What Is a Neural Network?

A [Neural Network](https://www.simplilearn.com/tutorials/deep-learning-tutorial/neural-network) consists of different layers connected to each other, working on the structure and function of a human brain. It learns from huge volumes of data and uses complex algorithms to train a neural net.

Here is an example of how neural networks can identify a dog’s breed based on their features.

* The image pixels of two different breeds of dogs are fed to the input layer of the neural network.
* The image pixels are then processed in the hidden layers for feature extraction.
* The output layer produces the result to identify if it’s a German Shepherd or a Labrador.
* Such networks do not require memorizing the past output.

Several neural networks can help solve different business problems. Let’s look at a few of them.

* Feed-Forward Neural Network: Used for general[Regression and Classification](https://www.simplilearn.com/regression-vs-classification-in-machine-learning-article) problems.
* Convolutional Neural Network: Used for object detection and image classification.
* Deep Belief Network: Used in healthcare sectors for cancer detection.
* RNN: Used for speech recognition, voice recognition, time series prediction, and [natural language processing.](https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/what-is-natural-language-processing-nlp)

What Are Recurrent Neural Networks (RNN)?

Recurrent Neural Networks (RNNs) are a type of artificial neural network designed to process sequences of data. They work especially well for jobs requiring sequences, such as time series data, voice, natural language, and other activities.

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:

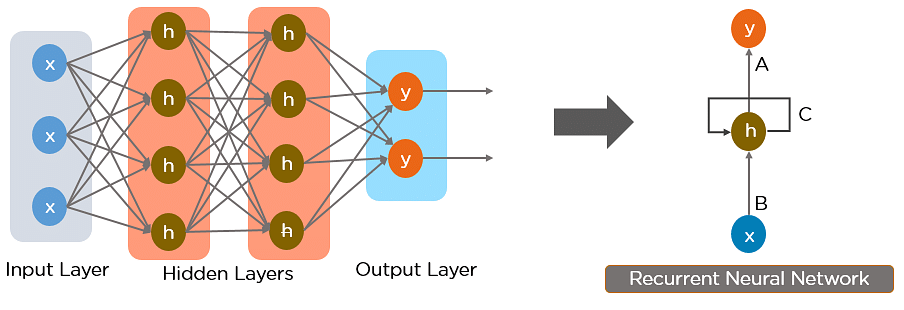


Fig: Simple Recurrent Neural Network

The nodes in different layers of the neural network are compressed to form a single layer of recurrent neural networks. A, B, and C are the parameters of the network.

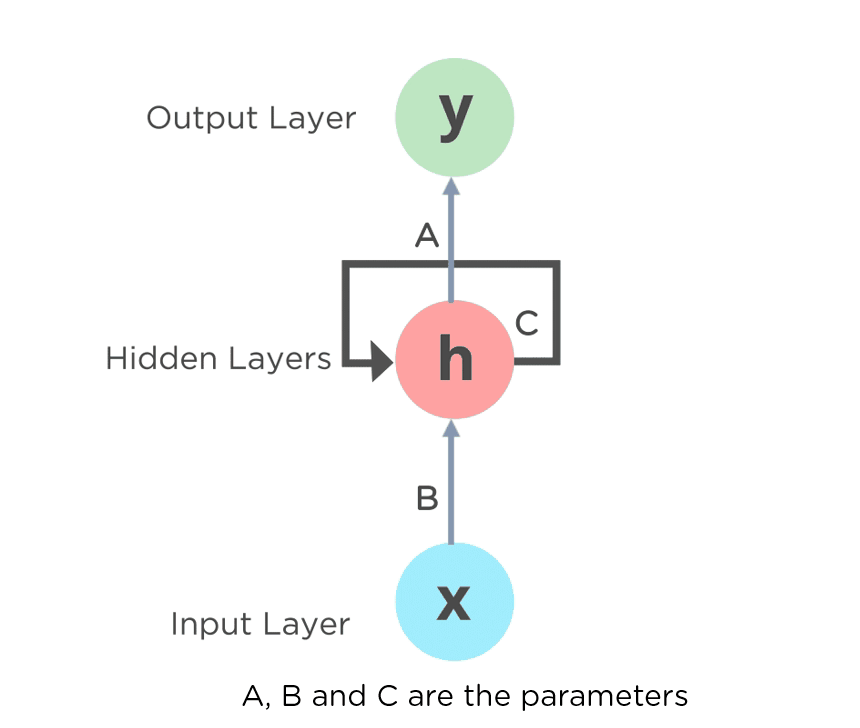


   Fig: Fully connected Recurrent Neural Network

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.

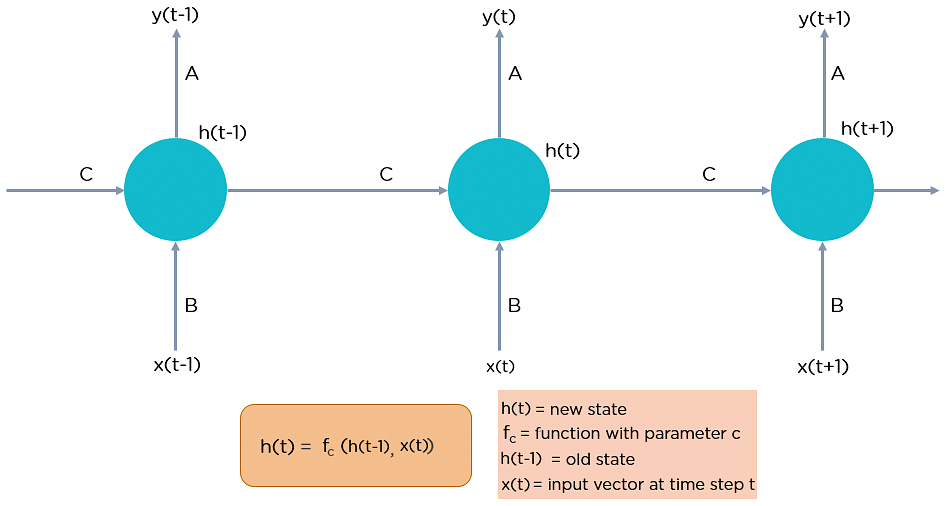


Fig: Fully connected Recurrent Neural Network

**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

The solution to these issues is the RNN. An RNN can handle sequential data, accepting the current input data, and previously received inputs. RNNs can memorize previous inputs due to their internal memory.

