Layers in Artificial Neural Networks (ANN)

Introduction to 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**

Input Layer

- **Description**: The initial layer that receives the raw input data. Each neuron in this layer represents a feature of the input.

Hidden Layers

- **Description**: Intermediate layers that transform the data through weights, biases, and activation functions. These layers extract features and patterns from the input data.

Output Layer

- **Description**: The final layer that produces the network's predictions. Converts the processed data from the last hidden layer into the desired output format.

Types of Hidden Layers in Artificial Neural Networks

1. Dense (Fully Connected) Layer

A dense layer is the most common type of hidden layer in an ANN. Every neuron in a dense layer is connected to every neuron in the previous and subsequent layers. This layer performs a weighted sum of inputs and applies an activation function to introduce non-linearity.
The activation function (like ReLU, Sigmoid, or Tanh) helps the network learn complex patterns.

Key Points:

- **Role**: Learns representations from input data.
- **Function**: Performs weighted sum and activation.
- **Example**: Common in fully connected neural networks.

2. Convolutional Layer

Convolutional layers are primarily used in Convolutional Neural Networks (CNNs) for image processing tasks. They apply convolution operations to the input, capturing spatial hierarchies in the data. Convolutional layers use filters to scan across the input and generate feature maps. This helps in detecting edges, textures, and other visual features.

Key Points:

- **Role**: Extracts spatial features from images.
- **Function**: Applies convolution using filters.
- **Example**: Detects edges and textures in images.

3. Recurrent Layer

Recurrent layers, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), are used in Recurrent Neural Networks (RNNs) for sequence data like time series or natural language. They have connections that loop back, allowing information to persist across time steps. This makes them suitable for tasks where context and temporal dependencies are important.

Key Points:

- **Role**: Processes sequential data with temporal dependencies.
- **Function**: Maintains state across time steps.
- **Example**: Language modeling, time series prediction.

4. Dropout Layer

Dropout layers are a regularization technique used to prevent overfitting. They randomly drop a fraction of the neurons during training, which forces the network to learn more robust features and reduces dependency on specific neurons. During training, each neuron is retained with a probability ppp.

Key Points:

- **Role**: Prevents overfitting.
- **Function**: Randomly drops neurons during training.
- **Example**: Common in deep learning models to improve generalization.

5. Pooling Layer

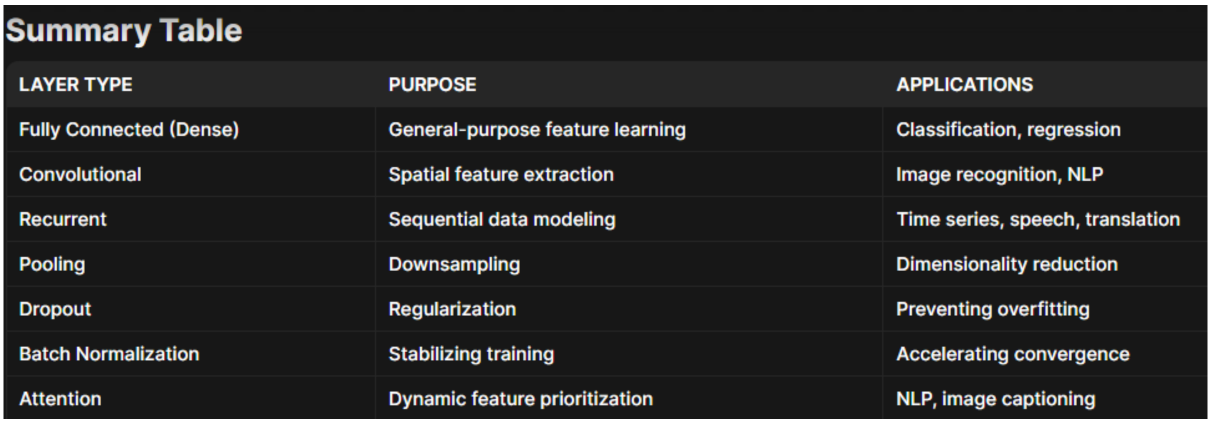
A **Pooling Layer** is used to reduce the spatial dimensions of the data, thereby decreasing the computational load and controlling overfitting. Common types of pooling include Max Pooling and Average Pooling.

Use Cases: Dimensionality reduction in CNNs

6. Batch Normalization Layer

A **Batch Normalization Layer** normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. This helps in accelerating the training process and improving the performance of the network.

