spectator perceived after watching the movie. Automating this task is very useful when the movie company can not have more time to review, consolidate, label, and analyse the reviews. The machine can do the job with a higher level of accuracy

**Machine Translation**
We make use of Recurrent Neural Networks in the translation engines to translate the text from one to another language. They do this with the combination of other models like LSTM (Long short-term memory)s.

**Speech Recognition**
Recurrent Neural Networks has replaced the traditional speech recognition models that made use of Hidden Markov Models. These Recurrent Neural Networks, along with LSTMs, are better poised at classifying speeches and converting them into text without loss of context.

**Sentiment Analysis**
We make use of sentiment analysis to positivity, negativity, or the neutrality of the sentence. Therefore, RNNs are most adept at handling data sequentially to find sentiments of the sentence.

**Automatic Image Tagger**
RNNs, in conjunction with convolutional neural networks, can detect the images and provide their descriptions in the form of tags. For example, a picture of a fox jumping over the fence is better explained appropriately using RNNs.

**Limitations of RNN**
RNN is supposed to carry the information in time. However, it is quite challenging to propagate all this information when the time step is too long. When a network has too many deep layers, it becomes untrainable. This problem is called: vanishing gradient problem.

If we remember, the neural network updates the weight use of the gradient descent algorithm. The gradient grows smaller when the network progress down to lower layers.

The gradient stays constant, meaning there is no space for improvement. The model learns from a change in its gradient; this change affects the network's output. If the difference in the gradient is too small (i.e., the weight change a little), the system can't learn anything and so the output. Therefore, a system facing a vanishing gradient problem cannot converge towards the right solution.

Recurrent Neural Networks (RNNs) have some limitations that can impact their performance in certain tasks. **Here are some of the key limitations:**

**Vanishing and Exploding Gradients:** Traditional RNNs can suffer from the vanishing and exploding gradient problems during training. When dealing with long sequences, the gradients can become extremely small (vanishing) or large (exploding), making it challenging for the network to learn and capture long-term dependencies.

**Difficulty in Capturing Long-Term Dependencies:** Even with techniques like LSTMs (Long Short-Term Memory) and GRUs (Gated Recurrent Unit) that address the vanishing gradient problem to some extent, RNNs may still struggle to capture long-term dependencies in sequences.

**Computationally Intensive:** Training RNNs can be computationally intensive, especially when dealing with long sequences. This can make them slow to train and may require substantial computational resources.

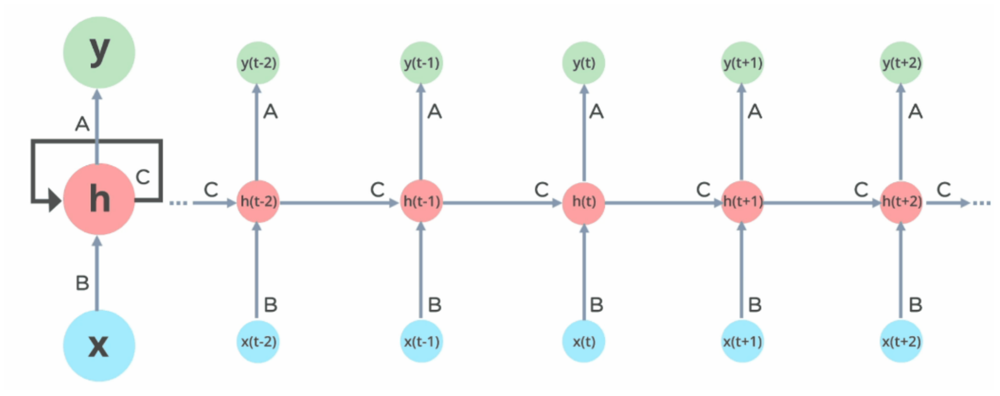
**Fixed-Length Input and Output:** RNNs typically require fixed-length input and output sequences. Handling variable-length sequences can be challenging, and padding or truncation may be necessary.

**Lack of Parallelism:** Due to their sequential nature, RNNs are inherently less parallelizable compared to other architectures. This can limit their efficiency, especially when training on hardware like GPUs, which excel at parallel processing.

