DEEP LEARNING

UNIT-5

Sequence Modelling - Recurrent and Recursive Nets, Unfolding Computational Graphs, Recurrent Neural Networks, Bidirectional RNN, Deep Recurrent Networks, Echo state Networks, LSTM, Gated RNN's, Optimization for Long-term Dependencies, Auto Encoders, Deep Generative Models.

RECURRENT NEURAL NETWORK

A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data.

In this network the output of the previous step is fed as the input of to the current step.

The main and most important feature of RNN is HIDDEN STATE, which remembers some information about a sequence.

RNN have a "memory" which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output.

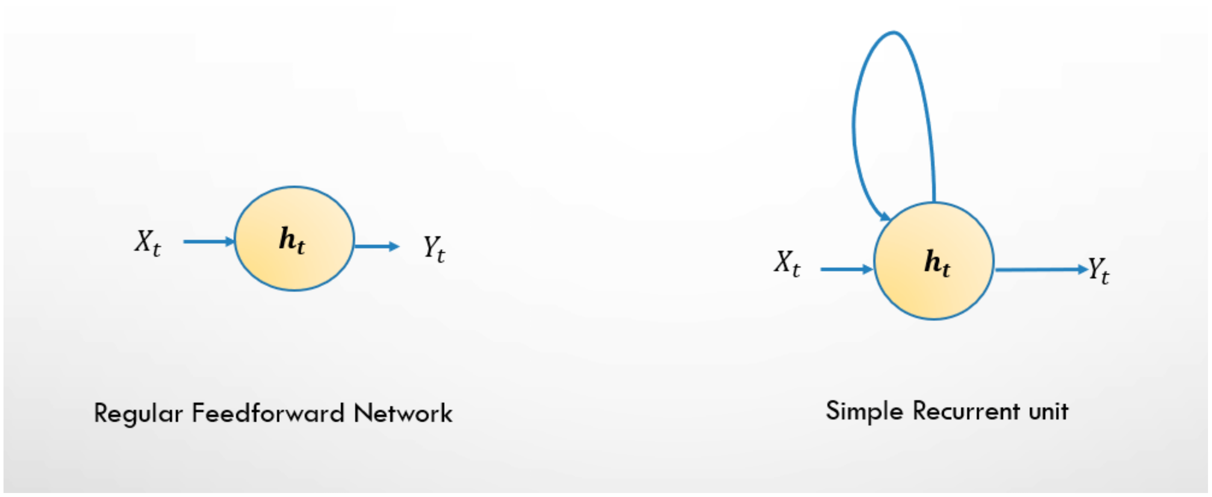
This reduces the complexity of parameters, unlike other neural networks.

To understand what is memory in RNN, what is recurrence unit in RNN, how do they store information of previous sequence, let's first understand the architecture of RNN.

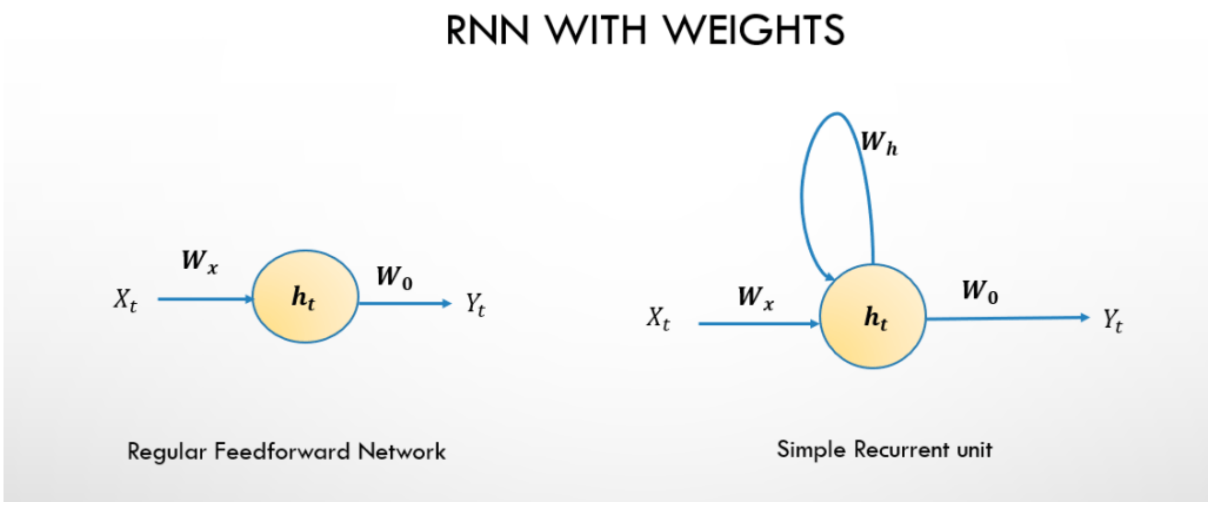
ARCHITECTURE OF RNN

So if you consider a simple feedforward neural network with one hidden layer. X_t is the input and Y_t is the output at timestep t then all we need is to create a feedback connection from hidden layer to itself to access information at timestep t-1. Feedback loop implies that there's a delay of one time unit. So one of input units into h_t is h_{t-1}, in turns hidden layer takes in both h_t and its own last value. So in nutshell this feedback loop allows information to be passed from one step of the network to the next and hence acts as memory in network.

The right diagram in below figure in below figure represents a simple Recurrent unit.



Below diagram depicts the architecture with weights –



So from above figure we can write below equations-

$$h_t = f\left(W_h{}^T h_{t-1} + W_x{}^T X_t + b_h \right)$$

$Y_t = \text{softmax}(W_O{}^T h_t + b_o)$

f= sigmoid, Relu, tanh

ADVANTAGES OF RNN

1. An RNN remembers each and every piece of information through time, useful in time series prediction.
2. RNN are even used with convolutional layer to extend the effective pixel neighbourhood.

DISADVANTAGES OF RNN

1. Gradient vanishing and exploding problems.
2. Training a RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

APPLICATIONS OF RNN

1. Language modelling and generating text.
2. Speech recognition.
3. Machine translation.
4. Image recognition, face detection.
5. Time series forecasting.