Use Cases: Stabilizing and speeding up training



Summary Table

LAYER TYPE	PURPOSE	APPLICATIONS
Fully Connected (Dense)	General-purpose feature learning	Classification, regression
Convolutional	Spatial feature extraction	Image recognition, NLP
Recurrent	Sequential data modeling	Time series, speech, translation
Pooling	Downsampling	Dimensionality reduction
Dropout	Regularization	Preventing overfitting
Batch Normalization	Stabilizing training	Accelerating convergence
Attention	Dynamic feature prioritization	NLP, image captioning

Activation Function

In artificial neural networks, an activation function is one that outputs a smaller value for tiny inputs and a higher value if its inputs are greater than a threshold. An activation function "fires" if the inputs are big enough; otherwise, nothing happens. An activation function, then, is a gate that verifies how an incoming value is higher than a threshold value.

Because they introduce non-linearities in neural networks and enable the neural networks can learn powerful operations, activation functions are helpful. A feedforward neural network might be refactored into a straightforward linear function or matrix transformation on to its input if indeed the activation functions were taken out.

By generating a weighted total and then including bias with it, the activation function determines whether a neuron should be turned on. The activation function seeks to boost a neuron's output's nonlinearity.

Need of Non-linear Activation Functions

An interconnected regression model without an activation function is all that a neural network is. Input is transformed nonlinearly by the activation function, allowing the system to learn and perform more challenging tasks.

It is merely a thing procedure that is used to obtain a node's output. It also goes by the name Transfer Function.

The mixture of two linear functions yields a linear function, so no matter how several hidden layers we add to a neural network, they all will behave in the same way. The neuron cannot learn if all it has is a linear model. It will be able to learn based on the difference with respect to error with a non-linear activation function.

The mixture of two linear functions yields a linear function in itself, so no matter how several hidden layers we add to a neural network, they all will behave in the same way. The neuron cannot learn if all it has is a linear model.

Linear Activation Function

As can be observed, the functional is linear or linear. Therefore, no region will be employed to restrict the functions' output.

Activation Function

- ### Linear Function

Equation: A linear function's equation, which is y = x, is similar to the eqn of a single direction.

The ultimate activation function of the last layer is nothing more than a linear function of input from the first layer, regardless of how many levels we have if they are all linear in nature. -inf to +inf is the range.

Uses: The output layer is the only location where the activation function's function is applied.

If we separate a linear function to add non-linearity, the outcome will no longer depend on the input "x," the function will become fixed, and our algorithm won't exhibit any novel behaviour.

A good example of a regression problem is determining the cost of a house. We can use linear activation at the output layer since the price of a house may have any huge or little value. The neural network's hidden layers must perform some sort of non-linear function even in this circumstance.

- ### Sigmoid Function

It is a functional that is graphed in a "S" shape.

A is equal to 1/(1 + e-x).

Non-linear in nature. Observe that while Y values are fairly steep, X values range from -2 to 2. To put it another way, small changes in x also would cause significant shifts in the value of Y. spans from 0 to 1.

Uses: Sigmoid function is typically employed in the output nodes of a classi?cation, where the result may only be either 0 or 1. Since the value for the sigmoid function only ranges from 0 to 1, the result can be easily anticipated to be 1 if the value is more than 0.5 and 0 if it is not.

- ### Tanh Function

The activation that consistently outperforms sigmoid function is known as tangent hyperbolic function. It's actually a sigmoid function that has been mathematically adjusted. Both are comparable to and derivable from one another.

$$f(x) = \tanh(x) = 2/(1 + e^{-2x}) - 1$$

OR

$$\tanh(x) = 2 * \text{sigmoid}(2x) - 1$$

Range of values: -1 to +1. non-linear nature

Uses: - Since its values typically range from -1 to 1, the mean again for hidden layer of a neural network will be 0 or very near to it. This helps to centre the data by getting the mean close to 0. This greatly facilitates learning for the following layer.

Equation:

max A(x) (0, x). If x is positive, it outputs x; if not, it outputs 0.

Value Interval: [0, inf]

Nature: non-linear, which allows us to simply backpropagate the mistakes and have the ReLU function activate many layers of neurons.

Uses: Because ReLu includes simpler mathematical processes than tanh and sigmoid, it requires less computer time to run. The system is sparse and efficient for computation since only a limited number of neurons are activated at any given time.

Simply said, RELU picks up information considerably more quickly than sigmoid and Tanh functions.