**Sensitivity to Hyperparameters:** RNNs are sensitive to the choice of hyperparameters, and finding the right set of hyperparameters for a specific task can be challenging.

**Difficulty in Capturing Global Patterns**: RNNs process sequences sequentially, and sometimes they may struggle to capture global patterns in the data, especially if those patterns depend on interactions between distant elements in the sequence.

**Working mechanism of RNN**



The input layer x receives and processes the neural network’s input before passing it on to the middle layer.

Multiple hidden layers can be found in the middle layer h, each with its own activation functions, weights, and biases. You can utilize a recurrent neural network if the various parameters of different hidden layers are not impacted by the preceding layer, i.e. There is no memory in the neural network.

The different activation functions, weights, and biases will be standardized by the Recurrent Neural Network, ensuring that each hidden layer has the same characteristics. Rather than constructing numerous hidden layers, it will create only one and loop over it as many times as necessary.

## Advantages and disadvantages of RNN

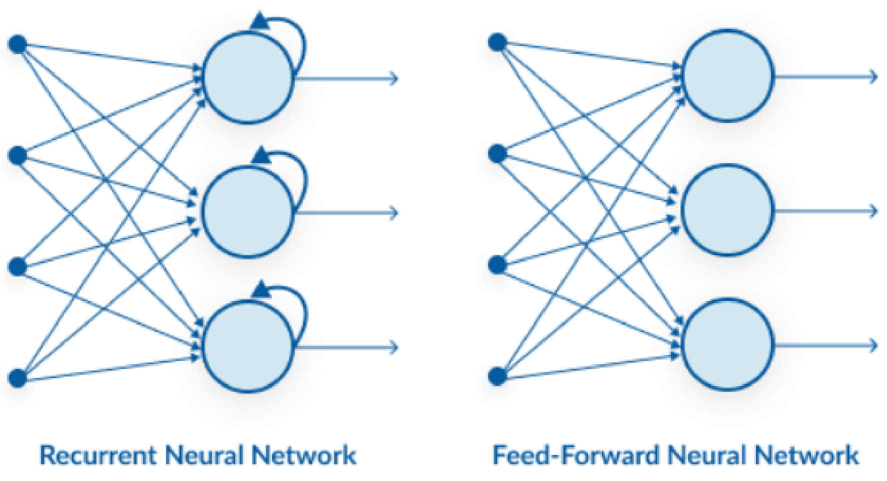
### Advantages of RNNs
Handle sequential data effectively, including text, speech, and time series.
Process inputs of any length, unlike feedforward neural networks.
Share weights across time steps, enhancing training efficiency.

### Disadvantages of RNNs:
Prone to vanishing and exploding gradient problems, hindering learning.
Training can be challenging, especially for long sequences.
Computationally slower than other neural network architectures.

## Recurrent Neural Network Vs Feedforward Neural Network
A feed-forward neural network has only one route of information flow: from the input layer to the output layer, passing through the hidden layers. The data flows across the network in a straight route, never going through the same node twice.

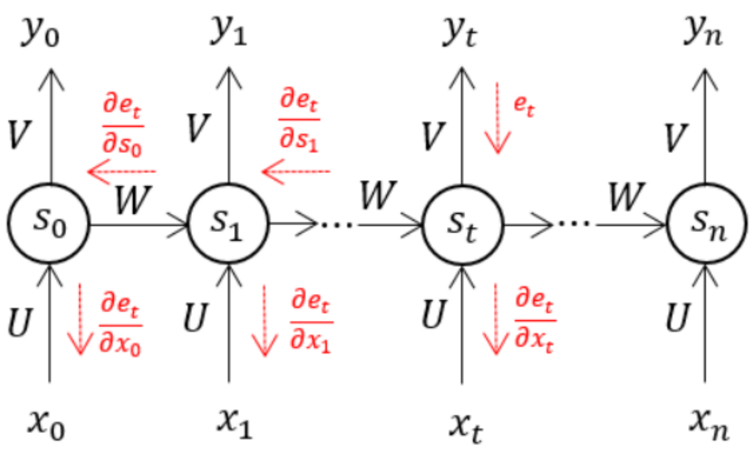
Feed-forward neural networks are poor predictions of what will happen next because they have no memory of the information they receive. Because it simply analyses the current input, a feed-forward network has no idea of temporal order. Apart from its training, it has no memory of what transpired in the past.