**How Does Recurrent Neural Networks Work?**

In Recurrent Neural networks, the information cycles through a loop to the middle hidden layer.

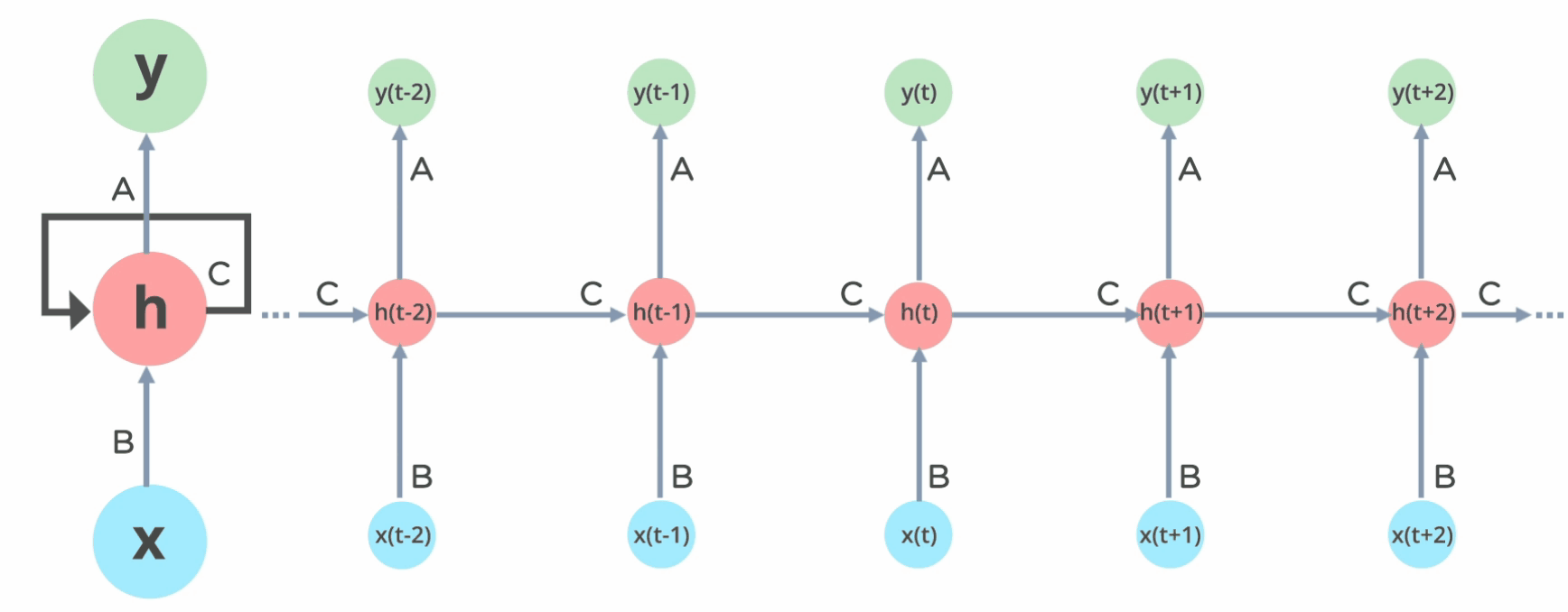


Fig: Working of Recurrent Neural Network

The input layer ‘x’ takes in the input to the neural network and processes it and passes it onto the middle layer.

The middle layer ‘h’ can consist of multiple hidden layers, each with its own activation functions and weights and biases. If you have a neural network where the various parameters of different hidden layers are not affected by the previous layer, ie: the neural network does not have memory, then you can use a recurrent neural network.

The Recurrent Neural Network will standardize the different activation functions and weights and biases so that each hidden layer has the same parameters. Then, instead of creating multiple hidden layers, it will create one and loop over it as many times as required.

**Feed-Forward Neural Networks vs Recurrent Neural Networks**

A feed-forward neural network allows information to flow only in the forward direction, from the input nodes, through the hidden layers, and to the output nodes. There are no cycles or loops in the network.

Below is how a simplified presentation of a feed-forward neural network looks like:

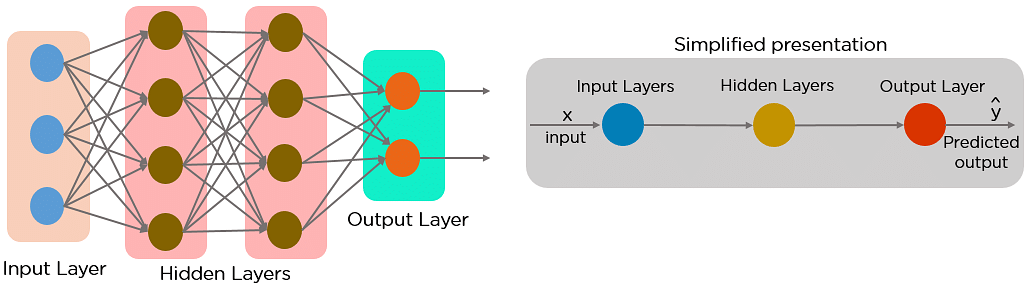


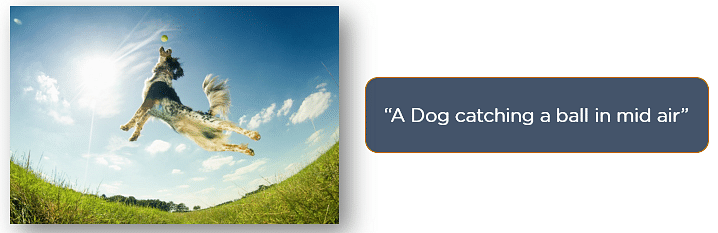
Fig: Feed-forward Neural Network

In a feed-forward neural network, the decisions are based on the current input. It doesn’t memorize the past data, and there’s no future scope. Feed-forward neural networks are used in general regression and classification problems.

