# ANN layers

In an ANN ,data flows from the input layer, through one or more hidden layers, to the output layer. Each layer consists of neurons that receive input, process it, and pass the output to the next layer. The layers work together to extract features, transform data, and make predictions.

An ANN typically consists of three primary types of layers:

- **Input Layer**
- **Hidden Layers**
- **Output Layer**

Each layer is composed of nodes (neurons) that are interconnected. The layers work together to process data through a series of transformations.

ANN Layers

**The Basic Layers in ANN**

**1. Input Layer**

The input layer is the first layer in an ANN and is responsible for receiving the raw input data. This layer's neurons correspond to the features in the input data. For example, in image processing, each neuron might represent a pixel value. The input layer doesn't perform any computations but passes the data to the next layer.

**Key Points:**

- **Role**: Receives raw data.
- **Function**: Passes data to the hidden layers.
- **Example**: For an image, the input layer would have neurons for each pixel value.

Input Layer in ANN

**2. Hidden Layers**

**Hidden Layers** are the intermediate layers between the input and output layers. They perform most of the computations required by the network. Hidden layers can vary in number and size, depending on the complexity of the task.

Each hidden layer applies a set of weights and biases to the input data, followed by an activation function to introduce non-linearity.

**3. Output Layer**

The **Output Layer** is the final layer in an ANN. It produces the output predictions. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem. The activation function used in the output layer depends on the type of problem:

- **Softmax** for multi-class classification
- **Sigmoid** for binary classification
- **Linear** for regression

**Types of Hidden Layers in Artificial Neural Networks**

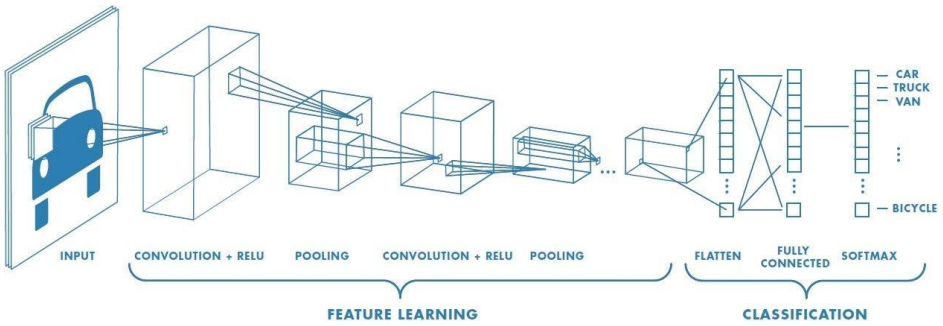
In deep learning, hidden layers are the layers between the input and output layers in a neural network. These layers are responsible for extracting features from the input data and transforming it into meaningful representations that can be used for tasks like classification, regression, or generation. There are various types of hidden layers, each with its own architecture and purpose. Below, I'll describe some common types of hidden layers and how they might be used in generating images.

**1. Fully Connected (Dense) Layer**

- **Description** : In a fully connected layer, every neuron in the layer is connected to every neuron in the previous layer. This type of layer is commonly used in traditional feedforward neural networks.
- **Use Case** : Fully connected layers are often used in the final stages of a neural network to combine features extracted by earlier layers.

**2. Convolutional Layer (Conv Layer)**

- **Description** : Convolutional layers apply a convolution operation to the input, passing the result to the next layer. They are designed to automatically and adaptively learn spatial hierarchies of features from input images.
- **Use Case** : Convolutional layers are widely used in image-related tasks such as image classification, object detection, and image generation.

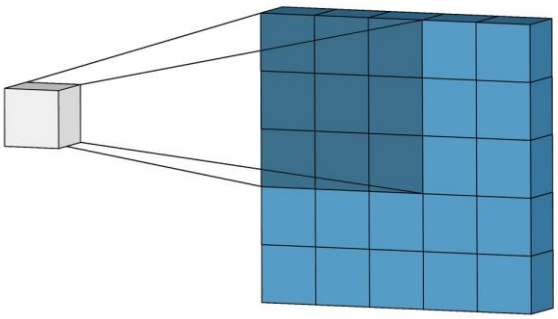


**3. Recurrent Layer (RNN, LSTM, GRU)**

- **Description** : Recurrent layers are designed to handle sequential data by maintaining a hidden state that captures information about previous inputs. Variants like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) are more advanced versions that address the vanishing gradient problem.
- **Use Case** : Recurrent layers are typically used in natural language processing (NLP) and time-series prediction, but they can also be used in image captioning or video generation.

**4. Deconvolutional Layer (Transposed Convolution)**

- **Description** : Deconvolutional layers, also known as transposed convolutional layers, perform the opposite operation of convolutional layers. They are used to upsample feature maps, increasing their spatial dimensions.
- **Use Case** : Deconvolutional layers are commonly used in image generation tasks, such as generating high-resolution images from low-resolution ones reconstructing images in autoencoders.



**5. Residual Layer (ResNet Block)**

- **Description** : Residual layers, introduced in ResNet (Residual Networks), allow the network to learn residual functions with reference to the layer inputs. This helps in training very deep networks by mitigating the vanishing gradient problem.
- **Use Case** : Residual layers are used in both discriminative and generative models to improve the flow of gradients during backpropagation.

**6. Attention Layer**

- **Description** : Attention mechanisms allow the model to focus on specific parts of the input when making predictions. Self-attention, transformer-based attention, and multi-head attention are popular variants.
- **Use Case** : Attention layers are widely used in NLP, but they have also been applied to computer vision tasks, including image generation.

**7. Batch Normalization Layer**

- **Description** : Batch normalization normalizes the inputs to each layer, stabilizing the learning process and allowing for faster convergence.
- **Use Case** : Batch normalization is commonly used in both discriminative and generative models to improve training stability.

**8. Dropout Layer**

- **Description** : Dropout is a regularization technique where randomly selected neurons are ignored during training. This prevents overfitting by ensuring that the network does not rely too heavily on specific neurons.
- **Use Case** : Dropout is used in both discriminative and generative models to prevent overfitting.