The information is in an RNN cycle via a loop. Before making a judgment, it evaluates the current input as well as what it has learned from past inputs. A recurrent neural network, on the other hand, may recall due to internal memory. It produces output, copies it, and then returns it to the network.



## Backpropagation Through Time (BPTT)
When we apply a Backpropagation algorithm to a Recurrent Neural Network with time series data as its input, we call it backpropagation through time.

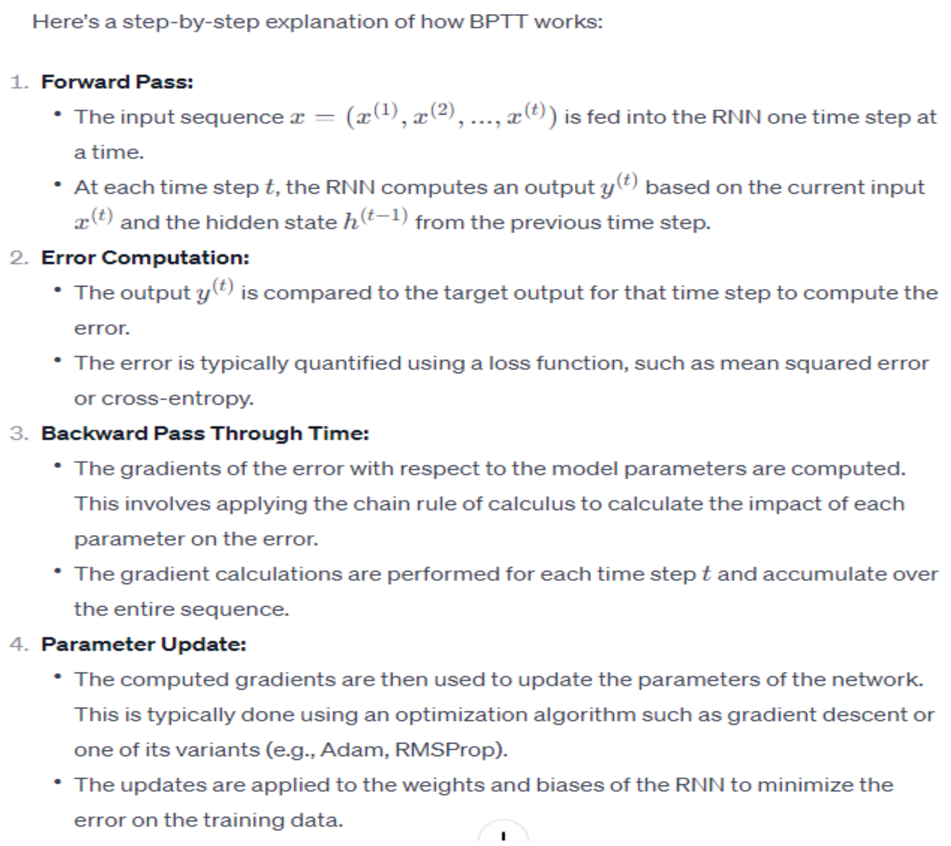
A single input is sent into the network at a time in a normal RNN, and a single output is obtained. Backpropagation, on the other hand, uses both the current and prior inputs as input. This is referred to as a timestep, and one timestep will consist of multiple time series data points entering the RNN at the same time.



$y_0$
$y_1$
$y_t$
$y_n$
$V$
$W$
$U$
$s_0$
$s_1$
$s_t$
$s_n$
$\frac{\partial e_t}{\partial s_0}$
$\frac{\partial e_t}{\partial s_1}$
$e_t$
$\frac{\partial e_t}{\partial x_0}$
$\frac{\partial e_t}{\partial x_1}$
$\frac{\partial e_t}{\partial x_t}$
$x_0$
$x_1$
$x_t$
$x_n$

The output of the neural network is used to calculate and collect the errors once it has trained on a time set and given you an output. The network is then rolled back up, and weights are recalculated and adjusted to account for the faults.