ReLU (Rectified Linear Unit) Activation Function

Currently, the ReLU is the activation function that is employed the most globally. Since practically all convolutional neural networks and deep learning systems employ it.

The derivative and the function are both monotonic.

However, the problem is that all negative values instantly become zero, which reduces the model's capacity to effectively fit or learn from the data. This means that any negative input to a ReLU activation function immediately becomes zero in the graph, which has an impact on the final graph by improperly mapping the negative values.

- ### Softmax Function

Although it is a subclass of the sigmoid function, the softmax function comes in handy when dealing with multiclass classification issues.

Used frequently when managing several classes. In the output nodes of image classification issues, the softmax was typically present. The softmax function would split by the sum of the outputs and squeeze all outputs for each category between 0 and

The output unit of the classifier, where we are actually attempting to obtain the probabilities to determine the class of each input, is where the softmax function is best applied.

The usual rule of thumb is to utilise RELU, which is a usual perceptron in hidden layers and is employed in the majority of cases these days, if we really are unsure of what encoder to apply.

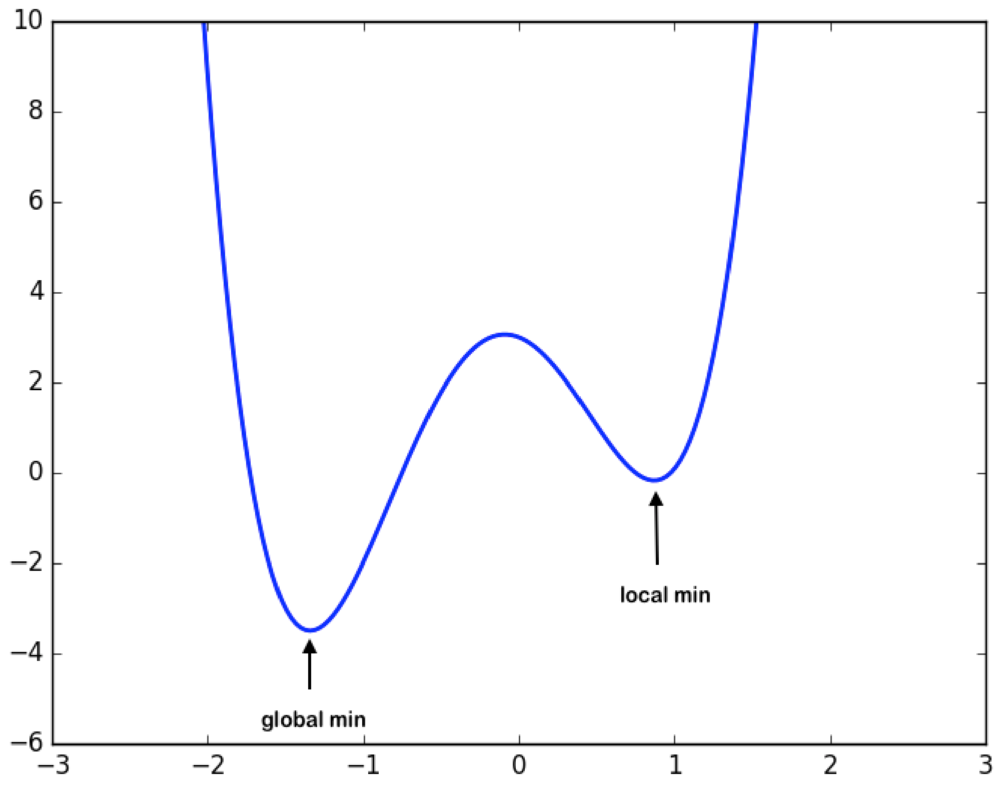
A very logical choice for the output layer is the sigmoid function if your input is for binary classification. If our output involves multiple classes, Softmax can be quite helpful in predicting the odds for each class.

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.