## What Are Activation Functions?

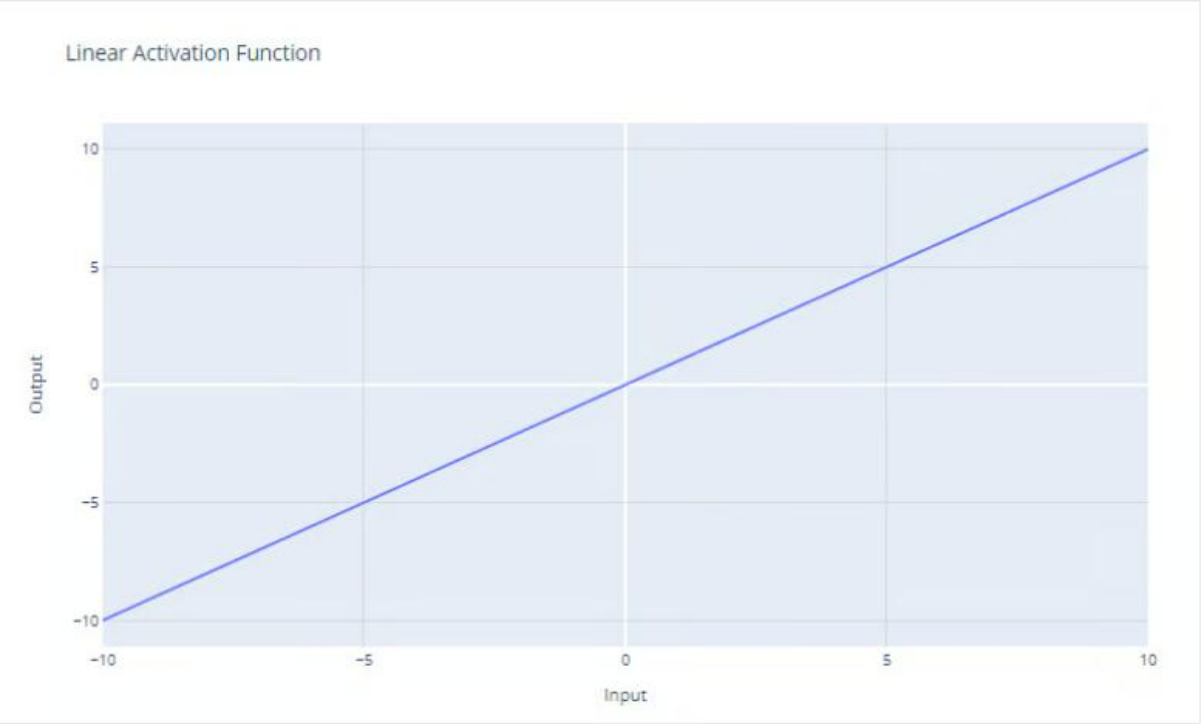
Activation functions are an integral building block of **neural networks** that enable them to learn complex patterns in data. They transform the input signal of a node in a neural network into an output signal that is then passed on to the next layer. Without activation functions, neural networks would be restricted to modeling only linear relationships between inputs and outputs.

Activation functions introduce non-linearities, allowing neural networks to learn highly complex mappings between inputs and outputs.

Choosing the right activation function is crucial for training neural networks that generalize well and provide accurate predictions. In this post, we will provide an overview of the most common activation functions, their roles, and how to select suitable activation functions for different use cases.

Whether you are just starting out in **deep learning** or are a seasoned practitioner, understanding activation functions in depth will build your intuition and improve your application of neural networks.

**Linear activation**




The linear activation function is the simplest activation function, defined as:

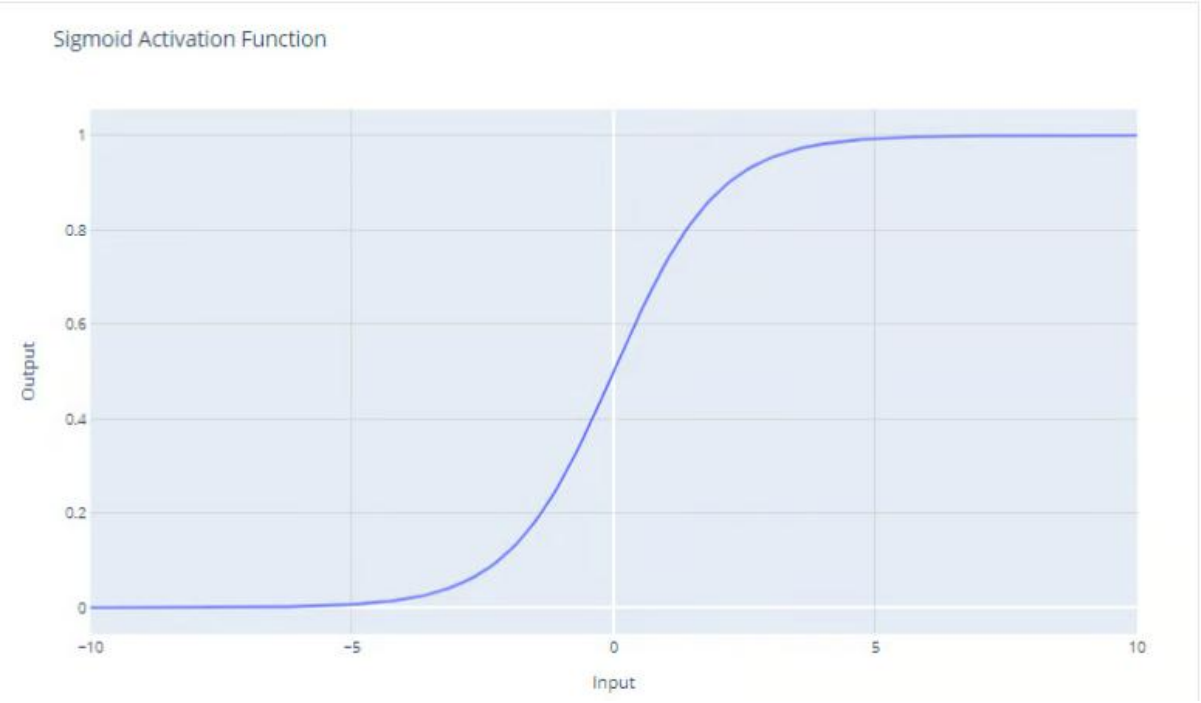
f(x) = x

It simply returns the input x as the output. Graphically, it looks like a straight line with a slope of 1.

The main use case of the linear activation function is in the output layer of a neural network used for regression. For regression problems where we want to predict a numerical value, using a linear activation function in the output layer ensures the neural network outputs a numerical value. The linear activation function does not squash or transform the output, so the actual predicted value is returned.

However, the linear activation function is rarely used in hidden layers of neural networks. This is because it does not provide any non-linearity. The whole point of hidden layers is to learn non-linear combinations of the input features. Using a linear activation throughout would restrict the model to just learning linear transformations of the input.

**Sigmoid activation**




The sigmoid activation function, often represented as σ(x), is a smooth, continuously differentiable function that is historically important in the development of neural networks. The sigmoid activation function has the mathematical form:

**f(x) = 1 / (1 + e^-x)**

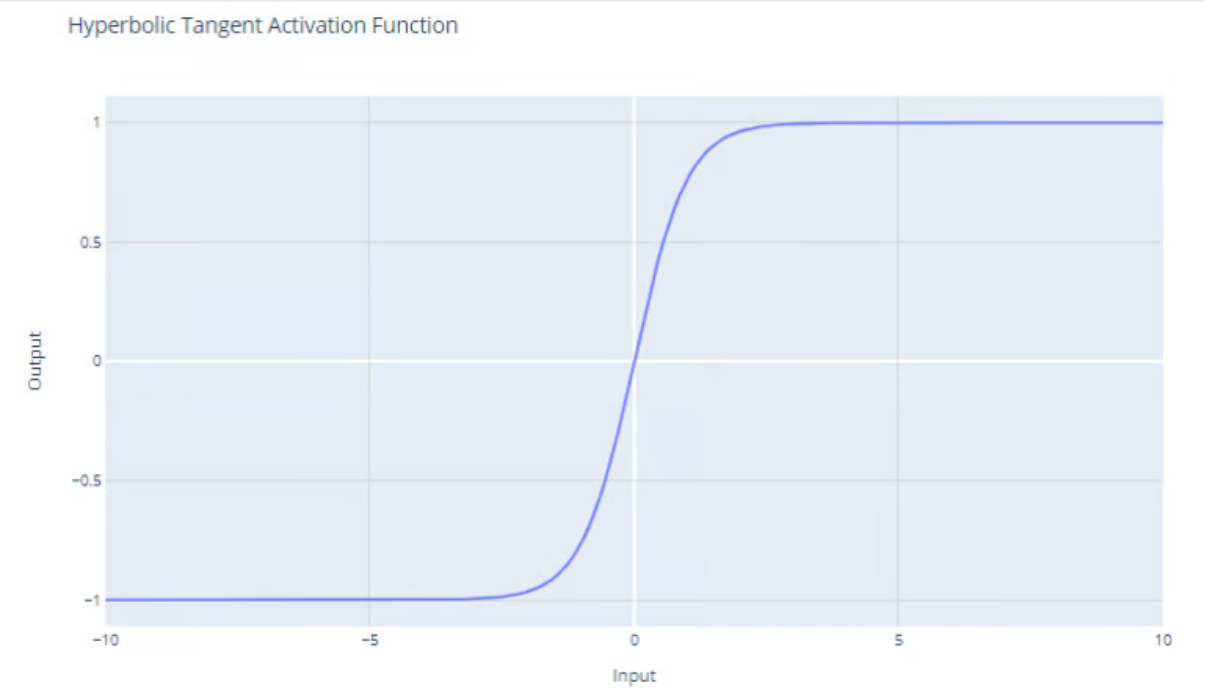
It takes a real-valued input and squashes it to a value between 0 and 1. The sigmoid function has an "S"-shaped curve that asymptotes to 0 for large negative numbers and 1 for large positive numbers. The outputs can be easily interpreted as probabilities, which makes it natural for binary classification problems.

Sigmoid units were popular in early neural networks since the gradient is strongest when the unit's output is near 0.5, allowing efficient backpropagation training. However, sigmoid units suffer from the "vanishing gradient" problem that hampers learning in deep neural networks.

As the input values become significantly positive or negative, the function saturates at 0 or 1, with an extremely flat slope. In these regions, the gradient is very close to zero. This results in very small changes in the weights during backpropagation, particularly for neurons in the earlier layers of deep networks, which makes learning painfully slow or even halts it. This is referred to as the vanishing gradient problem in neural networks.

The main use case of the sigmoid function is as the activation for the output layer of binary classification models. It squashes the output to a probability value between 0 and 1, which can be interpreted as the probability of the input belonging to a particular class.

**Tanh (hyperbolic tangent) activation**




The tanh (hyperbolic tangent) activation function is defined as:

**f(x) = (e^x - e^-x) / (e^x + e^-x)**

The tanh function outputs values in the range of -1 to +1. This means that it can deal with negative values more effectively than the sigmoid function, which has a range of 0 to 1.

Unlike the sigmoid function, tanh is zero-centered, which means that its output is symmetric around the origin of the coordinate system. This is often considered an advantage because it can help the learning algorithm converge faster.

Because the output of tanh ranges between -1 and +1, it has stronger gradients than the sigmoid function. Stronger gradients often result in faster learning and convergence during training because they tend to be more resilient against the problem of vanishing gradients when compared to the gradients of the sigmoid function.

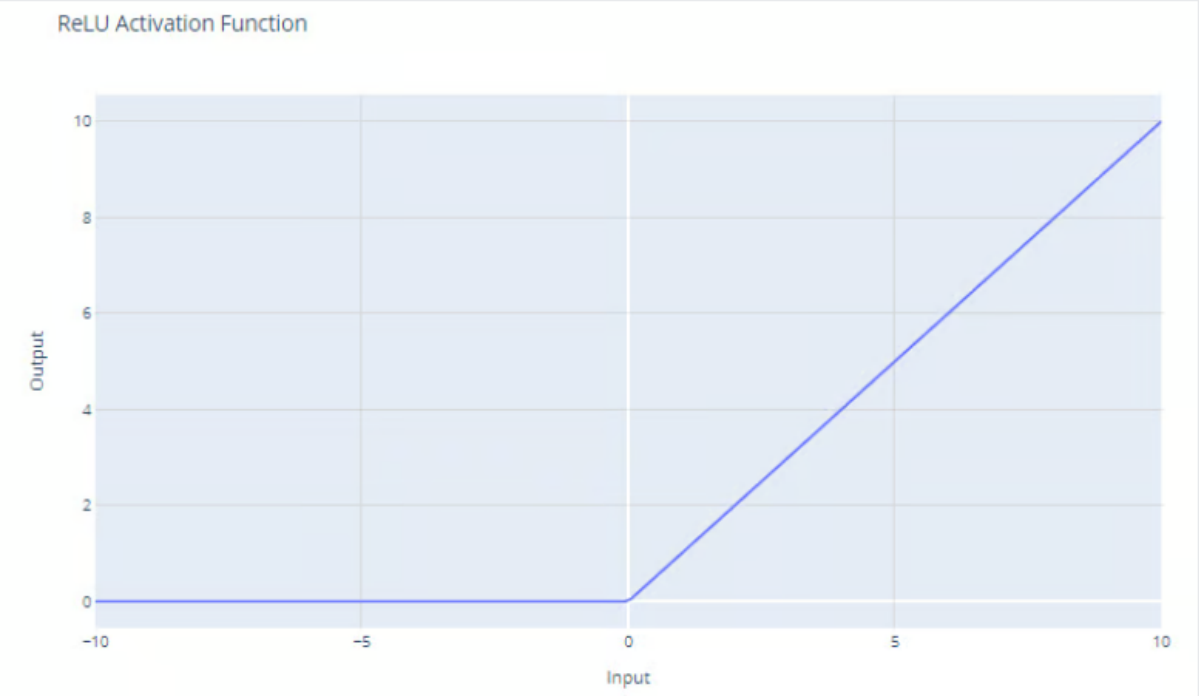
Despite these advantages, the tanh function still suffers from the vanishing gradient problem. During backpropagation, the gradients of the tanh function can become very small (close to zero). This issue is particularly problematic for deep networks with many layers; the gradients of the loss function may become too small to make significant changes in the weights during training as they propagate back to the initial layers. This can drastically slow down the training process and can lead to poor convergence properties.

The tanh function is frequently used in the hidden layers of a neural network. Because of its zero-centered nature, when the data is also normalized to have mean zero, it can result in more efficient training.

If one has to choose between the sigmoid and tanh and has no specific reason to prefer one over the other, tanh is often the better choice because of the reasons mentioned above. However, the decision can also be influenced by the specific use case and the behavior of the network during initial training experiments.

You can build a Simple Neural Network from scratch using PyTorch by following our tutorial by Kurtis Pykes, or, if you are an advanced user, our Deep Learning with PyTorch course is for you.

**ReLU (rectified linear unit) activation**




The Rectified Linear Unit (ReLU) activation function has the form:

**f(x) = max(0, x)**

It thresholds the input at zero, returning 0 for negative values and the input itself for positive values.