**Applications of Recurrent Neural Networks**

Image Captioning

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



[Time Series Prediction](https://www.simplilearn.com/tutorials/statistics-tutorial/what-is-time-series-analysis)

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

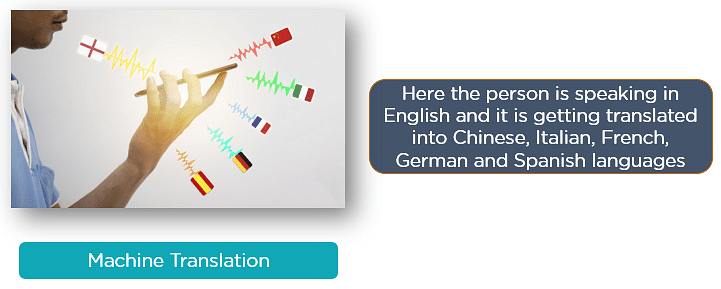
[Natural Language Processing](https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/what-is-natural-language-processing-nlp)

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



Machine Translation

Given an input in one language, RNNs can be used to translate the input into different languages as output.



**Advantages of Recurrent Neural Network**

Recurrent Neural Networks (RNNs) have several advantages over other types of neural networks, including:

Ability To Handle Variable-Length Sequences

RNNs are designed to handle input sequences of variable length, which makes them well-suited for tasks such as speech recognition, natural language processing, and time series analysis.

Memory Of Past Inputs

RNNs have a memory of past inputs, which allows them to capture information about the context of the input sequence. This makes them useful for tasks such as language modeling, where the meaning of a word depends on the context in which it appears.

Parameter Sharing

RNNs share the same set of parameters across all time steps, which reduces the number of parameters that need to be learned and can lead to better generalization.

Non-Linear Mapping

RNNs use non-linear activation functions, which allows them to learn complex, non-linear mappings between inputs and outputs.

Sequential Processing

RNNs process input sequences sequentially, which makes them computationally efficient and easy to parallelize.

Flexibility

RNNs can be adapted to a wide range of tasks and input types, including text, speech, and image sequences.

Improved Accuracy

RNNs have been shown to achieve state-of-the-art performance on a variety of sequence modeling tasks, including language modeling, speech recognition, and machine translation.

These advantages make RNNs a powerful tool for sequence modeling and analysis, and have led to their widespread use in a variety of applications, including natural language processing, speech recognition, and time series analysis.

**Disadvantages of Recurrent Neural Network**

Although Recurrent Neural Networks (RNNs) have several advantages, they also have some disadvantages. Here are some of the main disadvantages of RNNs:

Vanishing And Exploding Gradients

RNNs can suffer from the problem of vanishing or exploding gradients, which can make it difficult to train the network effectively. This occurs when the gradients of the loss function with respect to the parameters become very small or very large as they propagate through time.

Computational Complexity

RNNs can be computationally expensive to train, especially when dealing with long sequences. This is because the network has to process each input in sequence, which can be slow.

Difficulty In Capturing Long-Term Dependencies

Although RNNs are designed to capture information about past inputs, they can struggle to capture long-term dependencies in the input sequence. This is because the gradients can become very small as they propagate through time, which can cause the network to forget important information.

Lack Of Parallelism

RNNs are inherently sequential, which makes it difficult to parallelize the computation. This can limit the speed and scalability of the network.

Difficulty In Choosing The Right Architecture

There are many different variants of RNNs, each with its own advantages and disadvantages. Choosing the right architecture for a given task can be challenging, and may require extensive experimentation and tuning.

Difficulty In Interpreting The Output

The output of an RNN can be difficult to interpret, especially when dealing with complex inputs such as natural language or audio. This can make it difficult to understand how the network is making its predictions.

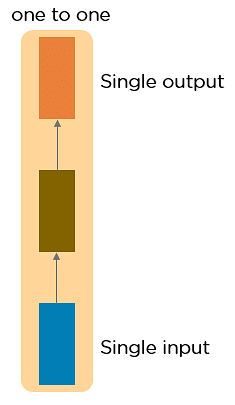
**Types of Recurrent Neural Networks**

There are four types of Recurrent Neural Networks:

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

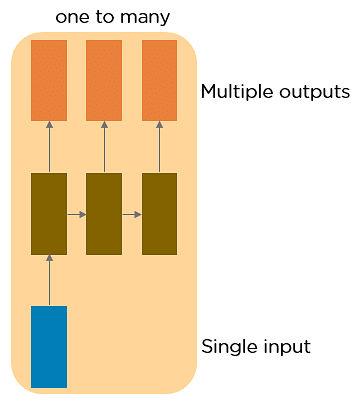
One to One RNN

This type of neural network is known as the Vanilla Neural Network. It's used for general machine learning problems, which has a single input and a single output.



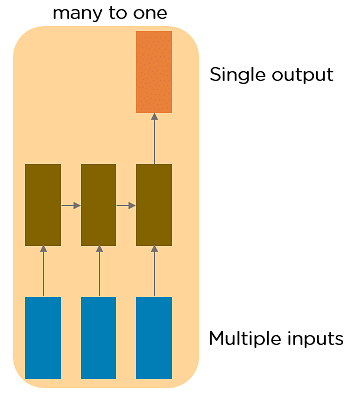
One to Many RNN

This type of neural network has a single input and multiple outputs. An example of this is the image caption.



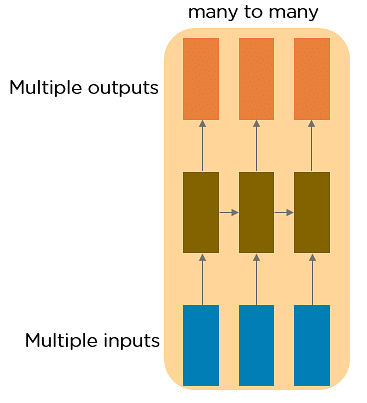
Many to One RNN

This RNN takes a sequence of inputs and generates a single output. Sentiment analysis is a good example of this kind of network where a given sentence can be classified as expressing positive or negative sentiments.



Many to Many RNN

This RNN takes a sequence of inputs and generates a sequence of outputs. Machine translation is one of the examples.

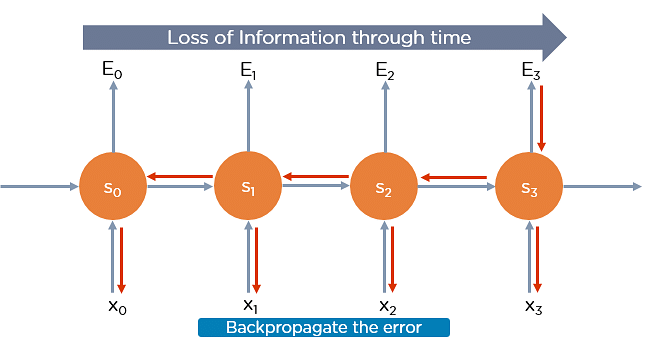


Two Issues of Standard RNNs

1. Vanishing Gradient Problem

Recurrent Neural Networks enable you to model time-dependent and sequential data problems, such as stock market prediction, machine translation, and text generation. You will find, however, RNN is hard to train because of the gradient problem.

