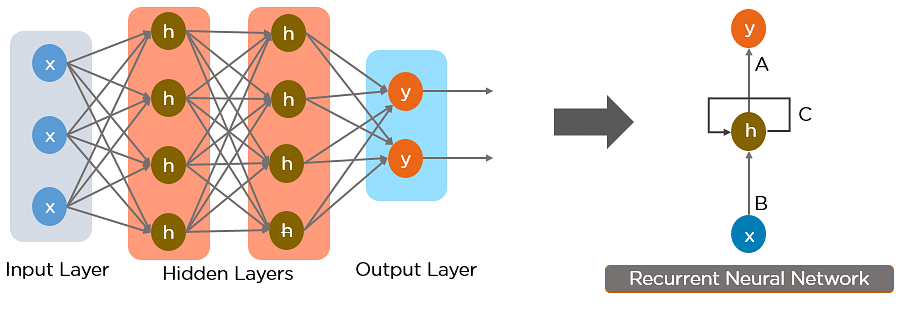
**RNN (Recurrent Neural Network)**

What Is a Recurrent Neural Network (RNN)?

RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer.

Below is how you can convert a Feed-Forward Neural Network into a Recurrent Neural Network:



Here, “x” is the input layer,

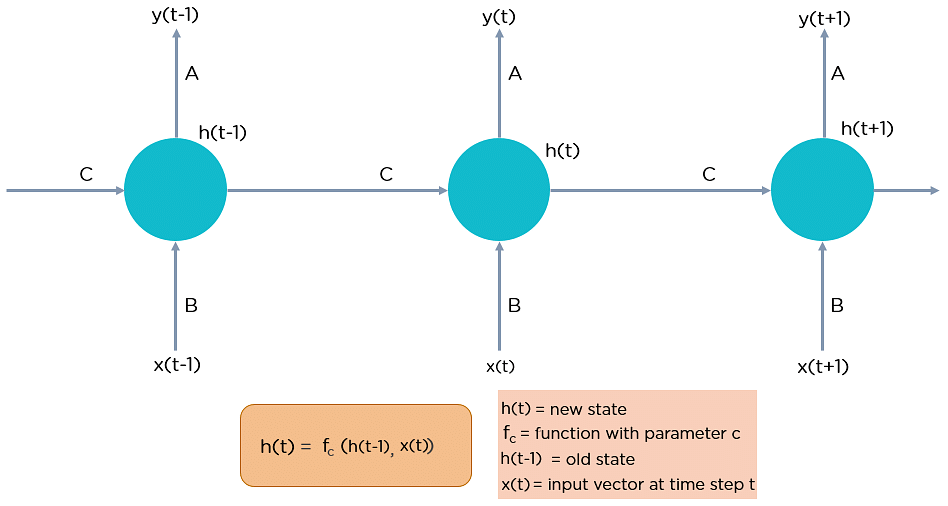
“h” is the hidden layer, and

“y” is the output layer.

A, B, and C are the network parameters used to improve the output of the model.

At any given time t, the current input is a combination of input at x(t) and x(t-1).

The output at any given time is fetched back to the network to improve on the output.



## Why Recurrent Neural Networks?

RNN were created because there were a few issues in the feed-forward neural network:

* Cannot handle sequential data
* Considers only the current input
* Cannot memorize previous inputs

## Applications of Recurrent Neural Networks

Natural Language Processing

Text mining and Sentiment analysis can be carried out using an RNN for Natural Language Processing (NLP).

Time Series Prediction

Any time series problem, like predicting the prices of stocks in a particular month, can be solved using an RNN.

Image Captioning

RNNs are used to caption an image by analyzing the activities present.