CONVOLUTIONAL NEURAL NETWORK

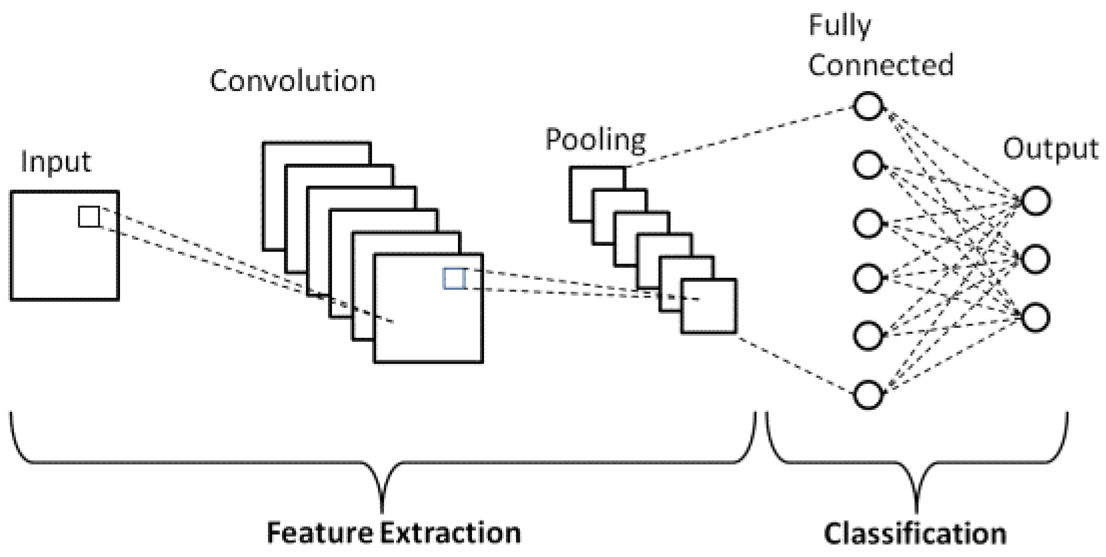
CNN are a class of Deep neural networks that can recognize and classify particular features from image and are widely used for analysing visual images.

Their applications range from image and video recognition, image classification, medical image analysis, computer vision and natural language processing because of its high accuracy.

ARCHITECTURE OF CNN

There are two main parts to a CNN architecture

- A convolution tool that separates and identifies the various features of the image for analysis in a process called as Feature Extraction.
- The network of feature extraction consists of many pairs of convolutional or pooling layers.
- A fully connected layer that utilizes the output from the convolution process and predicts the class of the image based on the features extracted in previous stages.
- This CNN model of feature extraction aims to reduce the number of features present in a dataset. It creates new features which summarises the existing features contained in an original set of features. There are many CNN layers as shown in the CNN architecture diagram.



1. Convolutional Layer

This layer is the first layer that is used to extract the various features from the input images. In this layer, the mathematical operation of convolution is performed between the input image and a filter of a particular size MxM. By sliding the filter over the input image, the dot product is taken between the filter and the parts of the input image with respect to the size of the filter (MxM).

2. Pooling Layer

The primary aim of this layer is to decrease the size of the convolved feature map to reduce the computational costs. This is performed by decreasing the connections between layers and independently operates on each feature map. Depending upon method used, there are several types of Pooling operations. It basically summarises the features generated by a convolution layer.

3. Fully Connected Layer

The Fully Connected (FC) layer consists of the weights and biases along with the neurons and is used to connect the neurons between two different layers. These layers are usually placed before the output layer and form the last few layers of a CNN Architecture.
In this, the input image from the previous layers are flattened and fed to the FC layer. The flattened vector then undergoes few more FC layers where the mathematical functions operations usually take place. In this stage, the classification process begins to take place. The reason two layers are connected is that two fully connected layers will perform better than a single connected layer. These layers in CNN reduce the human supervision

4. Dropout

Usually, when all the features are connected to the FC layer, it can cause overfitting in the training dataset. Overfitting occurs when a particular model works so well on the training data causing a negative impact in the model's performance when used on a new data.
To overcome this problem, a dropout layer is utilised wherein a few neurons are dropped from the neural network during training process resulting in reduced size of the model. On passing a dropout of 0.3, 30% of the nodes are dropped out randomly from the neural network.
Dropout results in improving the performance of a machine learning model as it prevents overfitting by making the network simpler. It drops neurons from the neural networks during training.

5. Activation Functions

Finally, one of the most important parameters of the CNN model is the activation function. They are used to learn and approximate any kind of continuous and complex relationship between variables of the network.
It adds non-linearity to the network. There are several commonly used activation functions such as the ReLU, Softmax, tanH and the Sigmoid functions.

ADVANTAGES OF CNN

The main advantage of CNN compared to its predecessors is that **it automatically detects the important features without any human supervision**. For example, given many pictures of cats and dogs it learns distinctive features for each class by itself. CNN is also computationally efficient.

It has the highest accuracy among all algorithms that predicts images.

DISADVANTAGES OF CNN

There are some drawbacks of CNN models which we have covered and attempts to fix it.
In short, the **disadvantages of CNN models** are:

- Classification of Images with different Positions
- Adversarial examples
- Coordinate Frame
- Other minor disadvantages like performance

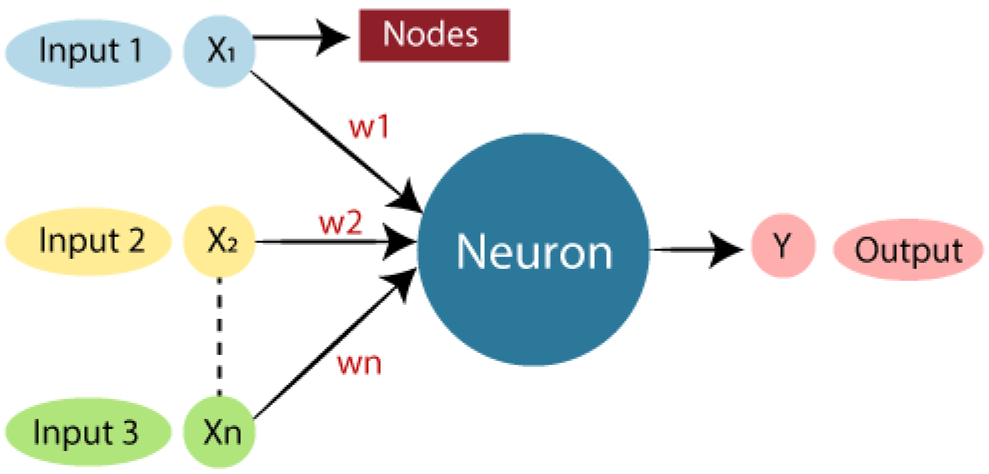
These disadvantages lead to other models/ ideas like Capsule neural network.