RNNs suffer from the problem of vanishing gradients. The gradients carry information used in the RNN, and when the gradient becomes too small, the parameter updates become insignificant. This makes the learning of long data sequences difficult.

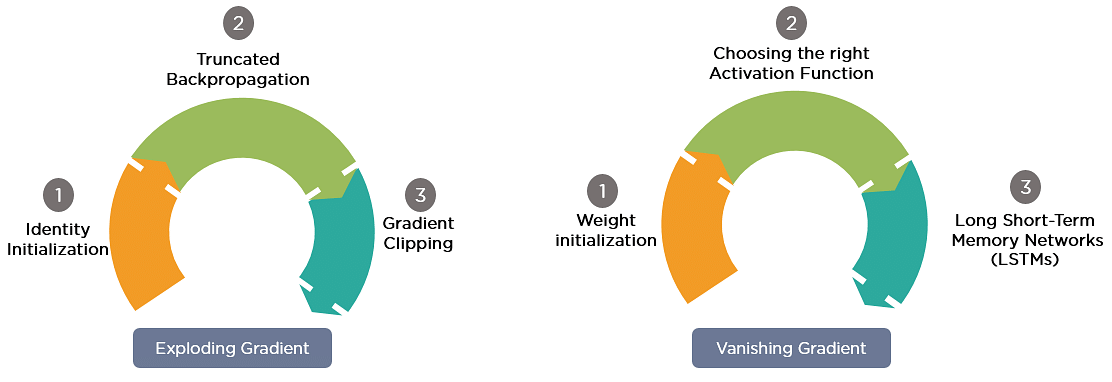


2. Exploding Gradient Problem

While training a neural network, if the slope tends to grow exponentially instead of decaying, this is called an Exploding Gradient. This problem arises when large error gradients accumulate, resulting in very large updates to the neural network model weights during the training process.

Long training time, poor performance, and bad accuracy are the major issues in gradient problems.

**Gradient Problem Solutions**



Now, let’s discuss the most popular and efficient way to deal with gradient problems,

**Long Short-Term Memory Network (LSTMs).**

First, let’s understand Long-Term Dependencies.

Suppose you want to predict the last word in the text: “The clouds are in the \_\_\_\_\_\_.”

The most obvious answer to this is the “sky.” We do not need any further context to predict the last word in the above sentence.

Consider this sentence: “I have been staying in Spain for the last 10 years…I can speak fluent \_\_\_\_\_\_.”

The word you predict will depend on the previous few words in context. Here, you need the context of Spain to predict the last word in the text, and the most suitable answer to this sentence is “Spanish.” The gap between the relevant information and the point where it's needed may have become very large. LSTMs help you solve this problem.

**Common Activation Functions**

Recurrent Neural Networks (RNNs) use activation functions just like other neural networks to introduce non-linearity to their models. Here are some common activation functions used in RNNs:

Sigmoid Function:

The sigmoid function is commonly used in RNNs. It has a range between 0 and 1, which makes it useful for binary classification tasks. The formula for the sigmoid function is:

σ(x) = 1 / (1 + e^(-x))

Hyperbolic Tangent (Tanh) Function:

The tanh function is also commonly used in RNNs. It has a range between -1 and 1, which makes it useful for non-linear classification tasks. The formula for the tanh function is:

tanh(x) = (e^x - e^(-x)) / (e^x + e^(-x))

Rectified Linear Unit (Relu) Function:

The ReLU function is a non-linear activation function that is widely used in deep neural networks. It has a range between 0 and infinity, which makes it useful for models that require positive outputs. The formula for the ReLU function is:

ReLU(x) = max(0, x)

Leaky Relu Function:

The Leaky ReLU function is similar to the ReLU function, but it introduces a small slope to negative values, which helps to prevent "dead neurons" in the model. The formula for the Leaky ReLU function is:

Leaky ReLU(x) = max(0.01x, x)

Softmax Function:

The softmax function is often used in the output layer of RNNs for multi-class classification tasks. It converts the network output into a probability distribution over the possible classes. The formula for the softmax function is:

softmax(x) = e^x / ∑(e^x)

These are just a few examples of the activation functions used in RNNs. The choice of activation function depends on the specific task and the model's architecture.

**Backpropagation Through Time**

Backpropagation through time is when we apply a Backpropagation algorithm to a Recurrent Neural network that has time series data as its input.

In a typical RNN, one input is fed into the network at a time, and a single output is obtained. But in backpropagation, you use the current as well as the previous inputs as input. This is called **a timestep** and one timestep will consist of many time series data points entering the RNN simultaneously.

Once the neural network has trained on a timeset and given you an output, that output is used to calculate and accumulate the errors. After this, the network is rolled back up and weights are recalculated and updated keeping the errors in mind.

**Variant RNN Architectures**

There are several variant RNN architectures that have been developed over the years to address the limitations of the standard RNN architecture. Here are a few examples:

**Long Short-Term Memory (LSTM) Networks**

LSTM is a type of RNN that is designed to handle the vanishing gradient problem that can occur in standard RNNs. It does this by introducing three gating mechanisms that control the flow of information through the network: the input gate, the forget gate, and the output gate. These gates allow the LSTM network to selectively remember or forget information from the input sequence, which makes it more effective for long-term dependencies.

**Gated Recurrent Unit (GRU) Networks**

GRU is another type of RNN that is designed to address the vanishing gradient problem. It has two gates: **the reset gate and the update gate**. The reset gate determines how much of the previous state should be forgotten, while the update gate determines how much of the new state should be remembered. This allows the GRU network to selectively update its internal state based on the input sequence.

Bidirectional RNNs:

Bidirectional RNNs are designed to process input sequences in both forward and backward directions. This allows the network to capture both past and future context, which can be useful for speech recognition and natural language processing tasks.

Encoder-Decoder RNNs:

Encoder-decoder RNNs consist of two RNNs: an encoder network that processes the input sequence and produces a fixed-length vector representation of the input and a decoder network that generates the output sequence based on the encoder's representation. This architecture is commonly used for sequence-to-sequence tasks such as machine translation.

.

Long Short-Term Memory Networks

LSTMs are a special kind of RNN — capable of learning long-term dependencies by remembering information for long periods is the default behavior.