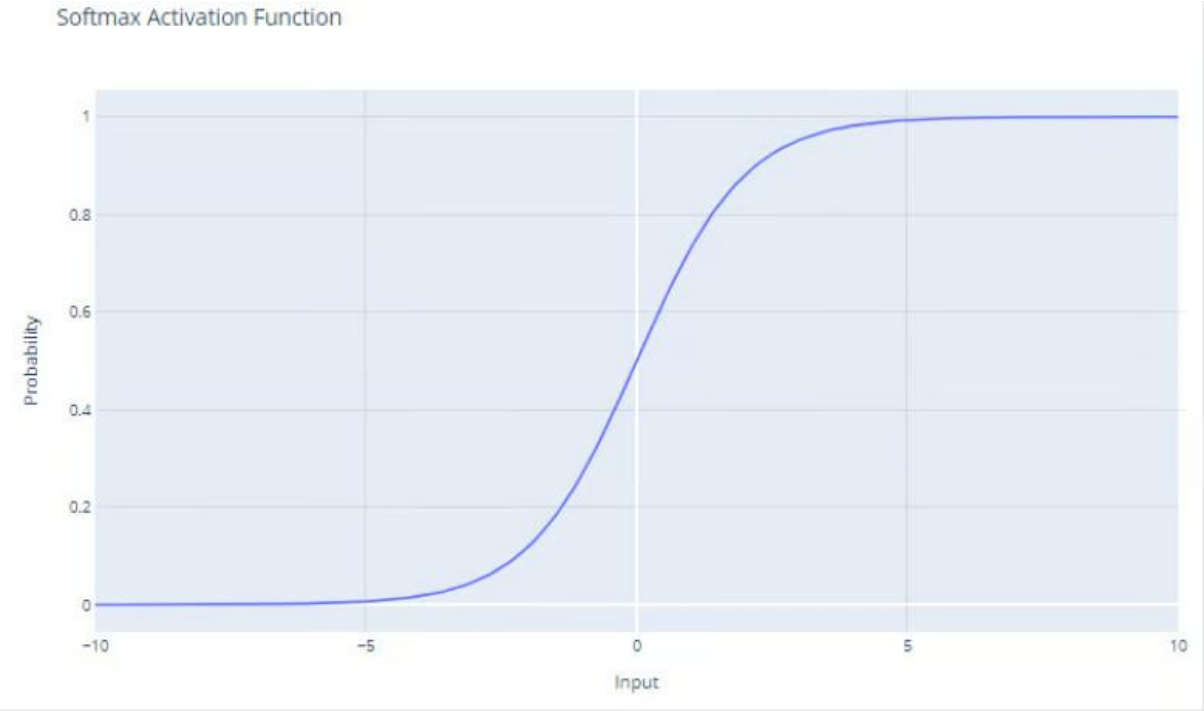
For inputs greater than 0, ReLU acts as a linear function with a gradient of 1. This means that it does not alter the scale of positive inputs and allows the gradient to pass through unchanged during backpropagation. This property is critical in mitigating the vanishing gradient problem.

Even though ReLU is linear for half of its input space, it is technically a non-linear function because it has a non-differentiable point at  x=0, where it abruptly changes from x. This non-linearity allows neural networks to learn complex patterns

Since ReLU outputs zero for all negative inputs, it naturally leads to sparse activations; at any time, only a subset of neurons are activated, leading to more efficient computation.

The ReLU function is computationally inexpensive because it involves simple thresholding at zero. This allows networks to scale to many layers without a significant increase in computational burden, compared to more complex functions like tanh or sigmoid.

**Softmax activation**




The softmax activation function, also known as the normalized exponential function, is particularly useful within the context of multi-class classification problems. This function operates on a vector, often referred to as the logits, which represents the raw predictions or scores for each class computed by the previous layers of a neural network.

For input vector x with elements x1, x2, ..., xC, the softmax function is defined as:

**f(xi) = e^xi / Σj e^xj**

The output of the softmax function is a probability distribution that sums up to one. Each element of the output represents the probability that the input belongs to a particular class.

The use of the exponential function ensures that all output values are non-negative. This is crucial because probabilities cannot be negative.

Softmax amplifies differences in the input vector. Even small differences in the input values can lead to substantial differences in the output probabilities, with the highest input value(s) tending to dominate in the resulting probability distribution.

Softmax is typically used in the output layer of a neural network when the task involves classifying an input into one of several (more than two) possible categories (multi-class classification).

The probabilities produced by the softmax function can be interpreted as confidence scores for each class, providing insight into the model's certainty about its predictions.

Because softmax amplifies differences, it can be sensitive to outliers or extreme values. For example, if the input vector has a very large value, softmax can "squash" the probabilities of other classes, leading to an overconfident model.

## ENCODER

The encoder is a set of convolutional blocks followed by pooling modules that compress the input to the model into a compact section called the bottleneck.

The bottleneck is followed by the decoder that consists of a series of upsampling modules to bring the compressed feature back into the form of an image. In case of simple autoencoders, the output is expected to be the same as the input data with reduced noise.

However, for variational autoencoders it is a completely new image, formed with information the model has been provided as input.

Here are five popular autoencoders that we will discuss:

1. Undercomplete autoencoders
2. Sparse autoencoders
3. Contractive autoencoders
4. Denoising autoencoders
5. Variational Autoencoders (for generative modelling)

**1. Undercomplete autoencoders**

An undercomplete autoencoder is one of the simplest types of autoencoders.

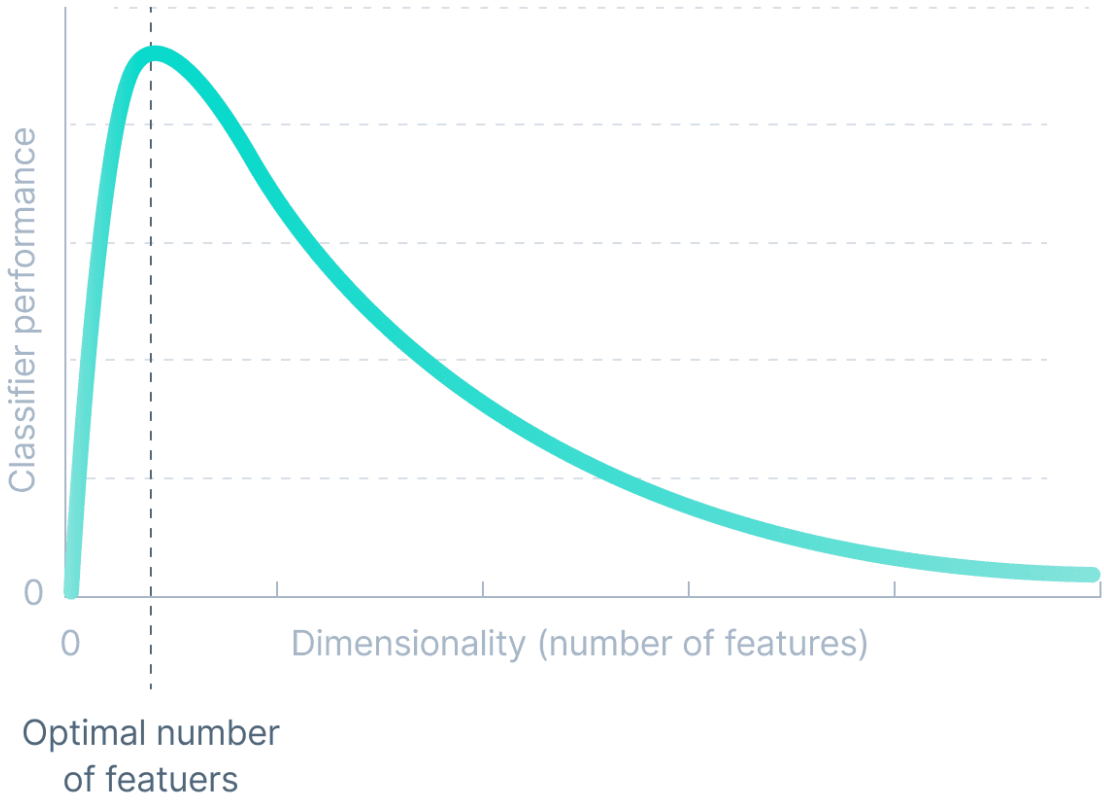
The way it works is very straightforward—