ARTIFICIAL NEURAL NETWORK

The term ANN is derived from biological neural networks that develop the structure of a human brain.

Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.

The typical ANN is as follows



An ANN attempts to mimic the network of neurons that makes up a human brain so that computers will have an option to understand things and make decisions in a human-like manner.

The ANN is designed by programming computers to simply behave like interconnected brain cells.

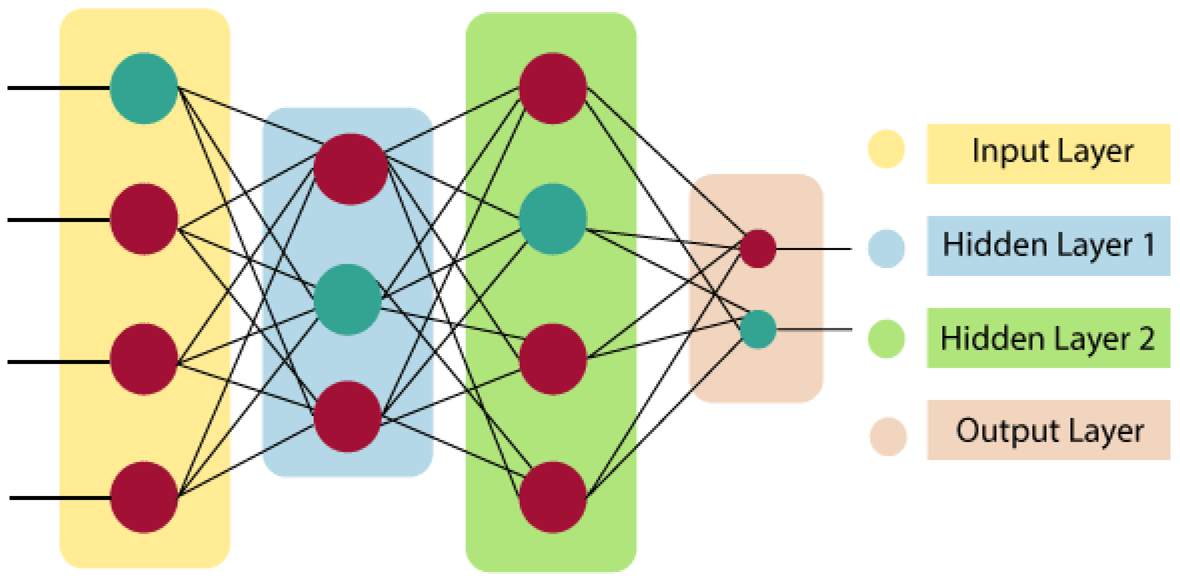
Relationship between Biological neural network and artificial neural network:

Biological Neural Network	**Artificial Neural Network**
Dendrites	Inputs
Cell nucleus	Nodes
Synapse	Weights
Axon	Output

Architecture of ANN

To understand the concept of the architecture of an artificial neural network, we have to understand what a neural network consists of. In order to define a neural network that consists of a large number of artificial neurons, which are termed units arranged in a sequence of layers. Let's us look at various types of layers available in an artificial neural network.

Artificial Neural Network primarily consists of three layers:



Input Layer:

As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer:

The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

Output Layer:

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

The ANN takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

$$\sum_{i=1}^{n} Wi * Xi + b$$

It determines weighted total is passed as an input to an activation function to produce the output. Activation functions choose whether a node should fire or not. Only those who are fired make it to the output layer. There are distinctive activation functions available that can be applied upon the sort of task we are performing.

ADVANTAGES OF ANN

1. Parallel processing capability
2. Storing data on the entire network
3. Capability to work with incomplete knowledge
4. Having a memory distribution
5. Having fault tolerance

DISADVANTAGES OF ANN

1. Assurance of proper network structure
2. Unrecognized behavior of the network
3. Hardware dependence
4. Difficulty of showing the issue to the network
5. The duration of the network is unknown

APPLICATIONS OF ANN

1. Social Media
2. Marketing and Sales
3. Healthcare
4. Personal Assistants

Bi-Directional Recurrent Neural Network

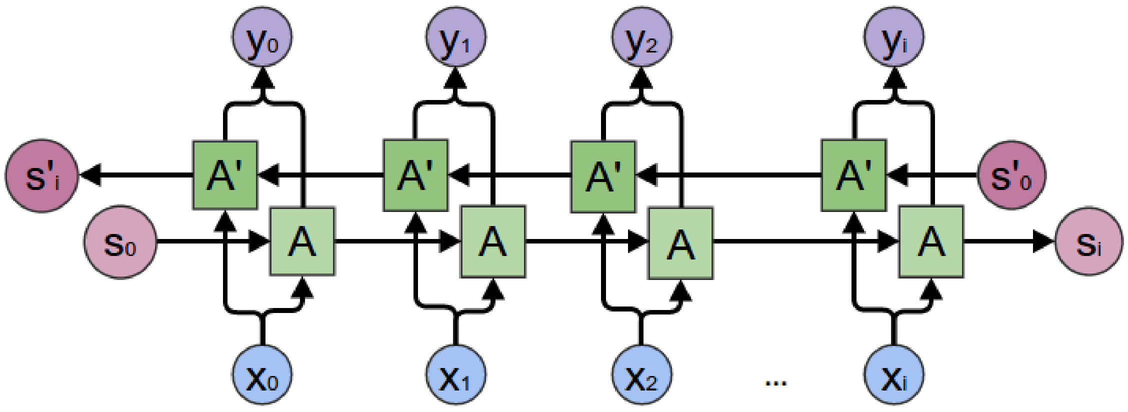
A typical state in an RNN (simple RNN, GRU, or LSTM) relies on the past and the present events. A state at time t depends on the states $X_1, X_2, \ldots, X_{t-1}$ and X_t. However, there can be situations where a prediction depends on the past, present, and future events.

For example, predicting a word to be included in a sentence might require us to look into the future, i.e., a word in a sentence could depend on a future event. Such linguistic dependencies are customary in several text prediction tasks.

Take speech recognition. When you use a voice assistant, you initially utter a few words after which the assistant interprets and responds. This interpretation may not entirely depend on the preceding words; the whole sequence of words can make sense only when the succeeding words are analyzed.

Thus, capturing and analyzing both past and future events is helpful in the above-mentioned scenarios.