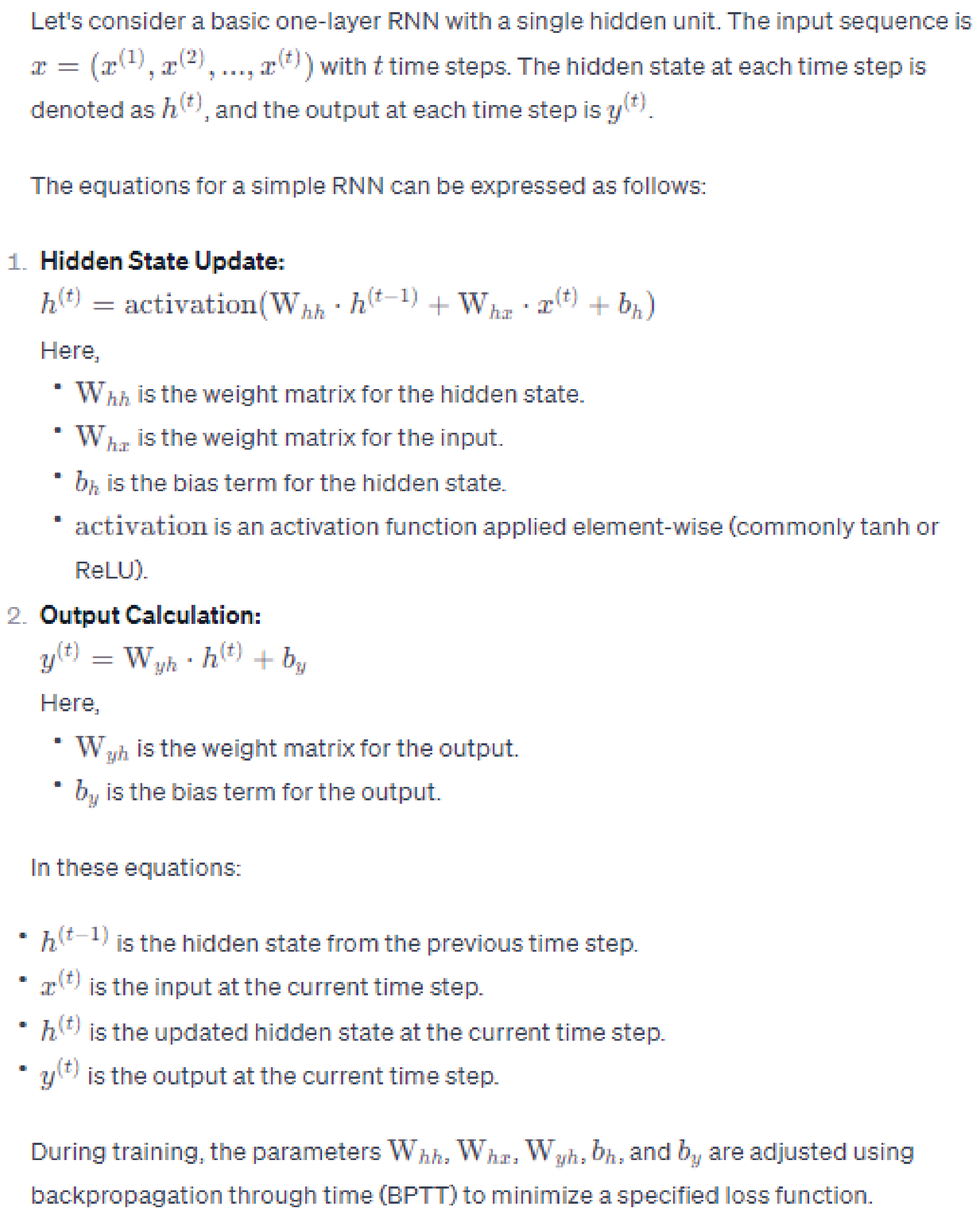
Insummary BPTT stands for Backpropagation Through Time, and it is a training algorithm used for updating the parameters of recurrent neural networks (RNNs) during the training process. RNNs, being designed for sequence data, unfold in time, and BPTT is a way to apply the backpropagation algorithm to update the weights of the network.