All RNN are in the form of a chain of repeating modules of a neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

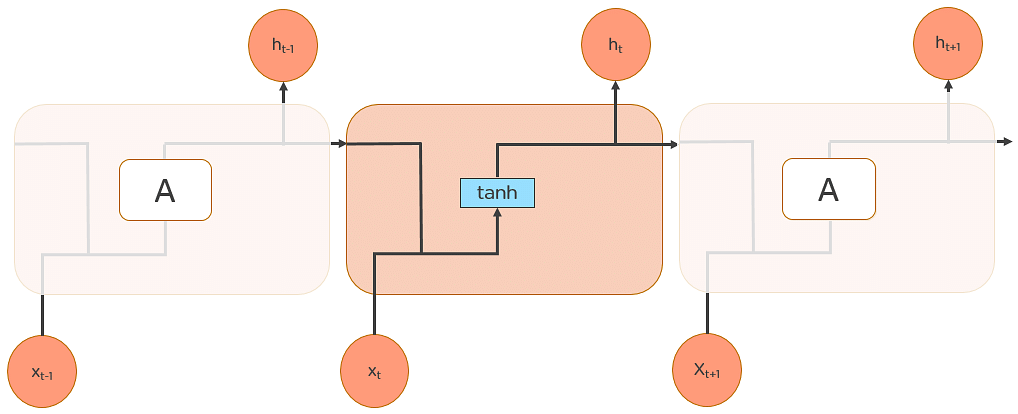
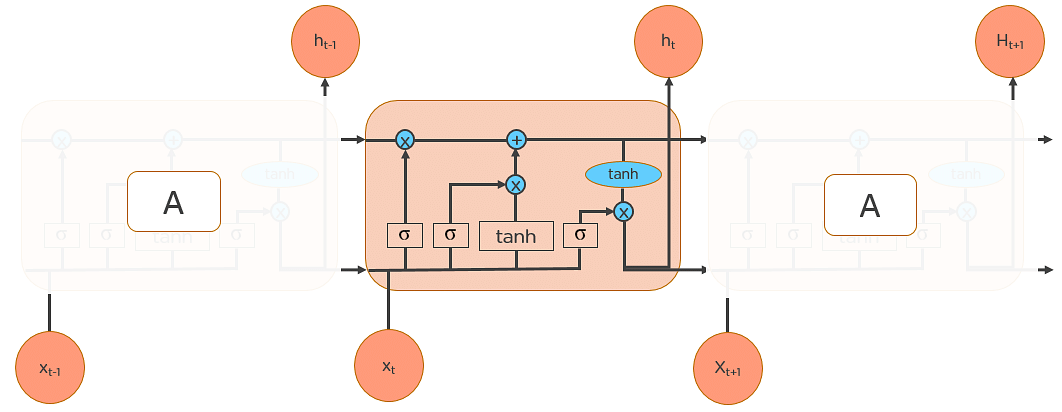
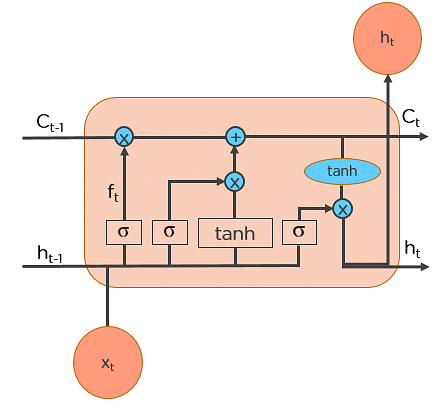


Fig: Long Short Term Memory Networks

LSTMs also have a chain-like structure, but the repeating module is a bit different structure. Instead of having a single neural network layer, four interacting layers are communicating extraordinarily.



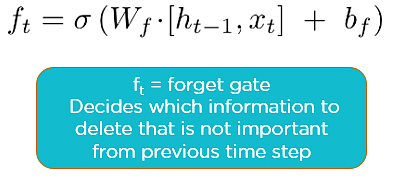
Workings of LSTMs in RNN



LSTMs work in a 3-step process.

Step 1: Decide How Much Past Data It Should Remember

The first step in the LSTM is to decide which information should be omitted from the cell in that particular time step. The sigmoid function determines this. It looks at the previous state (ht-1) along with the current input xt and computes the function.



Consider the following two sentences:

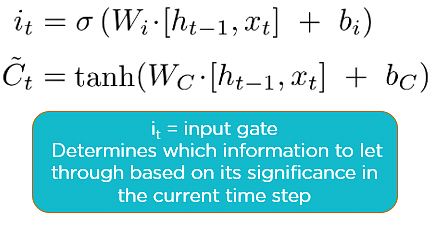
Let the output of h(t-1) be “Alice is good in Physics. John, on the other hand, is good at Chemistry.”

Let the current input at x(t) be “John plays football well. He told me yesterday over the phone that he had served as the captain of his college football team.”

The forget gate realizes there might be a change in context after encountering the first full stop. It compares with the current input sentence at x(t). The next sentence talks about John, so the information on Alice is deleted. The position of the subject is vacated and assigned to John.

Step 2: Decide How Much This Unit Adds to the Current State

In the second layer, there are two parts. One is the sigmoid function, and the other is the tanh function. In the sigmoid function, it decides which values to let through (0 or 1). tanh function gives weightage to the values which are passed, deciding their level of importance (-1 to 1).

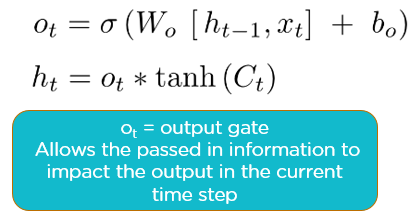


With the current input at x(t), the input gate analyzes the important information — John plays football, and the fact that he was the captain of his college team is important.

“He told me yesterday over the phone” is less important; hence it's forgotten. This process of adding some new information can be done via the input gate.

Step 3: Decide What Part of the Current Cell State Makes It to the Output

The third step is to decide what the output will be. First, we run a sigmoid layer, which decides what parts of the cell state make it to the output. Then, we put the cell state through tanh to push the values to be between -1 and 1 and multiply it by the output of the sigmoid gate.



Let’s consider this example to predict the next word in the sentence: “John played tremendously well against the opponent and won for his team. For his contributions, brave \_\_\_\_ was awarded player of the match.”