Undercomplete autoencoder takes in an image and tries to predict the same image as output, thus reconstructing the image from the compressed bottleneck region.

Undercomplete autoencoders are truly unsupervised as they do not take any form of label, the target being the same as the input.

The primary use of autoencoders like such is the generation of the latent space or the bottleneck, which forms a compressed substitute of the input data and can be easily decompressed back with the help of the network when needed.

This form of compression in the data can be modeled as a form of **dimensionality reduction**.




When we think of dimensionality reduction, we tend to think of methods like PCA (Principal Component Analysis) that form a lower-dimensional hyperplane to represent data in a higher-dimensional form without losing information.

However—

PCA can only build linear relationships. As a result, it is put at a disadvantage compared with methods like undercomplete autoencoders that can learn non-linear relationships and, therefore, perform better in dimensionality reduction.

This form of nonlinear dimensionality reduction where the autoencoder learns a non-linear manifold is also termed as *manifold learning*.

Effectively, if we remove all non-linear activations from an undercomplete autoencoder and use only linear layers, we reduce the undercomplete autoencoder into something that works at an equal footing with PCA.

The loss function used to train an undercomplete autoencoder is called *reconstruction loss,* as it is a check of how well the image has been reconstructed from the input data.

Although the reconstruction loss can be anything depending on the input and output, we will use an L1 loss to depict the term (also called the *norm loss*) represented by:

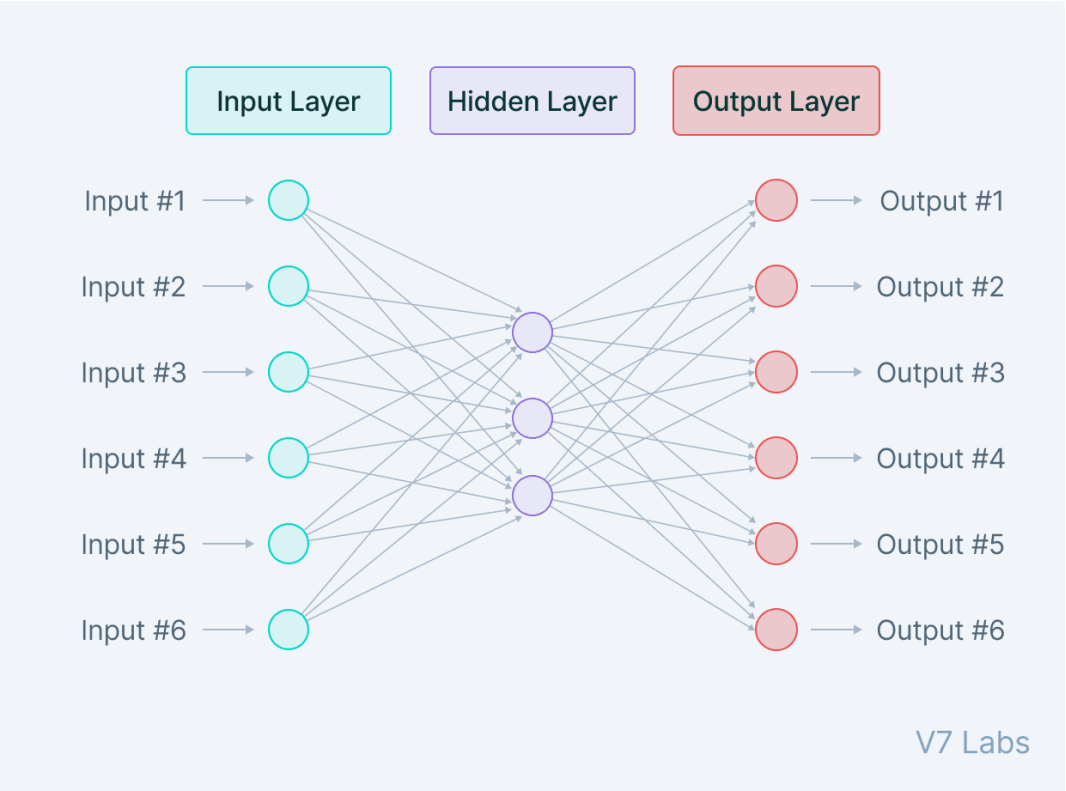
Where x^ represents the predicted output and x represents the ground truth.

As the loss function has no explicit regularisation term, the only method to ensure that the model is not memorising the input data is by regulating the size of the bottleneck and the number of hidden layers within this part of the network—the architecture.

**2. Sparse autoencoders**

Sparse autoencoders are similar to the undercomplete autoencoders in that they use the same image as input and ground truth. However—

The means via which encoding of information is regulated is significantly different.