Here's a step-by-step explanation of how BPTT works:

1. **Forward Pass:**
   - The input sequence $x = (x^{(1)}, x^{(2)}, \ldots, x^{(t)})$ is fed into the RNN one time step at a time.
   - At each time step $t$, the RNN computes an output $y^{(t)}$ based on the current input $x^{(t)}$ and the hidden state $h^{(t-1)}$ from the previous time step.
2. **Error Computation:**
   - The output $y^{(t)}$ is compared to the target output for that time step to compute the error.
   - The error is typically quantified using a loss function, such as mean squared error or cross-entropy.
3. **Backward Pass Through Time:**
   - The gradients of the error with respect to the model parameters are computed. This involves applying the chain rule of calculus to calculate the impact of each parameter on the error.
   - The gradient calculations are performed for each time step $t$ and accumulate over the entire sequence.
4. **Parameter Update:**
   - The computed gradients are then used to update the parameters of the network. This is typically done using an optimization algorithm such as gradient descent or one of its variants (e.g., Adam, RMSProp).
   - The updates are applied to the weights and biases of the RNN to minimize the error on the training data.

The process of unfolding the RNN in time during the training process is crucial for capturing dependencies across different time steps. However, it's worth noting that BPTT has some challenges, such as the vanishing gradient problem, where gradients become very small when backpropagating through many time steps. This can make it difficult for the network to learn long-term dependencies. More advanced architectures like LSTMs and GRUs have been introduced to address these challenges in capturing long-range dependencies during training.

RNN Calculations



Let's consider a basic one-layer RNN with a single hidden unit. The input sequence is $x = (x^{(1)}, x^{(2)}, \ldots, x^{(t)})$ with $t$ time steps. The hidden state at each time step is denoted as $h^{(t)}$, and the output at each time step is $y^{(t)}$.

The equations for a simple RNN can be expressed as follows:

1. **Hidden State Update:**
   $h^{(t)} = \text{activation}(\mathrm{W}_{hh} \cdot h^{(t-1)} + \mathrm{W}_{hx} \cdot x^{(t)} + b_h)$
   Here,
   - $\mathrm{W}_{hh}$ is the weight matrix for the hidden state.
   - $\mathrm{W}_{hx}$ is the weight matrix for the input.
   - $b_h$ is the bias term for the hidden state.
   - $\text{activation}$ is an activation function applied element-wise (commonly tanh or ReLU).
2. **Output Calculation:**
   $y^{(t)} = \mathrm{W}_{yh} \cdot h^{(t)} + b_y$
   Here,
   - $\mathrm{W}_{yh}$ is the weight matrix for the output.
   - $b_y$ is the bias term for the output.

In these equations:

- $h^{(t-1)}$ is the hidden state from the previous time step.
- $x^{(t)}$ is the input at the current time step.
- $h^{(t)}$ is the updated hidden state at the current time step.
- $y^{(t)}$ is the output at the current time step.