This shows the global and local minima with the loss on Y-axis and weights on X-axis

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*.

$$\underset{\text{New weight}}{{}^{*}W_x} = \overset{\text{Old weight}}{W_x} - \underset{\substack{\text{Learning} \\ \text{rate}}}{a} \left(\overset{\substack{\text{Derivative of Error} \\ \text{with respect to weight}}}{\frac{\partial Error}{\partial W_x}} \right)$$

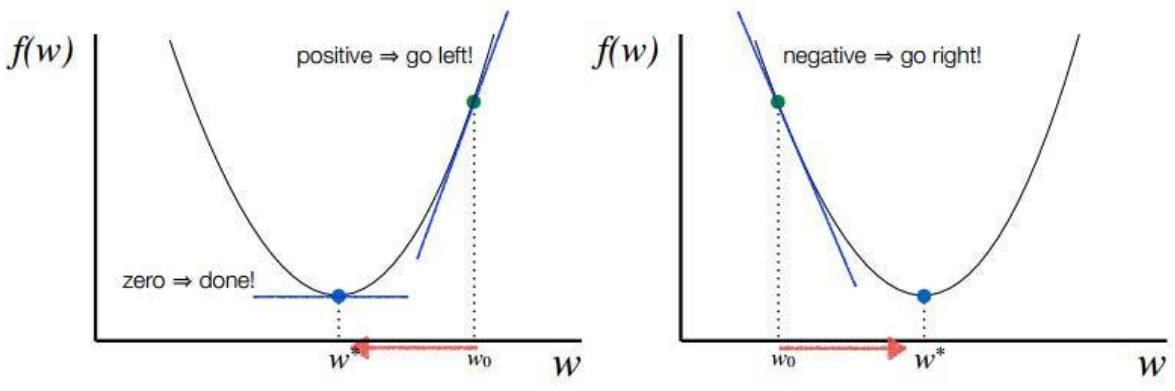
TYPES OF OPTIMIZERS :

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.

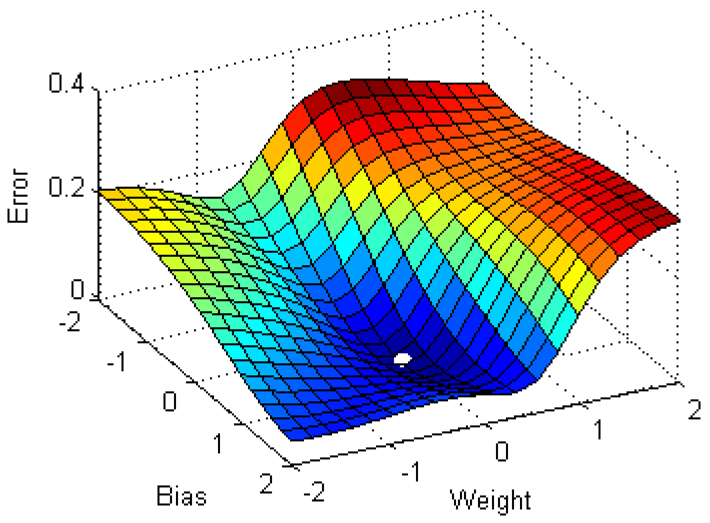


Stochastic Gradient Descent :

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.

However, if we choose a learning rate that is too small, it may lead to very slow convergence, while a larger learning rate can make it difficult to converge and cause the cost function to fluctuate around the minimum or even to diverge away from the 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.

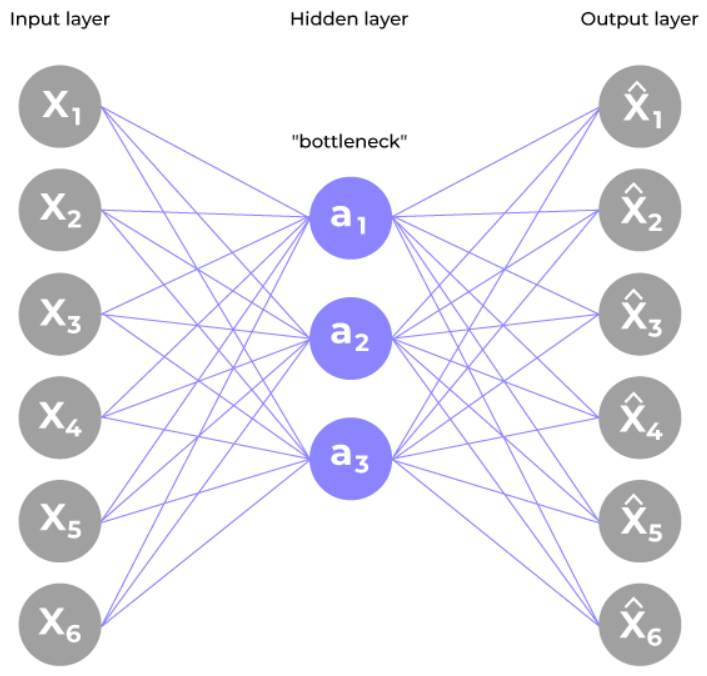
Thus it performs smaller updates(lower learning rates) for the weights corresponding to the high-frequency features and bigger updates(higher learning rates) for the weights corresponding to the low-frequency features, which in turn helps in better performance with higher accuracy. Adagrad is well-suited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

What are Autoencoders?