To enable straight (past) and reverse traversal of input (future), Bidirectional RNNs, or BRNNs, are used. A BRNN is a combination of two RNNs - one RNN moves forward, beginning from the start of the data sequence, and the other, moves backward, beginning from the end of the data sequence. The network blocks in a BRNN can either be simple RNNs, GRUs, or LSTMs.



A BRNN has an additional hidden layer to accommodate the backward training process.

The training of a BRNN is similar to Back-Propagation Through Time (BPTT) algorithm. BPTT is the back-propagation algorithm used while training RNNs. A typical BPTT algorithm works as follows:

- Unroll the network and compute errors at every time step.
- Roll-up the network and update weights.

In a BRNN however, since there's forward and backward passes happening simultaneously, updating the weights for the two processes could happen at the same point of time. This leads to erroneous results. Thus, to accommodate forward and backward passes separately, the following algorithm is used for training a BRNN:

Forward Pass

- Forward states (from t = 1 to NN) and backward states (from t = N to 1) are passed.
- Output neuron values are passed (from t= 1 to N).

Backward Pass

- Output neuron values are passed (t = NN to 1).
- Forward states (from t= NN to 1) and backward states (from t = 1 to NN) are passed.

Both the forward and backward passes together train a BRNN.

Example: Bi Directional RNN for word sequence

Consider the word sequence “I love mango juice”. The forward layer would feed the sequence as such. But, the Backward Layer would feed the sequence in the reverse order “juice mango love I”. Now, the outputs would be generated by concatenating the word sequences at each time and generating weights accordingly. This can be used for POS tagging problems as well.

Applications

BRNN is useful for the following applications:

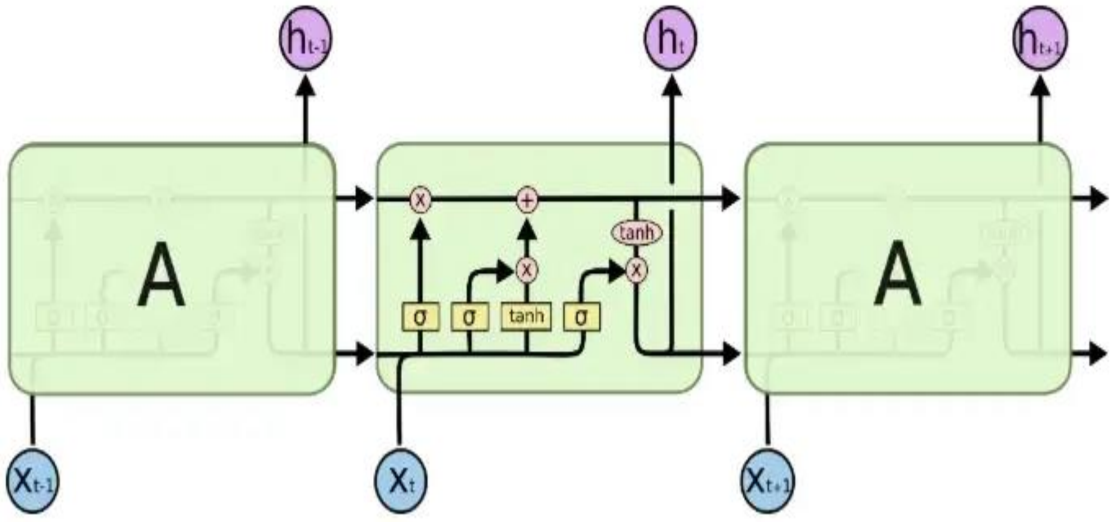
- Handwriting Recognition
- Speech Recognition
- Dependency Parsing
- Natural Language Processing

The bidirectional traversal idea can also be extended to 2D inputs such as images. We can have four RNNs each denoting one direction. Unlike a Convolutional Neural Network (CNN), a BRNN can assure long term dependency between the image feature maps.

LSTM (Long Short Term Memory)

When we have a small RNN, we would be able to effectively use the RNN because there is no problem of vanishing gradients. But, when we consider using long RNN’s there is not much we could do with the traditional RNN’s and hence it wasn’t widely used. That is the reason that lead to the finding of LSTM’s which basically uses a slightly different neuron structure. This was created with one basic thing in mind- the gradients shouldn’t vanish even if the sequence is very large.

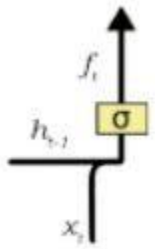
Long Short Term Memory networks (LSTMs) is a special kind of recurrent neural network capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber in 1997. Remembering information for longer periods of time is their default behavior. The Long short-term memory (LSTM) is made up of a memory cell, an input gate, an output gate and a forget gate. The memory cell is responsible for remembering the previous state while the gates are responsible for controlling the amount of memory to be exposed.



The memory cell is responsible for keeping track of the dependencies between the elements in the input sequence.The present input and the previous is passed to forget gate and the output of this forget gate is fed to the previous cell state. After that the output from the input gate is also fed to the previous cell state. By using this the output gate operates and will generate the output.

Forget Gate

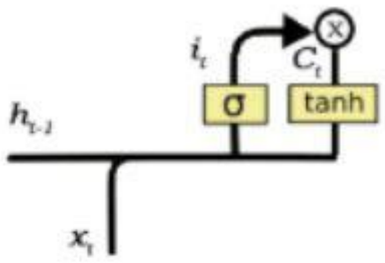
There are some information from the previous cell state that is not needed for the present unit in a LSTM. A forget gate is responsible for removing this information from the cell state. The information that is no longer required for the LSTM to understand or the information that is of less importance is removed via multiplication of a filter. This is required for optimizing the performance of the LSTM network. In other words we can say that it determines how much of previous state is to be passed to the next state.



Forget gate

The gate has two inputs X t and h t-1. h t-1 is the output of the previous cell and x t is the input at that particular time step. The given inputs are multiplied by the weight matrices and a bias is added. Following this, the sigmoid function(activation function) is applied to this value.

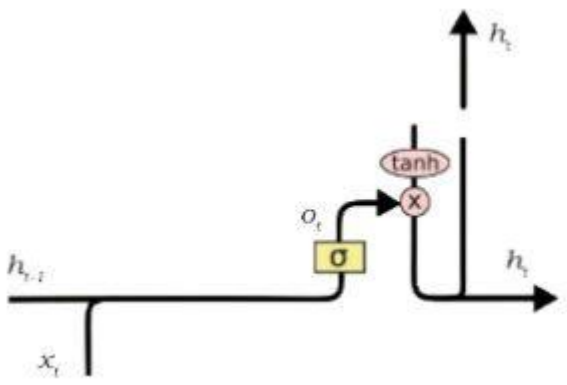
Input Gate



Input gate

The process of adding new information takes place in input gate. Here combination of x t and h t-1 is passed through sigmoid and tanh functions(activation functions) and added. Creating a vector containing all the possible values that can be added (as perceived from h t-1 and x t) to the cell state. This is done using the tanh function. By this step we ensure that only that information is added to the cell state that is important and is not redundant.