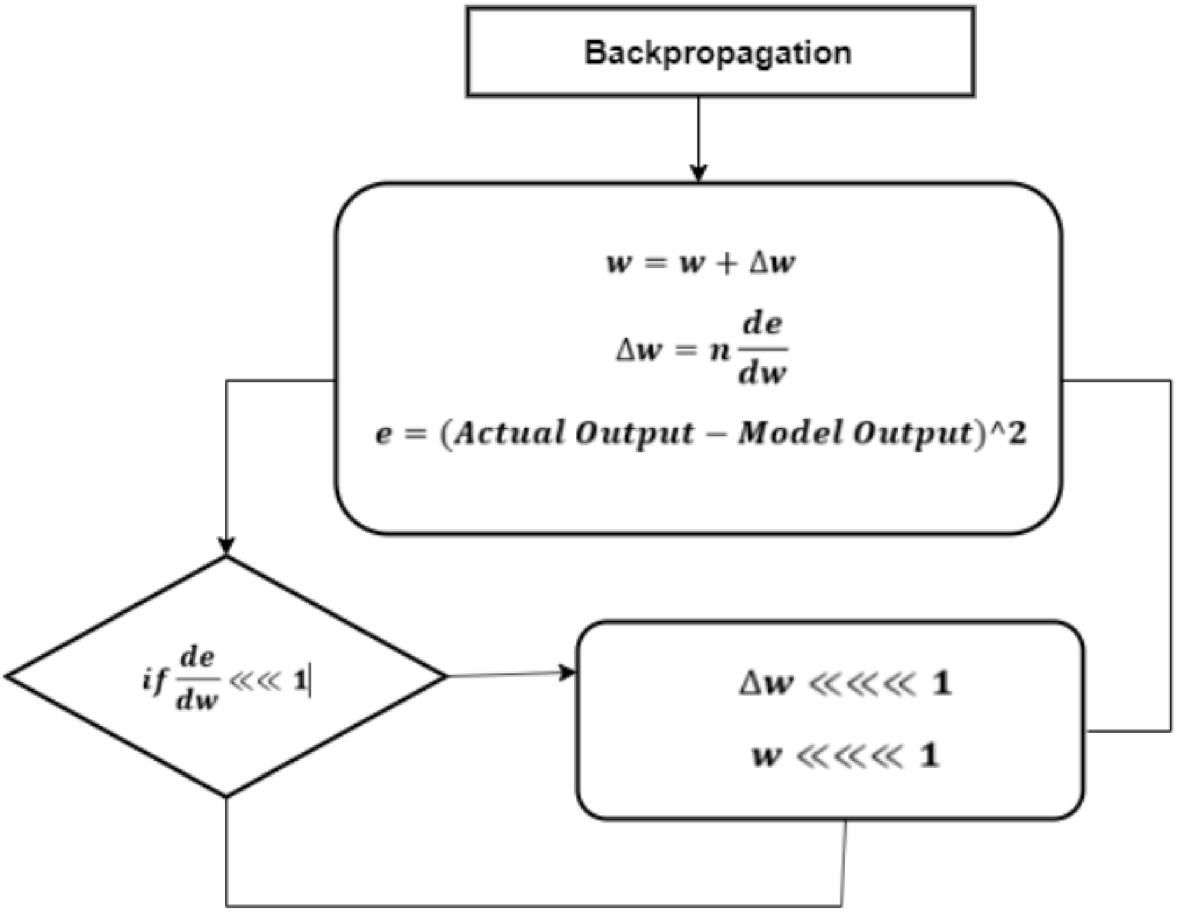
During training, the parameters $\mathrm{W}_{hh}$, $\mathrm{W}_{hx}$, $\mathrm{W}_{yh}$, $b_h$, and $b_y$ are adjusted using backpropagation through time (BPTT) to minimize a specified loss function.

**In Vanishing Gradient's** use of backpropagation, the goal is to calculate the error, which is found out by finding out the difference between the actual found out by finding out the difference between the actual output and the model output and raising that to a power of 2.

With the calculated error, the changes in the mistake concerning the difference in the weight are estimated. But with each learning rate, this can multiply with the same model.