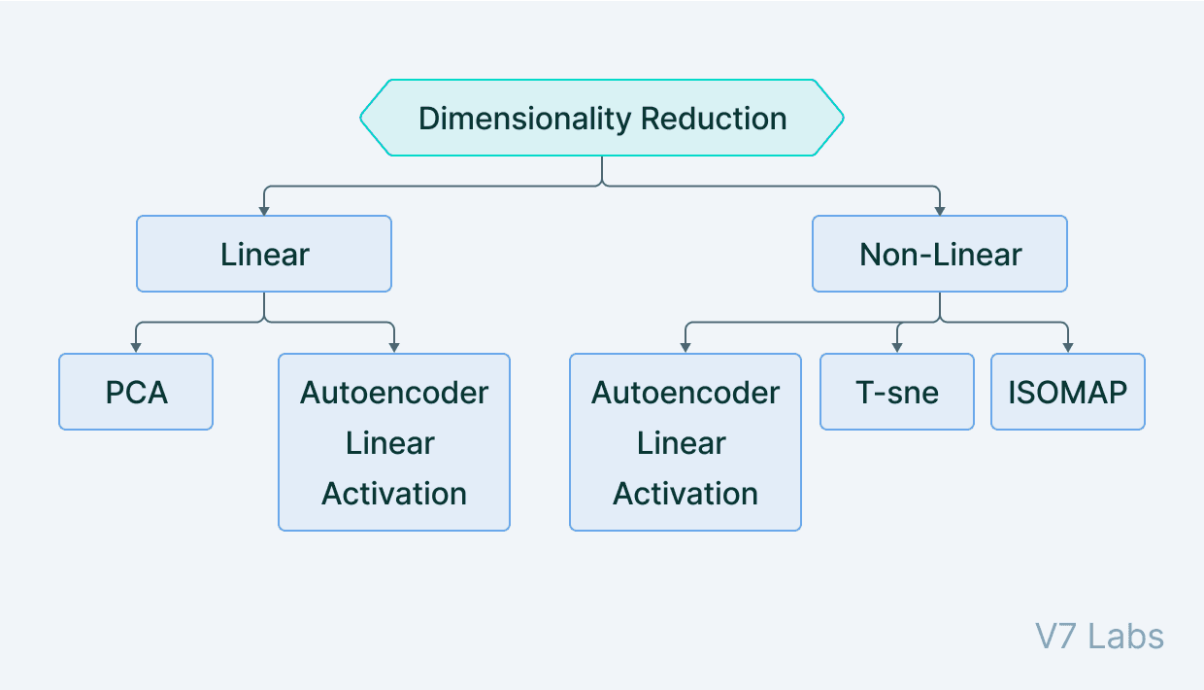


While undercomplete autoencoders are regulated and fine-tuned by regulating the size of the bottleneck, the sparse autoencoder is regulated by changing the number of nodes at each hidden layer.

Since it is not possible to design a neural network that has a flexible number of nodes at its hidden layers, sparse autoencoders work by penalizing the activation of some neurons in hidden layers.

In other words, the loss function has a term that calculates the number of neurons that have been activated and provides a penalty that is directly proportional to that.

This penalty, called the *sparsity function*, prevents the neural network from activating more neurons and serves as a regularizer.




While typical regularizers work by creating a penalty on the size of the weights at the nodes, sparsity regularizer works by creating a penalty on the number of nodes activated.

This form of regularization allows the network to have nodes in hidden layers dedicated to find specific features in images during training and treating the regularization problem as a problem separate from the latent space problem.

We can thus set latent space dimensionality at the bottleneck without worrying about regularization.

There are two primary ways in which the sparsity regularizer term can be incorporated into the loss function.

**L1 Loss**: In here, we add the magnitude of the sparsity regularizer as we do for general regularizers:

Where h represents the hidden layer, i represents the image in the minibatch, and a represents the activation.

**KL-Divergence**: In this case, we consider the activations over a collection of samples at once rather than summing them as in the L1 Loss method. We constrain the average activation of each neuron over this collection.

Considering the ideal distribution as a Bernoulli distribution, we include KL divergence within the loss to reduce the difference between the current distribution of the activations and the ideal (Bernoulli) distribution:

**3. Contractive autoencoders**

Similar to other autoencoders, contractive autoencoders perform task of learning a representation of the image while passing it through a bottleneck and reconstructing it in the decoder.

The contractive autoencoder also has a regularization term to prevent the network from learning the identity function and mapping input into the output.

Contractive autoencoders work on the basis that similar inputs should have similar encodings and a similar latent space representation. It means that the latent space should not vary by a huge amount for minor variations in the input.

To train a model that works along with this constraint, we have to ensure that the derivatives of the hidden layer activations are small with respect to the input data.

Mathematically:

$\delta h/\delta x$

Where h represents the hidden layer and x represents the input.

An important thing to note in the loss function (formed from the norm of the derivatives and the reconstruction loss) is that the two terms contradict each other.

While the reconstruction loss wants the model to tell differences between two inputs and observe variations in the data, the frobenius norm of the derivatives says that the model should be able to ignore variations in the input data.

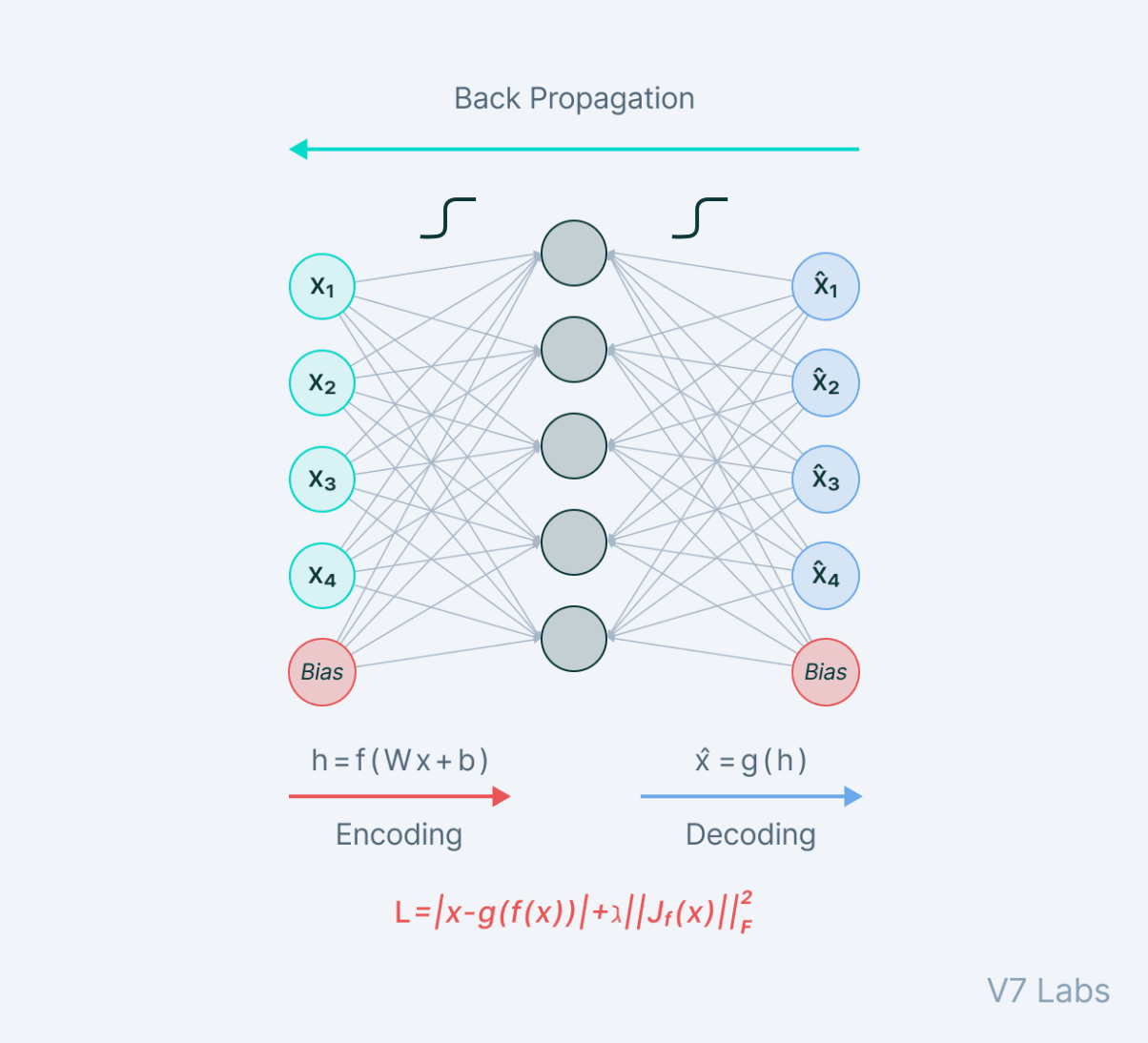
Putting these two contradictory conditions into one loss function enables us to train a network where the hidden layers now capture only the most essential information. This information is necessary to separate images and ignore information that is non-discriminatory in nature, and therefore, not important.