Output Gate



Output gate

A vector is created after applying tanh function to the cell state. Then making a filter using the values of h t-1 and x t, such that it can regulate the values that need to be output from the vector created above. This filter again employs a sigmoid function. Then both of them are multiplied to form output of that cell state.

Out of all the remarkable results achieved using recurrent neural network most of them are by using LSTM. The real magic behind LSTM networks is that they are achieving almost human-level of sequence generation quality, without any magic at all.

In LSTM, you can see that all the 3 sigmoid and 1 tanh activation functions for which the input would be a concatenation of h(t-1) and x(t), has different weights associated with them, say w(f), w(i), w(c) and w(o). Then the total parameters required for training an LSTM model is 4 times larger than a normal RNN. So, the computational cost is extremely higher. To solve this problem, they invented something called GRU.

Gated Recurrent Unit (GRU)

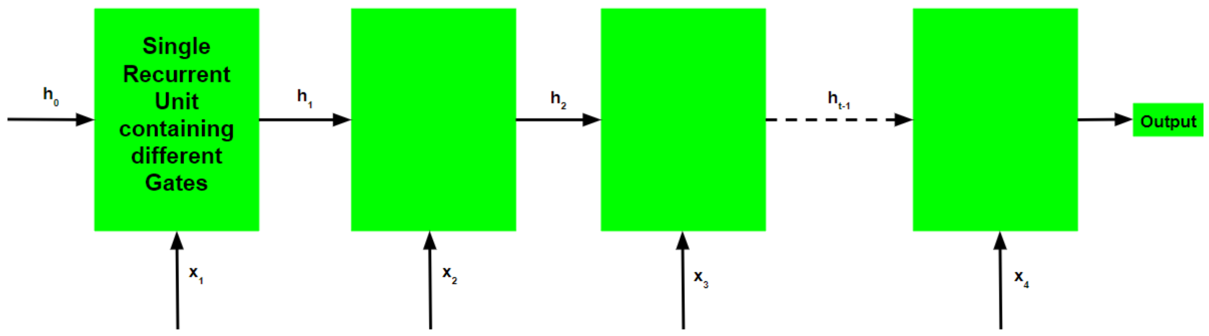
To solve the Vanishing-Exploding gradients problem often encountered during the operation of a basic Recurrent Neural Network, many variations were developed. One of the most famous variations is the **Long Short Term Memory Network(LSTM)**. One of the lesser-known but equally effective variations is the **Gated Recurrent Unit Network(GRU)**.

Unlike LSTM, it consists of only three gates and does not maintain an Internal Cell State. The information which is stored in the Internal Cell State in an LSTM recurrent unit is incorporated into the hidden state of the Gated Recurrent Unit. This collective information is passed onto the next Gated Recurrent Unit. The different gates of a GRU are as described below:-

1. **Update Gate(z):** It determines how much of the past knowledge needs to be passed along into the future. It is analogous to the Output Gate in an LSTM recurrent unit.
2. **Reset Gate(r):** It determines how much of the past knowledge to forget. It is analogous to the combination of the Input Gate and the Forget Gate in an LSTM recurrent unit.

3. **Current Memory Gate(   ):** It is often overlooked during a typical discussion on Gated Recurrent Unit Network. It is incorporated into the Reset Gate just like the Input Modulation Gate is a sub-part of the Input Gate and is used to introduce some non-linearity into the input and to also make the input Zero-mean. Another reason to make it a sub-part of the Reset gate is to reduce the effect that previous information has on the current information that is being passed into the future.

The basic work-flow of a Gated Recurrent Unit Network is similar to that of a basic Recurrent Neural Network when illustrated, the main difference between the two is in the internal working within each recurrent unit as Gated Recurrent Unit networks consist of gates which modulate the current input and the previous hidden state.



Note that just like the workflow, the training process for a GRU network is also diagrammatically similar to that of a basic Recurrent Neural Network and differs only in the internal working of each recurrent unit.

Encoder-Decoder Sequence to Sequence Model

A sequence to sequence model lies behind numerous systems which you face on a daily basis. For instance, seq2seq model powers applications like Google Translate, voice-enabled devices and online chatbots.

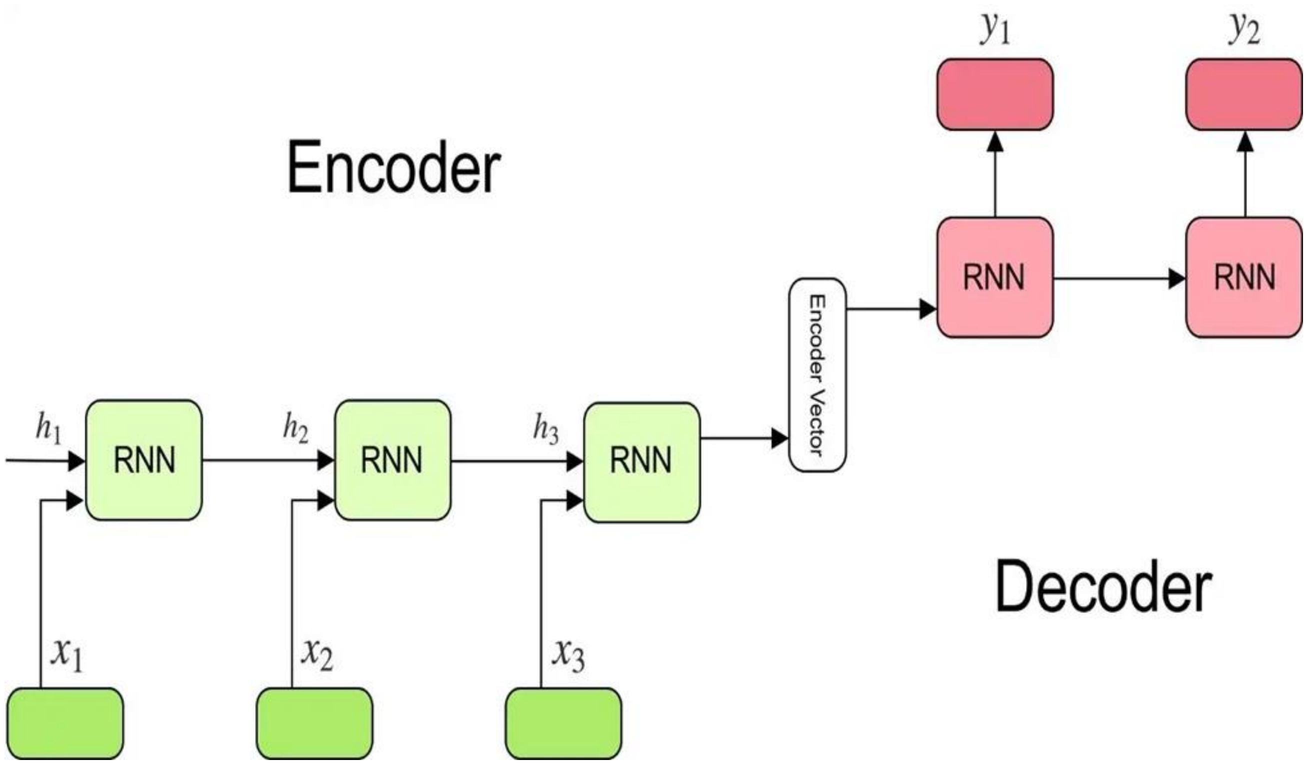
Definition of the Sequence to Sequence Model:

Introduced for the first time in 2014 by Google, a sequence to sequence model aims to map a fixed-length input with a fixed-length output where the length of the input and output may differ.

For example, translating "What are you doing today?" from English to Chinese has input of 5 words and output of 7 symbols (今天你在做什麼？). Clearly, we can't use a regular LSTM network to map each word from the English sentence to the Chinese sentence.

This is why the sequence to sequence model is used to address problems like that one.

In order to fully understand the model's underlying logic, we will go over the below illustration:



The model consists of 3 parts: encoder, intermediate (encoder) vector and decoder.

Encoder

- A stack of several recurrent units (LSTM or GRU cells for better performance) where each accepts a single element of the input sequence, collects information for that element and propagates it forward.

- In question-answering problem, the input sequence is a collection of all words from the question. Each word is represented as *x_i* where *i* is the order of that word.

- The hidden states *h_i* are computed using the formula:

$$h_t = f(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$$

This simple formula represents the result of an ordinary recurrent neural network. As you can see, we just apply the appropriate weights to the previous hidden state *h_(t-1)* and the input vector *x_t.*

Encoder Vector

- This is the final hidden state produced from the encoder part of the model. It is calculated using the formula above.

- This vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions.

- It acts as the initial hidden state of the decoder part of the model.

Decoder

- A stack of several recurrent units where each predicts an output *y_t* at a time step *t*.

- Each recurrent unit accepts a hidden state from the previous unit and produces and output as well as its own hidden state.

- In the question-answering problem, the output sequence is a collection of all words from the answer. Each word is represented as *y_i* where *i* is the order of that word.

- Any hidden state *h_i* is computed using the formula:

$$h_t = f(W^{(hh)} h_{t-1})$$

As you can see, we are just using the previous hidden state to compute the next one.

- The output *y_t* at time step *t* is computed using the formula:

$$y_t = softmax(W^S h_t)$$

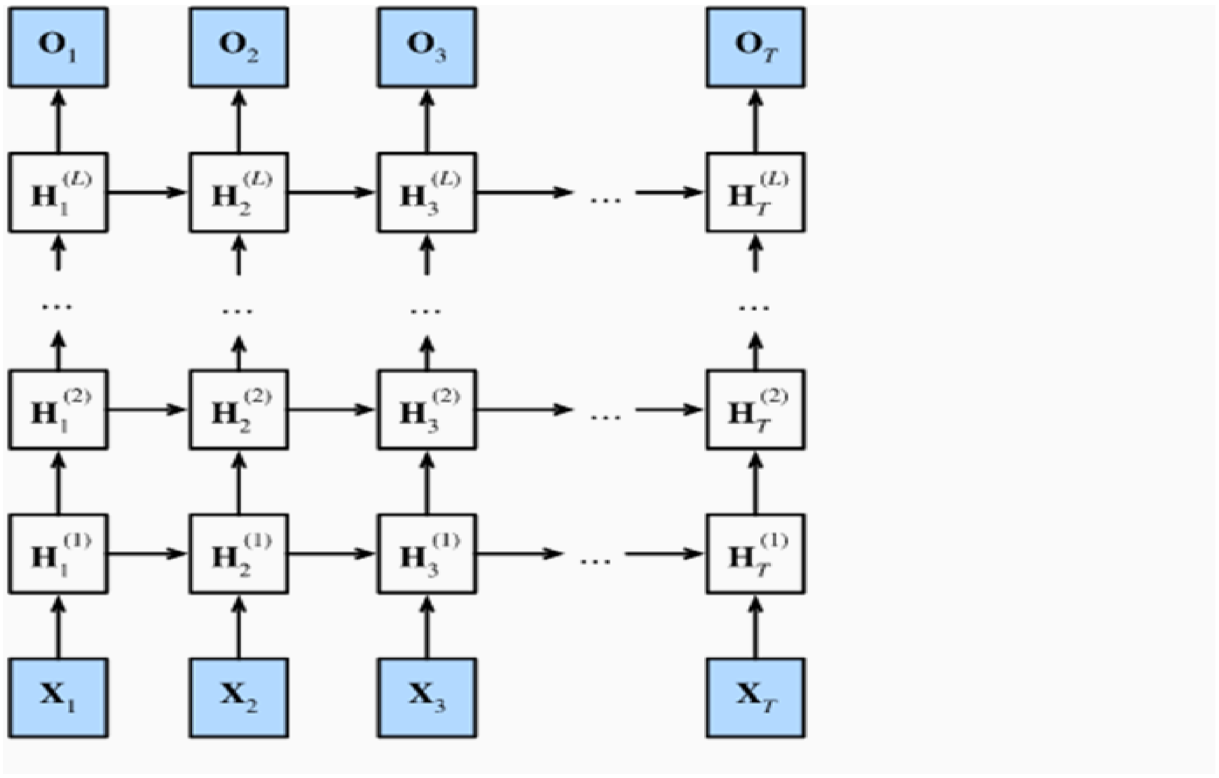
We calculate the outputs using the hidden state at the current time step together with the respective weight W(S). Softmax is used to create a probability vector which will help us determine the final output (e.g. word in the question-answering problem).