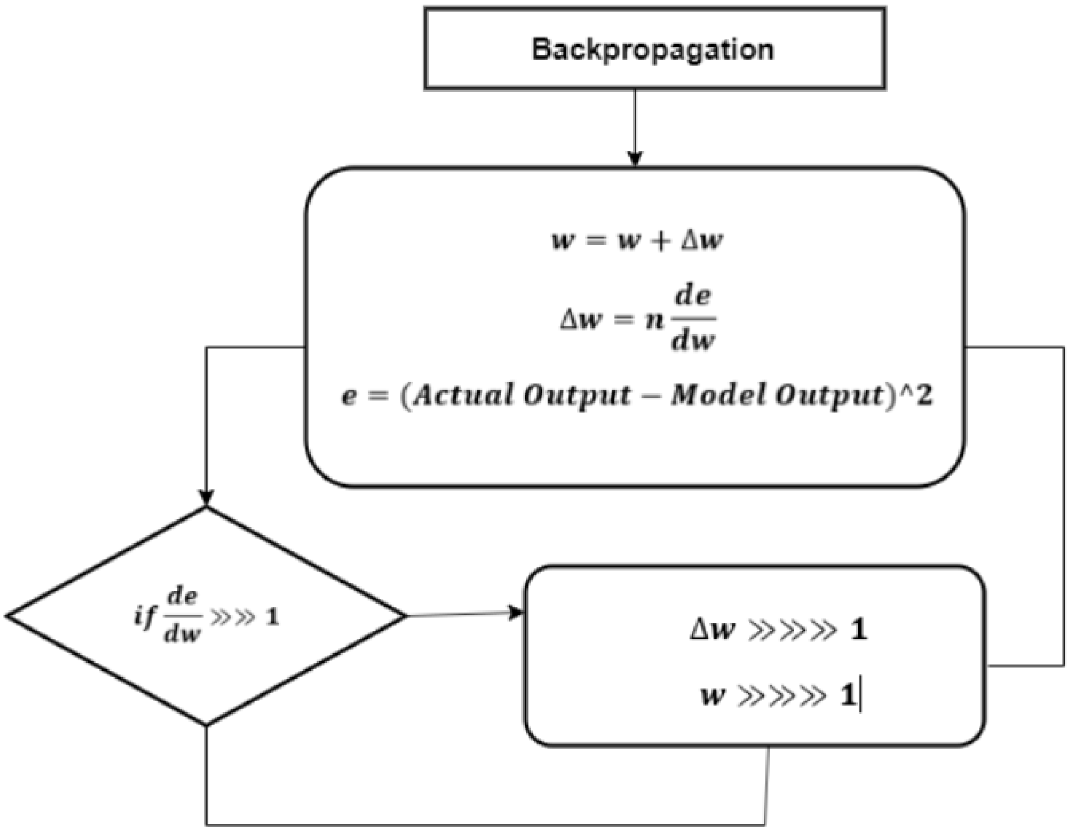
So, Product of the learning rate with the change pass to the value, which is a definite change in weight.

The change in weight is added to the old set of weights for every training iteration, as shown in the below figure. The assign here is when the change in weight is multiplied, and then the value is less.



Backpropagation
$w = w + \Delta w$
$\Delta w = n\frac{de}{dw}$
$e = (Actual\ Output - Model\ Output)^\wedge 2$
$if \frac{de}{dw} \lll 1$
$\Delta w \lll\lll 1$
$w \lll\lll 1$

**The working of the collapse gradient** is similar, but the weights here change extremely instead of negligible change. Notice the small here:



Backpropagation
$w = w + \Delta w$
$\Delta w = n \frac{de}{dw}$
$e = (Actual\ Output - Model\ Output)^\wedge 2$
$if \frac{de}{dw} \gg\gg 1$
$\Delta w \gg\gg\gg 1$
$w \gg\gg\gg 1$

| Exploding gradients | Vanishing gradients |
|---|---|
| ○ **Truncated BTT** Instead of starting backpropagation at the last timestamp, we can choose a smaller timestamp like 10 | ○ **ReLU activation function** We can use activation like ReLU, which gives output one while calculating the gradient |
| ○ **Clip gradients at the threshold** Clip the gradient when it goes more than a threshold | ○ **RMSprop** Clip the gradient when it goes higher than a threshold |
| ○ **RMSprop to adjusting the learning rate** | ○ **LSTM, GRUs** The different network architecture that has been specially designed can be used to combat this problem |

The following steps to train a recurrent neural network:

Step 1- Input a specific example from the dataset.

Step 2- The network will take an example and compute some calculations using randomly initialized variables.

Step 3- A predicted result is then computed.

