Activation Functions Demystified

om pramod · Follow

10 min read · Aug 15, 2023

Part 2: Classification of Activation Functions

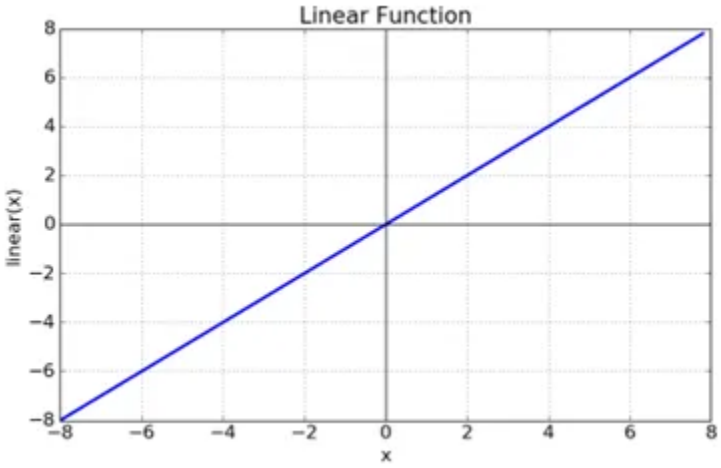
The Activation Functions can be basically divided into 2 types

1. Linear Activation Function

2. Non-linear Activation Functions

Linear Activation Function:

The linear activation function, also known as “identity” (multiplied by 1), simply returns the input as the output. It is also called as “no activation”, because the linear activation function does not change the weighted sum of the input in any way and instead returns the value directly. A linear function is also known as a straight-line function where the activation is proportional to the input i.e. the weighted sum from neurons.



It’s represented mathematically as f(x) = x

Range: (-infinity to infinity)

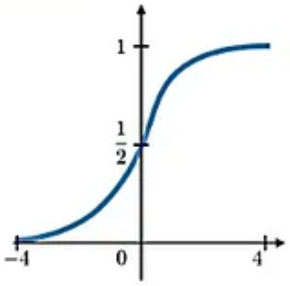
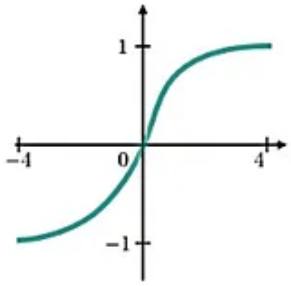
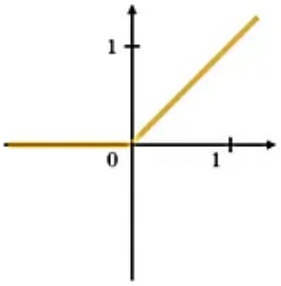
However, it's true that neural network with multiple hidden layers can learn non-linear relationships through the combination of multiple linear activation functions, it might not be able to model complex non-linear relationships in the data. It is not commonly used in modern deep learning models because it's a relatively weak activation function.

It's used in neural networks when a linear relationship is assumed between the inputs and outputs. It's often used as the activation function in the output layer of a regression problem, where a direct prediction is required.

Here is an example of how to use a linear activation function in a neural network using Python and the Keras library:

```
from keras.models import Sequential
from keras.layers import Dense

# create a sequential model
model = Sequential()

# add a fully connected dense layer with 128 neurons and a linear activation
model.add(Dense(128, activation='linear', input_shape=(input_shape, )))

# add a final output layer with a single neuron and a linear activation funct
model.add(Dense(1, activation='linear'))

# compile the model with a loss function and an optimizer
model.compile(loss='mean_squared_error', optimizer='adam')

# fit the model on the training data
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_val,
```

Non-linear Activation Functions:

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$g(z) = \max(0, z)$
		

Here ‘z’ is the input to the activation function. Reference

Sigmoid / Logistic Activation Function:

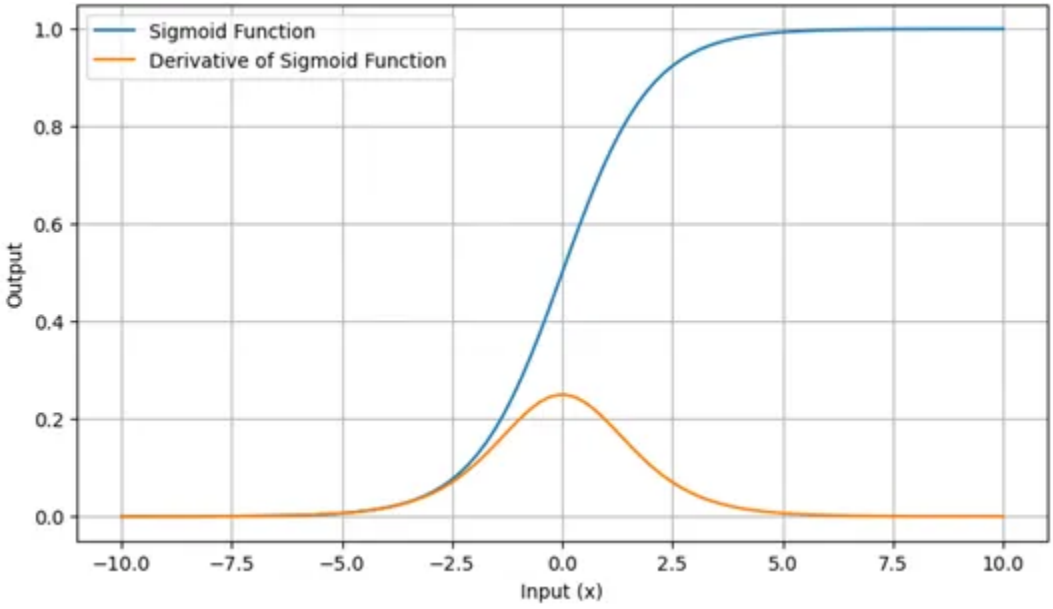
This function takes any real value as input and outputs values in the range of 0 to 1. The larger the input (more positive), the closer the output value will be to 1, whereas the smaller the input (more negative), the closer the output will be to 0, as the value of the exponential term becomes very small for large positive values of z, and becomes very large for large negative values of z.

Usually used in the output layer of a binary classification, where the result is either 0 or 1, as the value for the sigmoid function lies between 0 and 1 only so, the result can be predicted easily to be 1 if the value is greater than 0.5 and 0 otherwise (This threshold can be adjusted based on the requirements of the specific problem)

Disadvantage — Vanishing gradient problem:

The vanishing gradient problem is a significant disadvantage associated with the sigmoid activation function in artificial neural networks. Neural networks are trained using optimization algorithms, with the most common one being gradient descent. During training, the algorithm adjusts the network’s weights to minimize the error between predicted and actual output. This adjustment is guided by the gradient of the loss function with respect to the network’s weights. The gradient is computed using a process called backpropagation, where the chain rule is applied to compute the derivative of the loss with respect to each weight.

The derivative of the sigmoid function σ’(x) is given by σ(x) * (1 — σ(x))



The sigmoid function takes any real number as input and maps it to a value between 0 and 1. As you can see in the plot, when the input x is very negative (far left), the sigmoid function outputs a value close to 0. When the input x is very positive (far right), the sigmoid function outputs a value close to 1. This plot shows that the derivatives of the sigmoid function are at most 0.25. Notice that the derivative is highest around the point where the sigmoid function's output is approximately 0.5. This is where the slope of the sigmoid function is steepest. As you move away from this point in either direction (towards 0 or 1), the derivative decreases. This means that when the sigmoid function is applied to a large input, the derivative will be very small. As a result, the gradient of the error function with respect to the weight will also be very small, and the weight will not be updated very much.

This behaviour of the sigmoid function and its derivative leads to the so-called "vanishing gradient" problem. During the training process of a neural network, the weights are updated proportional to the derivative of the activation function. When the derivative is close to 0, the updates to the weights become very small, effectively slowing down the learning process or causing it to stop altogether.