The power of this model lies in the fact that it can map sequences of different lengths to each other. As you can see the inputs and outputs are not correlated and their lengths can differ.

Deep Recurrent Networks

Up until now, we have focused on defining networks consisting of a sequence input, a single hidden RNN layer, and an output layer. Despite having just one hidden layer between the input at any time step and the corresponding output, there is a sense in which these networks are deep. Inputs from the first time step can influence the outputs at the final time step T (often 100s or 1000s of steps later). These inputs pass through T applications of the recurrent layer before reaching the final output. However, we often also wish to retain the ability to express complex relationships between the inputs at a given time step and the outputs at that same time step. Thus we often construct RNNs that are deep not only in the time direction but also in the input-to-output direction. This is precisely the notion of depth that we have already encountered in our development of MLPs and deep CNNs.

The standard method for building this sort of deep RNN is strikingly simple: we stack the RNNs on top of each other. Given a sequence of length T, the first RNN produces a sequence of outputs, also of length T. These, in turn, constitute the inputs to the next RNN layer. In this short section, we illustrate this design pattern and present a simple example for how to code up such stacked RNNs. Below, in the figure, we illustrate a deep RNN with L hidden layers. Each hidden state operates on a sequential input and produces a sequential output. Moreover, any RNN cell at each time step depends on both the same layer's value at the previous time step and the previous layer's value at the same time step.



<u>Echo state networks</u>

Echo state network is a type of Recurrent Neural Network, part of the *reservoir computing* framework, which has the following particularities:

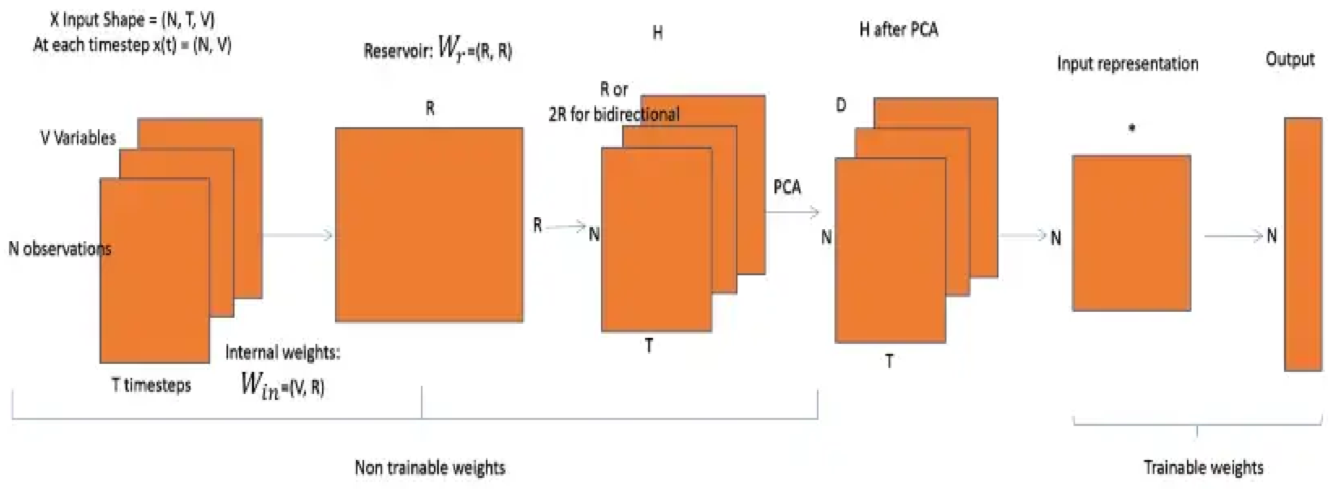
- the weights between the input -the hidden layer (the 'reservoir') :*Win* and also the weights of the 'reservoir': *Wr* are **randomly assigned** and **not trainable**
- the weights of the output neurons (the 'readout' layer) are **trainable** and can be learned so that the network can reproduce specific temporal patterns
- the hidden layer (or the 'reservoir') is very sparsely connected (typically < 10% connectivity)
- the reservoir architecture creates a recurrent **non linear** embedding (H on the image below) of the input which can be then connected to the desired output and these final weights will be trainable
- it is possible to connect the embedding to a different predictive model (a trainable NN or a ridge regressor/SVM for classification problems)

Reservoir Computing

Reservoir computing is an extension of neural networks in which the input signal is connected to a fixed (non-trainable) and random dynamical system (the reservoir), thus creating a higher dimension representation (embedding). This embedding is then connected to the desired output via trainable units.

The non-recurrent equivalent of reservoir computing is the Extreme Learning Machine and consists only of feed forward networks having only the readout layer trainable.

The figure below is a simplification of the paper Reservoir computing approaches for representation and classification of multivariate time series but it captures well the gist of ESNs.



$$\mathbf{h}(t) = f\left(\mathbf{W}_{\text{in}}\mathbf{x}(t) + \mathbf{W}_{\text{r}}\mathbf{h}(t-1)\right)$$

Echo State Networks are recurrent networks. f is a nonlinear function (such as tanh) which makes the current state dependent on the previous state and the current input

Workflow

For an input of shape N, T, V, where N is the number of observations, T is the number of time steps and V is the number of variables we will:

- choose the size of the reservoir R and other parameters governing the level of sparsity of connection, if we want to model a leakage, the ideal number of components after the dimensionality reduction, etc
- generate (V, R) input weights *Win* by sampling from a random binomial distribution

- generate (R, R) reservoir weights *Wr* by sampling from an uniform distribution with a given density, parameter which sets the level of sparsity

- calculate the high dimensional state representation H as a non linear function (typically tanh) of the input at the current time step (N, V) multiplied by the internal weights plus the previous state multiplied by the reservoir matrix (R, R)

- optionally we can run a dimensionality reduction algorithm such as PCA to D components, which brings H to (N, T, D)

- create an input representation either by using for example the entire reservoir and training a regressor to map states t to t+1: one representation could be the matrix of all calculated slopes and intercepts. Another option could be to use the mean or the last value of H

- connect this embedding to the desired output, either by using a NN structure which will be trainable or by using other types of predictors. The above mentioned paper suggest the use of Ridge regression

Why and when should you use Echo State Networks?