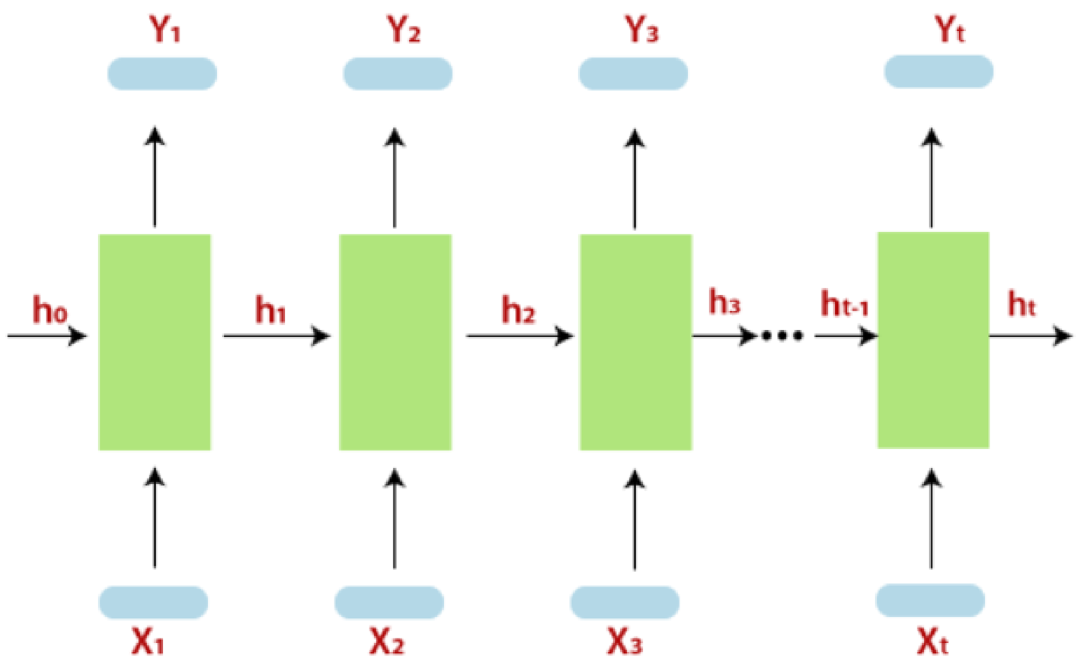
Step 4- The comparison of the actual results generated with the expected value will produce an error.

Step 5- It is propagated through the same path where the variable is also adjusted to trace the error.

Step 6- The levels from 1 to 5 are repeated until we are confident that the variables declared to get the output are appropriately defined.

Step 7- In the last step, a systematic prediction is made by applying these variables to get new unseen input.

The schematic approach of representing recurrent neural network is described below-



**CNN VS RNN**

Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are both popular types of neural network architectures, but they are designed for different types of data and tasks. Here's a comparison between CNNs and RNNs:

**Data Type:**
CNNs: Primarily used for grid-like data such as images where spatial relationships are crucial. They excel at capturing local patterns and hierarchical representations.
RNNs: Designed for sequential data where the order of elements matters, such as time series, natural language, and speech. RNNs are suitable for tasks involving temporal dependencies.
**Architecture:**
CNNs: Consist of convolutional layers followed by pooling layers. Convolutional layers use filters to scan through the input data, capturing local patterns.
RNNs: Feature recurrent connections, allowing them to maintain hidden states over time. This enables RNNs to capture sequential dependencies.
**Memory:**
CNNs: Typically lack explicit memory of past inputs. Each output is determined by the current input and the learned filters.
RNNs: Maintain hidden states that capture information about past inputs. This allows them to consider context from previous time steps.
**Use Cases:**
CNNs: Suited for tasks like image classification, object detection, and image segmentation. They are effective in tasks where local patterns contribute to the overall understanding.
RNNs: Applied in tasks such as natural language processing (e.g., language modeling, machine translation), speech recognition, and time series analysis. RNNs excel in tasks where the order of elements matters.
**Parallelization:**
CNNs: Highly parallelizable due to the nature of convolutions, making them efficient for GPU acceleration.
RNNs: Sequential nature makes them less parallelizable, potentially slowing down training on hardware like GPUs.
**Long-Term Dependencies:**
CNNs: Not inherently designed to capture long-term dependencies. They are more focused on local patterns.
RNNs: Have the ability to capture long-term dependencies in sequential data. However, traditional RNNs may struggle with vanishing gradient problems.
Architectural Variants:

CNNs: Have evolved with architectures like ResNet, Inception, and others to address challenges in capturing hierarchical features.
RNNs: Improved variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been introduced to address issues like vanishing gradients and better capture long-term dependencies.