**Built-in RNN layers:** There are three built-in RNN layers in Keras:

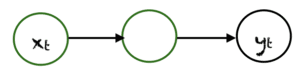
1. keras.layers.SimpleRNN, a fully-connected RNN where the output from previous timestep is to be fed to next timestep.
2. keras.layers.GRU,
3. keras.layers.LSTM

## Types of Recurrent Neural Networks

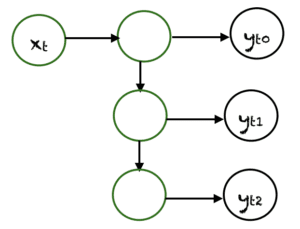
There are four types of Recurrent Neural Networks:

1. One to One
2. One to Many
3. Many to One
4. Many to Many

### One to One

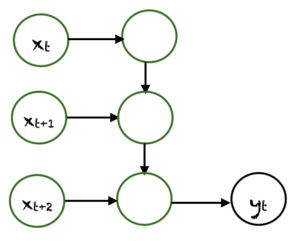
[](https://machinelearningmastery.com/wp-content/uploads/2021/09/rnn2.png)

### One to Many

[](https://machinelearningmastery.com/wp-content/uploads/2021/09/rnn3.png)

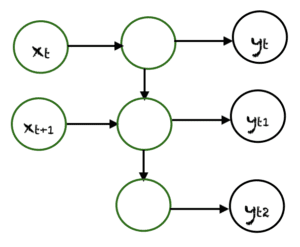
In one-to-many networks, a single input can produce multiple outputs,

### Many to One

[](https://machinelearningmastery.com/wp-content/uploads/2021/09/rnn4.png)

In this case, many inputs from different time steps produce a single output.

### Many to Many

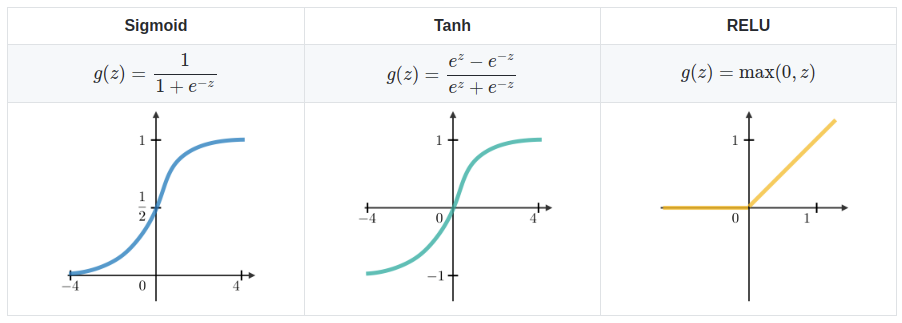
[](https://machinelearningmastery.com/wp-content/uploads/2021/09/rnn5.png)

There are many possibilities for many-to-many. An example is shown above, where two inputs produce three outputs. Many-to-many networks are applied in machine translation (language translation).

## The Activation Function

We can use any activation function we like in the recurrent neural network. Common choices are:

* Sigmoid function
* Tanh function
* Relu function



**RNNs have various advantages,**

* Ability to handle sequence data
* Ability to handle inputs of varying lengths
* Ability to store or “memorize” historical information

**The disadvantages are:**

* The computation can be very slow.
* The network does not take into account future inputs to make decisions.
* Vanishing gradient problem, where the gradients used to compute the weight update may get very close to zero, preventing the network from learning new weights. The deeper the network, the more pronounced this problem is.

**Basic Python Implementation (RNN with Keras)**

**Import the required libraries**

|  |
| --- |
| import numpy as np  import tensorflow as tf  from tensorflow import keras  from tensorflow.keras import layers |

Here’s a simple Sequential model that processes integer sequences, embeds each integer into a 64-dimensional vector, and then uses an LSTM layer to handle the sequence of vectors.

|  |
| --- |
| model = keras.Sequential()  model.add(layers.Embedding(input\_dim=1000, output\_dim=64))  model.add(layers.LSTM(128))  model.add(layers.Dense(10))  model.summary() |

**Output:**

|  |
| --- |
| Model: "sequential"  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Layer (type) Output Shape Param #  =================================================================  embedding (Embedding) (None, None, 64) 64000  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  lstm (LSTM) (None, 128) 98816  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense (Dense) (None, 10) 1290  =================================================================  Total params: 164,106  Trainable params: 164,106  Non-trainable params: 0 |