The total loss function can be mathematically expressed as:

L=x−x^+λ∑i∇xai(h)(x)2

Where h is the hidden layer for which a gradient is calculated and represented with respect to the input x as ∇xai(h)(x).

The gradient is summed over all training samples, and a frobenius norm of the same is taken.

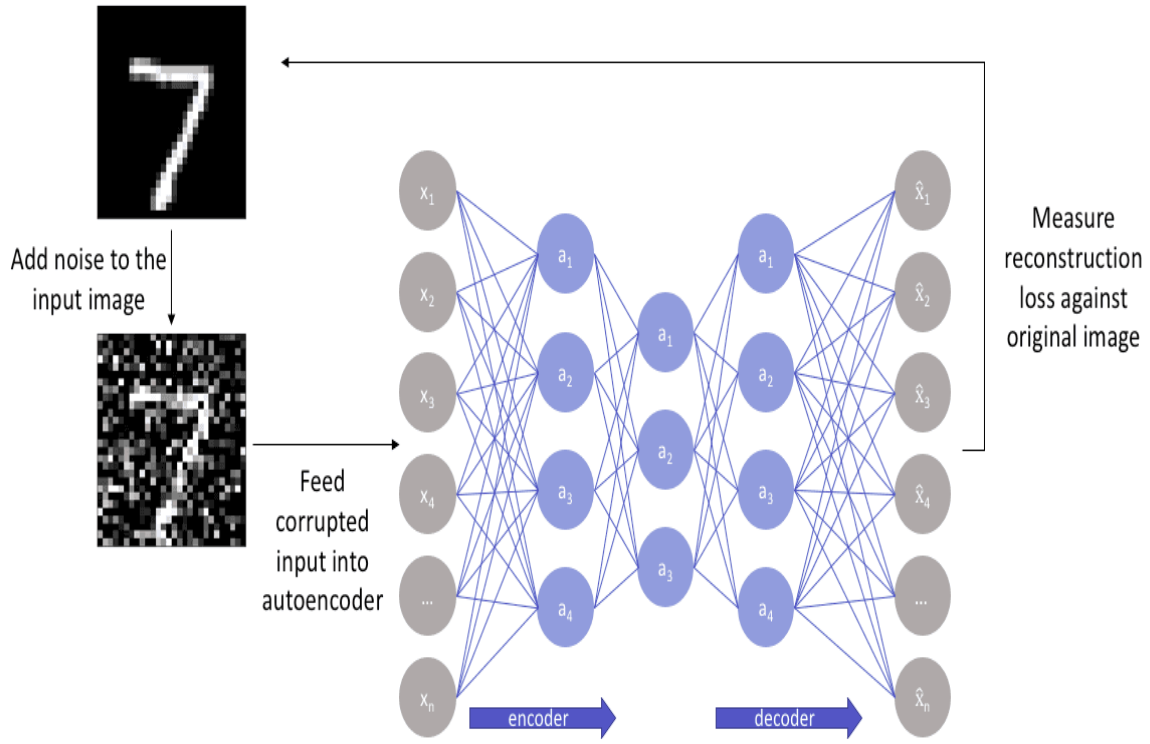



## 4. Denoising autoencoders

Denoising autoencoders, as the name suggests, are autoencoders that remove noise from an image.

As opposed to autoencoders we've already covered, this is the first of its kind that does not have the input image as its ground truth.

In denoising autoencoders, we feed a noisy version of the image, where noise has been added via digital alterations. The noisy image is fed to the encoder-decoder architecture, and the output is compared with the ground truth image.




The denoising autoencoder gets rid of noise by learning a representation of the input where the noise can be filtered out easily.

While removing noise directly from the image seems difficult, the autoencoder performs this by mapping the input data into a lower-dimensional manifold (like in undercomplete autoencoders), where filtering of noise becomes much easier.

Essentially, denoising autoencoders work with the help of non-linear dimensionality reduction. The loss function generally used in these types of networks is L2 or L1 loss.