There could be many choices for the empty space. The current input brave is an adjective, and adjectives describe a noun. So, “John” could be the best output after brave.

LSTM Use Case

Now that you understand how LSTMs work, let’s do a practical implementation to predict the prices of stocks using the “Google stock price” data.

Based on the stock price data between 2012 and 2016, we will predict the stock prices of 2017.

1. Import the required libraries



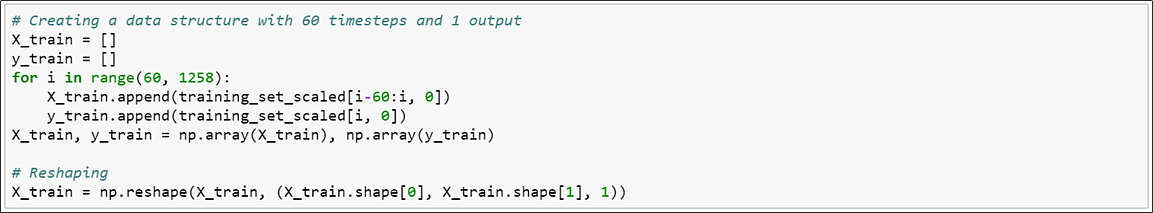
2. Import the training dataset



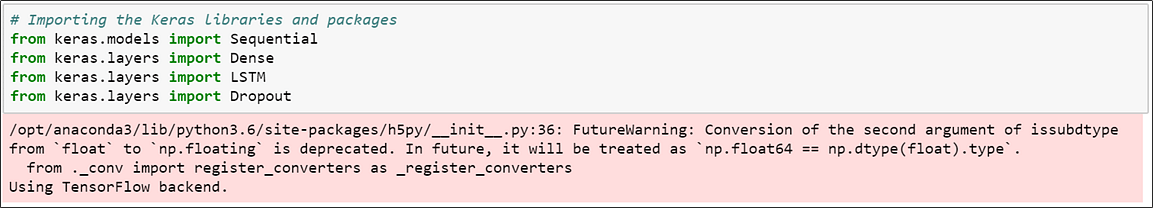
3. Perform feature scaling to transform the data



4. Create a data structure with 60-time steps and 1 output



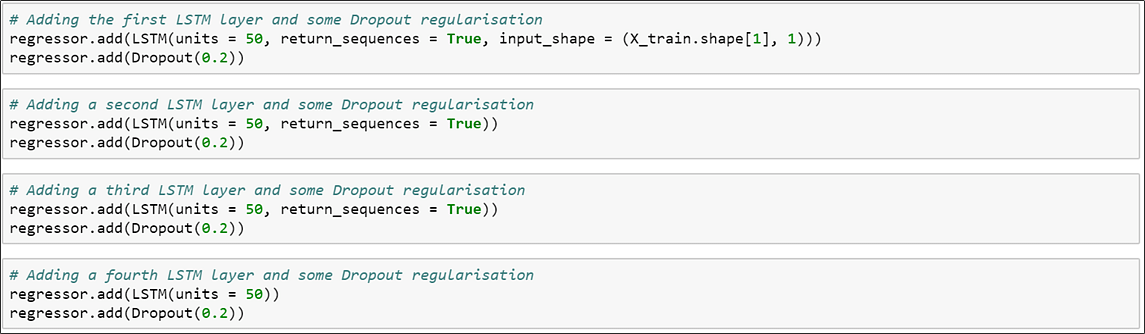
5. Import Keras library and its packages



6. Initialize the RNN



7. Add the LSTM layers and some dropout regularization.



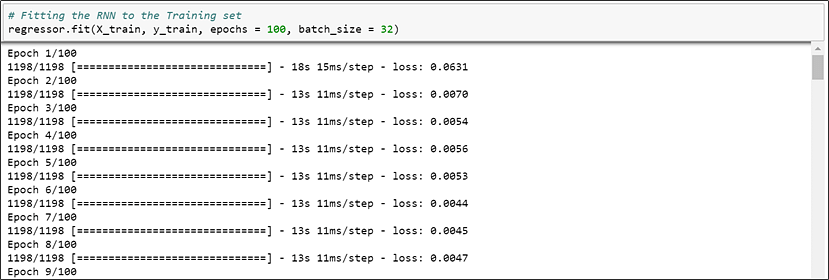
8. Add the output layer.



9. Compile the RNN



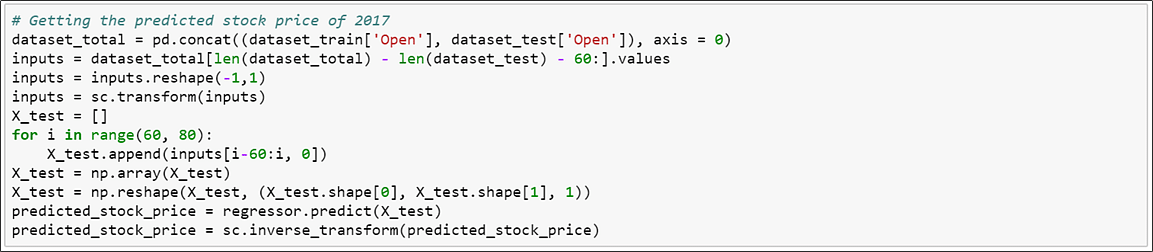
10. Fit the RNN to the training set



11. Load the stock price test data for 2017



12. Get the predicted stock price for 2017



13. Visualize the results of predicted and real stock price



Word Embeddings (Word2Vec, GloVe) and language modeling with RNNs

*A word embedding is a learned representation for text where words that have the same meaning and save similar representation*

Courtesy: Machinelearningmastery.com

* This approach to representing words and documents may be considered one of the key breakthroughs of deep learning on challenging NLP problems
* Word embeddings are alternative to one-hot encoding along with dimensionality reduction.

One-hot word vectors — Sparse, High-dimensional and Hard-coded

Word embeddings — Dense, Lower-Dimensional and Learned from the data

* Keras library has embeddings layer which does word representation of given text corpus

The provided code example demonstrates the training of a Word2Vec model using the Gensim library on a toy dataset. Tokenization of sentences, model training, and access to word embeddings are showcased.