Autoencoders are a specialized class of algorithms that can learn efficient representations of input data with no need for labels. It is a class of artificial neural networks designed for unsupervised learning. Learning to compress and effectively represent input data without specific labels is the essential principle of an automatic decoder. This is accomplished using a two-fold structure that consists of an encoder and a decoder. The encoder transforms the input data into a reduced-dimensional representation, which is often referred to as “latent space” or “encoding”. From that representation, a decoder rebuilds the initial input. For the network to gain meaningful patterns in data, a process of encoding and decoding facilitates the definition of essential features.

The general architecture of an autoencoder includes an encoder, decoder, and bottleneck layer.



1. Encoder
    - Input layer take raw input data
    - The hidden layers progressively reduce the dimensionality of the input, capturing important features and patterns. These layer compose the encoder.
    - The bottleneck layer (latent space) is the final hidden layer, where the dimensionality is significantly reduced. This layer represents the compressed encoding of the input data.
2. Decoder
    - The bottleneck layer takes the encoded representation and expands it back to the dimensionality of the original input.
    - The hidden layers progressively increase the dimensionality and aim to reconstruct the original input.
    - The output layer produces the reconstructed output, which ideally should be as close as possible to the input data.
3. The loss function used during training is typically a reconstruction loss, measuring the difference between the input and the reconstructed output. Common choices include mean squared error (MSE) for continuous data or binary cross-entropy for binary data.
4. During training, the autoencoder learns to minimize the reconstruction loss, forcing the network to capture the most important features of the input data in the bottleneck layer.

After the training process, only the encoder part of the autoencoder is retained to encode a similar type of data used in the training process. The different ways to constrain the network are: –

- **Keep small Hidden Layers:** If the size of each hidden layer is kept as small as possible, then the network will be forced to pick up only the representative features of the data thus encoding the data.
- **Regularization:** In this method, a loss term is added to the cost function which encourages the network to train in ways other than copying the input.
- **Denoising:** Another way of constraining the network is to add noise to the input and teach the network how to remove the noise from the data.
- **Tuning the Activation Functions:** This method involves changing the activation functions of various nodes so that a majority of the nodes are dormant thus, effectively reducing the size of the hidden layers.

Types of Autoencoders

There are diverse types of autoencoders and analyze the advantages and disadvantages associated with different variation:

Denoising Autoencoder

Denoising autoencoder works on a partially corrupted input and trains to recover the original undistorted image. As mentioned above, this method is an effective way to constrain the network from simply copying the input and thus learn the underlying structure and important features of the data.

Sparse Autoencoder

This type of autoencoder typically contains more hidden units than the input but only a few are allowed to be active at once. This property is called the sparsity of the network. The sparsity of the network can be controlled by either manually zeroing the required hidden units, tuning the activation functions or by adding a loss term to the cost function.

Variational Autoencoder

Variational autoencoder makes strong assumptions about the distribution of latent variables and uses the **Stochastic Gradient Variational Bayes** estimator in the training process. It assumes that the data is generated by a **Directed Graphical Model** and tries to learn an approximation to qϕ(z|x) $q\phi(z|x)$ to the conditional property qθ(z|x) $q\theta(z|x)$ where ϕ ϕ and θ θ are the parameters of the encoder and the decoder respectively.

Convolutional Autoencoder

Convolutional autoencoders are a type of autoencoder that use convolutional neural networks (CNNs) as their building blocks. The encoder consists of multiple layers that take a image or a grid as input and pass it through different convolution layers thus forming a compressed representation of the input. The decoder is the mirror image of the encoder it deconvolves the compressed representation and tries to reconstruct the original image.

Applications of Vanilla Autoencoders:

Vanilla Autoencoders find applications across a wide spectrum of domains, owing to their versatility and simplicity. Some notable applications include:

- Data Compression: By learning a compressed representation of the input data, Vanilla Autoencoders excel in data compression tasks, facilitating efficient storage and transmission of information.
- Feature Learning: Vanilla Autoencoders are adept at capturing salient features of the input data, making them valuable for feature learning tasks in domains such as computer vision, natural language processing, and sensor data analysis.