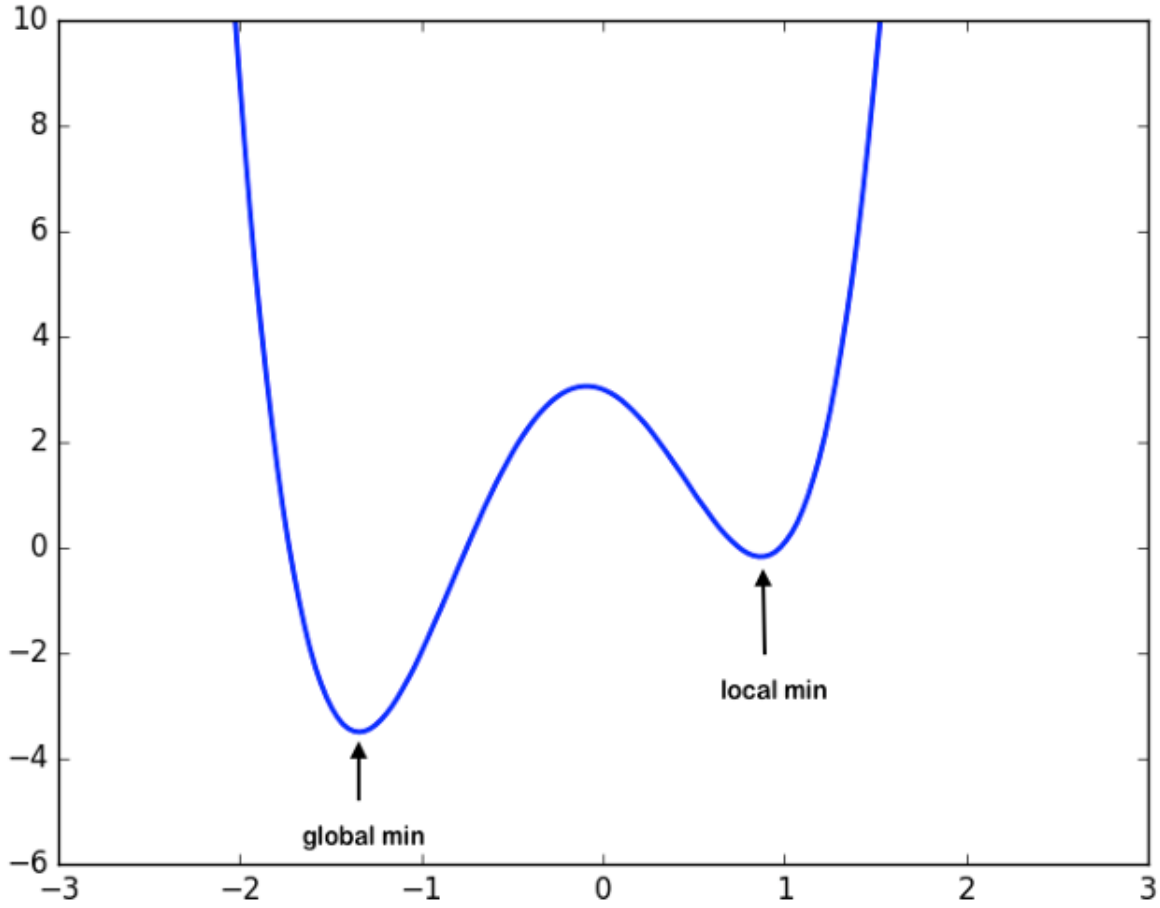
**5. Variational autoencoders**

Standard and variational autoencoders learn to represent the input just in a compressed form called the latent space or the bottleneck.

Therefore, the latent space formed after training the model is not necessarily continuous and, in effect, might not be easy to interpolate.

# Optimizers in Deep Learning

During the training process of a Neural Network, our aim is to try and minimize the loss function, by updating the values of the parameters (Weights) and make our predictions as accurate as possible. But how exactly do you do that? Then comes the question of how do you change the parameters of your model and by how much? Now if you use brute force method to identify the best parameters for your Deep Neural Network it will take about $3.42*10^{50}$ years for the world's fastest supercomputer Sunway Taihulight at a speed of 93 PFLOPS(Peta Fluid Operations/Sec), while a normal computer works at a speed of several Giga FLOPS. This is where optimizer comes into the picture. It tries to lower the loss function by updating the model parameters in response to the output of the loss function. Thereby helping to reach the Global Minima with the lowest loss and most accurate output.




Updating Weights: Before going deep into various types of optimizers, it is very essential to know that the most important function of the optimizer is to update the weights of the learning algorithm to reach the least cost function. Here is the formula used by all the optimizers for updating the weights with a certain value of the learning rate.

TYPES OF OPTIMIZERS :

1. Gradient Descent

2. Stochastic Gradient Descent

3. Adagrad

4. Adadelta

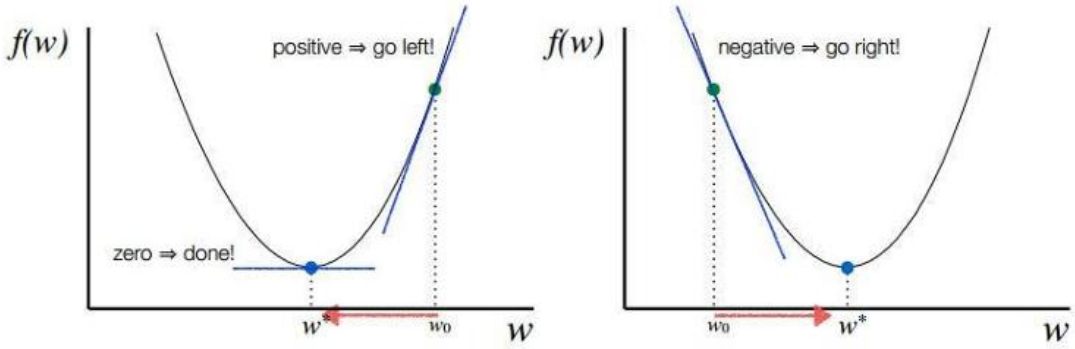
5. RMSprop

6. Adam

Gradient Descent :

This is one of the oldest and the most common optimizer used in neural networks, best for the cases where the data is arranged in a way that it possesses a convex

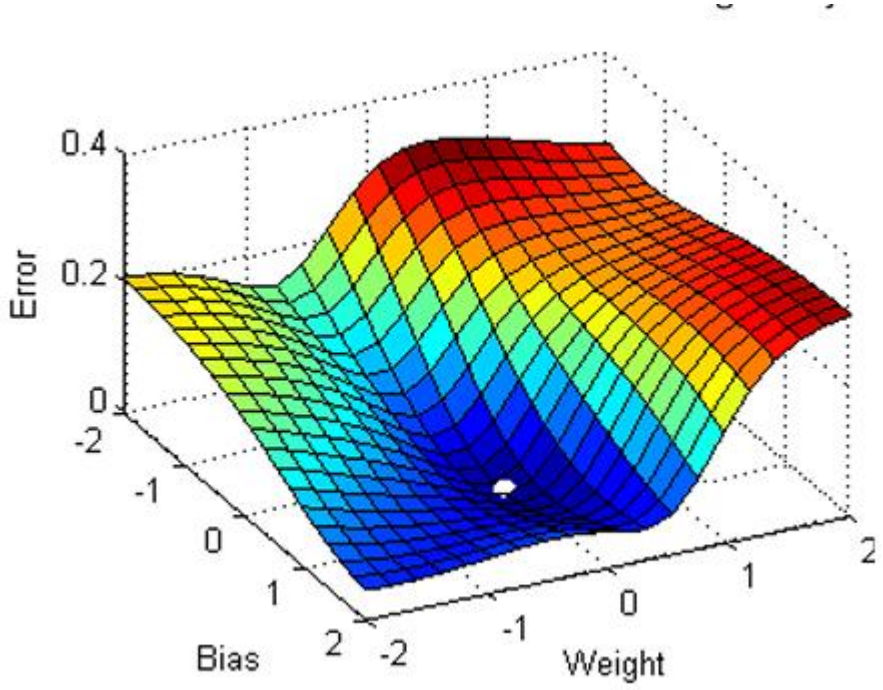
optimization problem. It will try to find the least cost function value by updating the weights of your learning algorithm and will come up with the best-suited parameter values corresponding to the Global Minima. This is done by moving down the hill with a negative slope, increasing the older weight, and positive slope reducing the older weight.




Stochastic Gradient Descent :

This is another variant of the Gradient Descent optimizer with an additional capability of working with the data with a non-convex optimization problem. The problem with such data is that the cost function results to rest at the local minima which are not suitable for your learning algorithm.

Rather than going for batch processing, this optimizer focuses on performing one update at a time. It is therefore usually much faster, also the cost function minimizes after each iteration (EPOCH). It performs frequent updates with a high variance that causes the objective function(cost function) to fluctuate heavily. Due to which it makes the gradient to jump to a potential Global Minima.




Adagrad :

This is the Adaptive Gradient optimization algorithm, where the learning rate plays an important role in determining the updated parameter values. Unlike Stochastic Gradient descent, this optimizer uses a different learning rate for each iteration(EPOCH) rather than using the same learning rate for determining all the parameters.

Thus it performs smaller updates(lower learning rates) for the weights corresponding to the high-frequency features and bigger updates(higher learning rates) for the weights corresponding to the low-frequency features, which in turn helps in better performance with higher accuracy. Adagrad is well-suited for dealing with sparse data.

So at each iteration, first the alpha at time t will be calculated and as the iterations increase the value of t increases, and thus alpha t will start increasing.