[Understanding Word Embeddings and Building your First RNN Model - (analyticsvidhya.com)](https://www.analyticsvidhya.com/blog/2022/09/understanding-word-embeddings-and-building-your-first-rnn-model/) ----coding

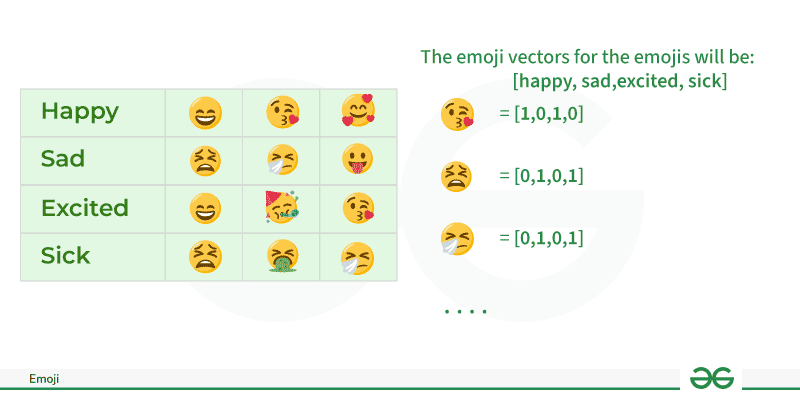
**MasterCard Stock Price Prediction Using LSTM & GRU**

[Recurrent Neural Network Tutorial (RNN) | DataCamp](https://www.datacamp.com/tutorial/tutorial-for-recurrent-neural-network) --code

**How are Word Embeddings used?**

* They are used as input to machine learning models.  
  Take the words —-> Give their numeric representation —-> Use in training or inference.
* To represent or visualize any underlying patterns of usage in the corpus that was used to train them.

Let’s take an example to understand how word vector is generated by taking emotions which are most frequently used in certain conditions and transform each emoji into a vector and the conditions will be our features.



**One-Hot Encoding**

One-hot encoding is a simple method for representing words in natural language processing (NLP). In this encoding scheme, each word in the vocabulary is represented as a unique vector, where the dimensionality of the vector is equal to the size of the vocabulary. The vector has all elements set to 0, except for the element corresponding to the index of the word in the vocabulary, which is set to 1.

* Python3

|  |
| --- |
| **def** one\_hot\_encode(text):      words **=** text.split()      vocabulary **=** set(words)      word\_to\_index **=** {word: i **for** i, word **in** enumerate(vocabulary)}      one\_hot\_encoded **=** []  **for** word **in** words:          one\_hot\_vector **=** [0] **\*** len(vocabulary)          one\_hot\_vector[word\_to\_index[word]] **=** 1          one\_hot\_encoded.append(one\_hot\_vector)    **return** one\_hot\_encoded, word\_to\_index, vocabulary    # sample  example\_text **=** "cat in the hat dog on the mat bird in the tree"    one\_hot\_encoded, word\_to\_index, vocabulary **=** one\_hot\_encode(example\_text)    print("Vocabulary:", vocabulary)  **print**("Word to Index Mapping:", word\_to\_index)  **print**("One-Hot Encoded Matrix:")  **for** word, encoding **in** zip(example\_text.split(), one\_hot\_encoded):      print(f"{word}: {encoding}") |

**Output:**

Vocabulary: {'mat', 'the', 'bird', 'hat', 'on', 'in', 'cat', 'tree', 'dog'}  
Word to Index Mapping: {'mat': 0, 'the': 1, 'bird': 2, 'hat': 3, 'on': 4, 'in': 5, 'cat': 6, 'tree': 7, 'dog': 8}  
One-Hot Encoded Matrix:  
cat: [0, 0, 0, 0, 0, 0, 1, 0, 0]  
in: [0, 0, 0, 0, 0, 1, 0, 0, 0]  
the: [0, 1, 0, 0, 0, 0, 0, 0, 0]  
hat: [0, 0, 0, 1, 0, 0, 0, 0, 0]  
dog: [0, 0, 0, 0, 0, 0, 0, 0, 1]  
on: [0, 0, 0, 0, 1, 0, 0, 0, 0]  
the: [0, 1, 0, 0, 0, 0, 0, 0, 0]  
mat: [1, 0, 0, 0, 0, 0, 0, 0, 0]  
bird: [0, 0, 1, 0, 0, 0, 0, 0, 0]  
in: [0, 0, 0, 0, 0, 1, 0, 0, 0]  
the: [0, 1, 0, 0, 0, 0, 0, 0, 0]  
tree: [0, 0, 0, 0, 0, 0, 0, 1, 0]

**Bag of Word (Bow)**

[Bag-of-Words (BoW)](https://www.geeksforgeeks.org/bag-of-words-bow-model-in-nlp/) is a text representation technique that represents a document as an unordered set of words and their respective frequencies. It discards the word order and captures the frequency of each word in the document, creating a vector representation.

* Python3

|  |
| --- |
| **from** sklearn.feature\_extraction.text **import** CountVectorizer  documents **=** ["This is the first document.",                "This document is the second document.",                "And this is the third one.",                "Is this the first document?"]    vectorizer **=** CountVectorizer()  X **=** vectorizer.fit\_transform(documents)  feature\_names **=** vectorizer.get\_feature\_names\_out()    **print**("Bag-of-Words Matrix:")  print(X.toarray())  print("Vocabulary (Feature Names):", feature\_names) |

**Output:**

Bag-of-Words Matrix:  
[[0 1 1 1 0 0 1 0 1]  
 [0 2 0 1 0 1 1 0 1]  
 [1 0 0 1 1 0 1 1 1]  
 [0 1 1 1 0 0 1 0 1]]  
Vocabulary (Feature Names): ['and' 'document' 'first' 'is' 'one' 'second' 'the' 'third' 'this']

**Neural Approach**

**2.1. Word2Vec**

[Word2Vec](https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/) is a neural approach for generating word embeddings. It belongs to the family of neural word embedding techniques and specifically falls under the category of distributed representation models. It is a popular technique in natural language processing (NLP) that is used to represent words as continuous vector spaces. Developed by a team at Google, Word2Vec aims to capture the semantic relationships between words by mapping them to high-dimensional vectors. The underlying idea is that words with similar meanings should have similar vector representations. In Word2Vec every word is assigned a vector. We start with either a random vector or **one-hot vector**.

There are two**neural embedding methods** for Word2Vec, Continuous Bag of Words (CBOW) and Skip-gram.