- Traditional NN architectures suffer from the vanishing/exploding gradient problem and as such, the parameters in the hidden layers either don't change that much or they lead to numeric instability and chaotic behavior. Echo state networks don't suffer from this problem

- Traditional NN architectures are computationally expensive, Echo State Networks are very fast as there is no back propagation phase on the reservoir

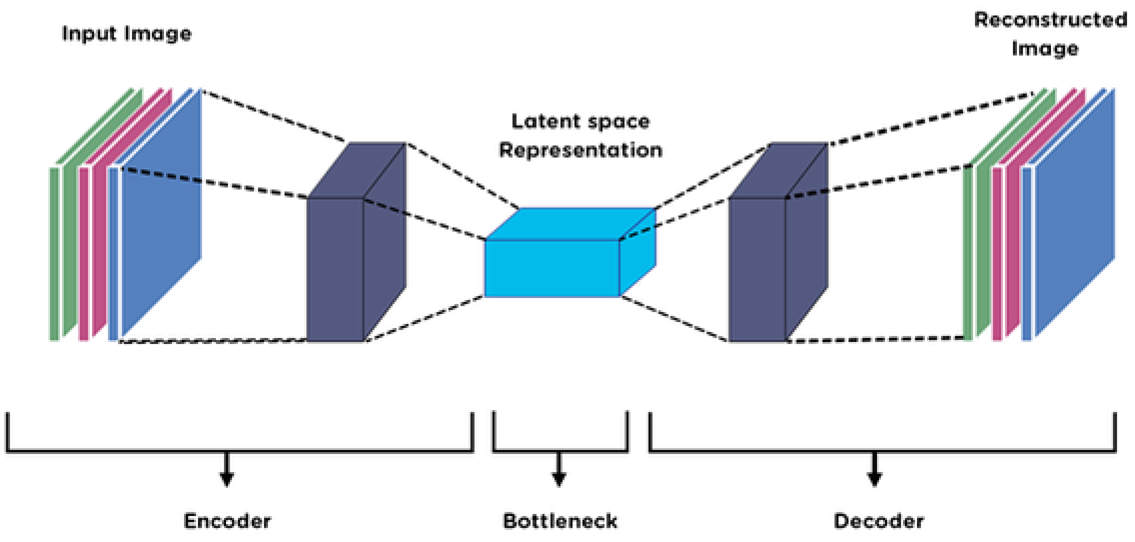
- Traditional NN can be disrupted by bifurcations

- ESN are well adapted for handling chaotic time series

Auto encoders

Auto-encoders are a type of neural network that attempts to mimic its input as closely as possible to its output. It aims to take an input, transform it into a reduced representation called embedding. Then, this embedding is transformed back into the original input. The embedding is also called the **latent-space representation.**

An auto-encoder has an hourglass-like structure that has two parts; an encoder and a decoder. When training an auto-encoder, we provide it with the input image as well as the target image. The goal here for the auto-encoder is to learn how closely it can mimic or copy or learn to recreate the target image given the input image.



Generally, the way an auto-encoder works is that the goal of the encoder section of the auto-encoder is to provide a much better latent representation or embedding. When each example is passed the encoder works on encoding it and converting it to embedding and the decoder learns to re-make the image or produce the target image from the embedding. The loss is calculated based on the distance between the input and the target image.

For example, there is a game where two children have to complete a task without speaking to each other so if you tell the first child a shape and a color that you want them to fill the image with then this will be your input image, the child has to pass this information to the second child, consider a scenario where the first child tries a lot of ways to make the second child understand the job. This right here is the encoder constantly updating and making better embedding for the decoder to understand. Now consider that the second child starts understanding and starts making some shapes and filling them with colors, now you know what the final image is supposed to look like, you will compare their final drawn image with the target image and this right here is you calculating the loss or how far off the model is from its target. Eventually, the children learn perfectly well to communicate and they have developed a code through which they know what shape to draw and what color to fill.

As the training process goes on the encoder constantly improves the embedding and the decoder constantly tries to make results better using that embedding.

Because of this kind of behavior auto-encoder and its variations are used for multiple tasks including style transfer, anomaly detection, one-class classification, etc. For anomaly detection consider the above example of children again. This time say for example instead of the usual shapes and the usual task you give them a shape that is completely irrelevant to the examples they have seen before and a color that they didn't make a code for earlier, now the tag team will get confused and they'll make a very bad output image.
We can use this to our advantage. So say we have only one kind of data that our encoder-

decoder tag team has perfectly learned to work with now whenever our model sees a sample different from the domain of training images it'll produce an output far off from the expected output. Based on the distance being much larger we can threshold and say since our model has never seen this kind of example before so most probably it's an anomaly.

Architecture

Let's explore the details of the architecture of the auto-encoder. An auto-encoder consists of three main components:

1. Encoder
2. Code or Embedding
3. Decoder

The encoder compresses the given input into a fixed dimension code or embedding and the decoder transforms that code or embedding the same as the original input. The decoder architecture is the mirror image of an encoder.

Some of the important things to know about the auto-encoders are:

- **Data-specific compression:** Auto-encoders compress the data that is similar to what it had been trained on. An auto-encoder trained on dog photos cannot compress human faces photos easily.
- **Unsupervised:** Training an auto-encoder is easy as we don't need labelled data. It is easily trained on any kind of input data.
- **Lossy in nature:** There is always going to be some difference between the input and output of the auto-encoder. The output will always have some missing information in it.

While building an auto-encoder, we aim to make sure that the auto-encoder does not memorize all the information. Instead, it should be constrained to prioritize which information should be kept, which information should be discarded. This constraint is introduced in the following ways:

1. Reducing the number of units or nodes in the layers.
2. Adding some noise to the input images.
3. Adding some regularization.

When we start building your first auto-encoder, you will encounter some of the hyper-parameters that will affect the performance of your auto-encoder. These hyper-parameters are discussed here:

1. **Numbers of layers:** You can keep as many layers in the encoder and decoder as you require. You can also choose how many nodes or units you want in your layers. Usually, the number of nodes decreases as we increase the number of layers in the encoder and vice-versa for the decoder.

2. **Number of nodes in the code layer:** It is always better to have a lesser number of nodes in this layer than the input size. A smaller size of code layer leads to better compression.

3. **Loss:** For the loss function, we generally use Mean Squared Error or Binary Cross Entropy. We are going to learn more about the loss function in the next section.

Types of Auto Encoders

There are multiple types or variants of auto-encoders that are developed for specific tasks.

1. Vanilla Autoencoder

2. Deep autoencoder

3. Convolutional autoencoder

4. Denoising autoencoder

5. Variational autoencoder

The feature learned by the auto-encoder can be used for other tasks like image classification or text classification. It is also useful for dimensionality reduction or compression of the data which can be important in some applications.