**2.2. Continuous Bag of Words(CBOW)**

[Continuous Bag of Words (CBOW)](https://www.geeksforgeeks.org/continuous-bag-of-words-cbow-in-nlp/) is a type of neural network architecture used in the Word2Vec model. The primary objective of CBOW is to predict a target word based on its context, which consists of the surrounding words in a given window. Given a sequence of words in a context window, the model is trained to predict the target word at the center of the window.

CBOW is a feedforward neural network with a single hidden layer. The input layer represents the context words, and the output layer represents the target word. The hidden layer contains the learned continuous vector representations (word embeddings) of the input words.

The architecture is useful for learning distributed representations of words in a continuous vector space.

A screenshot of a computer screen

Description automatically generated

The hidden layer contains the continuous vector representations (word embeddings) of the input words.

* The weights between the input layer and the hidden layer are learned during training.
* The dimensionality of the hidden layer represents the size of the word embeddings (the continuous vector space).
* Python3

|  |
| --- |
| **import** torch  **import** torch.nn as nn  **import** torch.optim as optim    # Define CBOW model  **class** CBOWModel(nn.Module):  **def** \_\_init\_\_(self, vocab\_size, embed\_size):          super(CBOWModel, self).\_\_init\_\_()          self.embeddings **=** nn.Embedding(vocab\_size, embed\_size)          self.linear **=** nn.Linear(embed\_size, vocab\_size)    **def** forward(self, context):          context\_embeds **=** self.embeddings(context).sum(dim**=**1)          output **=** self.linear(context\_embeds)  **return** output    # Sample data  context\_size **=** 2  raw\_text **=** "word embeddings are awesome"  tokens **=** raw\_text.split()  vocab **=** set(tokens)  word\_to\_index **=** {word: i **for** i, word **in** enumerate(vocab)}  data **=** []  **for** i **in** range(2, len(tokens) **-** 2):      context **=** [word\_to\_index[word] **for** word **in** tokens[i **-** 2:i] **+** tokens[i **+** 1:i **+** 3]]      target **=** word\_to\_index[tokens[i]]      data.append((torch.tensor(context), torch.tensor(target)))    # Hyperparameters  vocab\_size **=** len(vocab)  embed\_size **=** 10  learning\_rate **=** 0.01  epochs **=** 100    # Initialize CBOW model  cbow\_model **=** CBOWModel(vocab\_size, embed\_size)  criterion **=** nn.CrossEntropyLoss()  optimizer **=** optim.SGD(cbow\_model.parameters(), lr**=**learning\_rate)    # Training loop  **for** epoch **in** range(epochs):      total\_loss **=** 0  **for** context, target **in** data:          optimizer.zero\_grad()          output **=** cbow\_model(context)          loss **=** criterion(output.unsqueeze(0), target.unsqueeze(0))          loss.backward()          optimizer.step()          total\_loss **+=** loss.item()      print(f"Epoch {epoch + 1}, Loss: {total\_loss}")    # Example usage: Get embedding for a specific word  word\_to\_lookup **=** "embeddings"  word\_index **=** word\_to\_index[word\_to\_lookup]  embedding **=** cbow\_model.embeddings(torch.tensor([word\_index]))  **print**(f"Embedding for '{word\_to\_lookup}': {embedding.detach().numpy()}") |

**Output:**

Embedding for 'embeddings': [[-2.7053456 2.1384873 0.6417674 1.2882394 0.53470695 0.5651745  
 0.64166373 -1.1691749 0.32658175 -0.99961764]]

**Pretrained Word-Embedding**

Pre-trained word embeddings are representations of words that are learned from large corpora and are made available for reuse in various natural language processing (NLP) tasks. These embeddings capture semantic relationships between words, allowing the model to understand similarities and relationships between different words in a meaningful way.

**3.1. GloVe**

[GloVe](https://www.geeksforgeeks.org/pre-trained-word-embedding-using-glove-in-nlp-models/) is trained on global word co-occurrence statistics. It leverages the global context to create word embeddings that reflect the overall meaning of words based on their co-occurrence probabilities. this method, we take the corpus and iterate through it and get the co-occurrence of each word with other words in the corpus. We get a co-occurrence matrix through this. The words which occur next to each other get a value of 1, if they are one word apart then 1/2, if two words apart then 1/3 and so on.

Let us take an example to understand how the matrix is created. We have a small corpus:

Corpus:  
It is a nice evening.  
Good Evening!  
Is it a nice evening?

|  | **it** | **is** | **a** | **nice** | **evening** | **good** |
| --- | --- | --- | --- | --- | --- | --- |
| **it** | 0 |  |  |  |  |  |
| **is** | 1+1 | 0 |  |  |  |  |
| **a** | 1/2+1 | 1+1/2 | 0 |  |  |  |
| **nice** | 1/3+1/2 | 1/2+1/3 | 1+1 | 0 |  |  |
| **evening** | 1/4+1/3 | 1/3+1/4 | 1/2+1/2 | 1+1 | 0 |  |
| **good** | 0 | 0 | 0 | 0 | 1 | 0 |

The upper half of the matrix will be a reflection of the lower half. We can consider a window frame as well to calculate the co-occurrences by shifting the frame till the end of the corpus. This helps gather information about the context in which the word is used.

Initially, the vectors for each word is assigned randomly. Then we take two pairs of vectors and see how close they are to each other in space. If they occur together more often or have a higher value in the co-occurrence matrix and are far apart in space then they are brought close to each other. If they are close to each other but are rarely or not frequently used together then they are moved further apart in space.

After many iterations of the above process, we’ll get a vector space representation that approximates the information from the co-occurrence matrix. The performance of GloVe is better than Word2Vec in terms of both semantic and syntactic capturing

**from** gensim.models **import** KeyedVectors

**from** gensim.downloader **import** load

glove\_model **=** load('glove-wiki-gigaword-50')

word\_pairs **=** [('learn', 'learning'), ('india', 'indian'), ('fame', 'famous')]

# Compute similarity for each pair of words

**for** pair **in** word\_pairs:

    similarity **=** glove\_model.similarity(pair[0], pair[1])

**print**(f"Similarity between '{pair[0]}' and '{pair[1]}' using GloVe: {similarity:.3f}")

**Output:**

Similarity between 'learn' and 'learning' using GloVe: 0.802  
Similarity between 'india' and 'indian' using GloVe: 0.865  
Similarity between 'fame' and 'famous' using GloVe: 0.589

[Deep Learning - Text Generation using RNN (kaggle.com)](https://www.kaggle.com/code/manishguptads/deep-learning-text-generation-using-rnn)