Hyperbolic Tangent (tanh) Activation function:

It is similar to the sigmoid function, but it maps the input of a neuron to an output range of -1 to 1. In Tanh, the larger the input (more positive), the closer the output

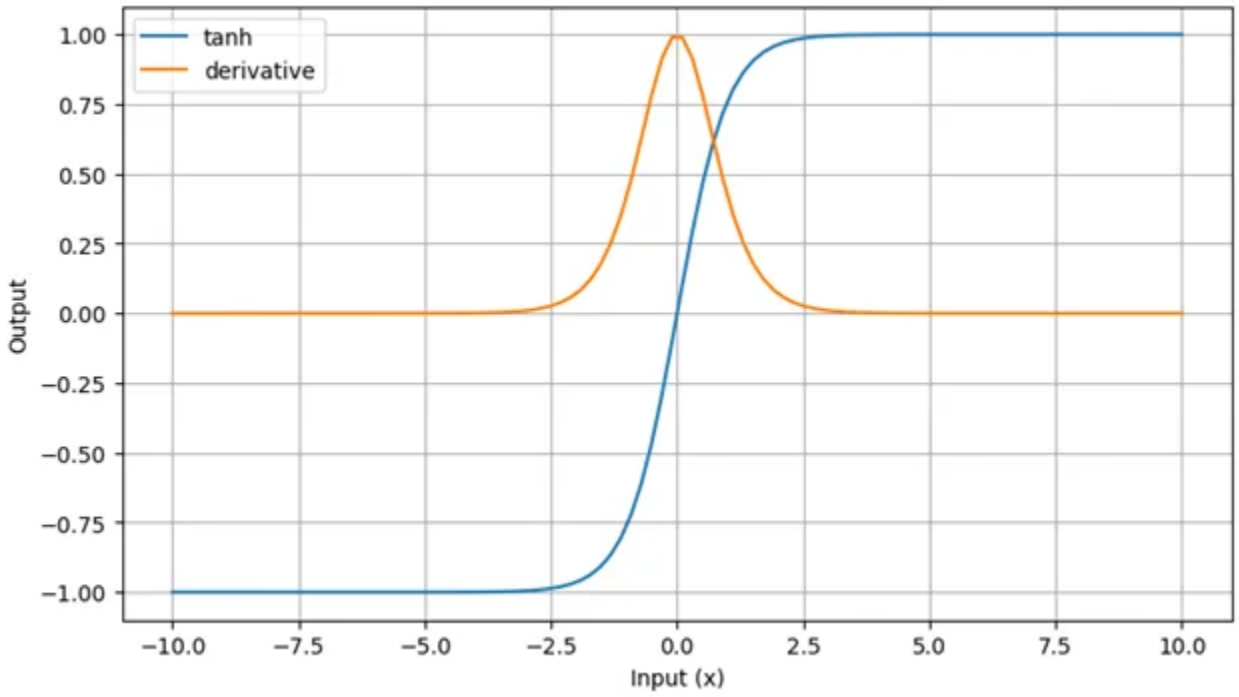
value will be to 1, whereas the smaller the input (more negative), the closer the output will be to -1.

Usually used in hidden layers of a neural network as it's values lies between -1 to 1 hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in centering the data by bringing mean close to 0. This makes learning for the next layer much easier. When the input data to a hidden layer is well-centered, it helps to prevent the weights from becoming too large or too small, which can cause the gradients to explode or vanish during training. This can make the training process more stable and faster.

Note — When using the TanH function for hidden layers, it is a good practice to use a "Xavier Normal" or "Xavier Uniform" weight initialization (also referred to Glorot initialization, named for Xavier Glorot) and scale input data to the range -1 to 1 (e.g. the range of the activation function) prior to training.

Disadvantage — Vanishing gradient problem:

The vanishing gradient problem is not only associated with the sigmoid activation function but also extends to the hyperbolic tangent (tanh) activation function. The derivative of the tanh function is given by $1 - \tanh(x)^2$. The tanh function's output is between -1 and 1, and its derivatives are in the range (0, 1], which means that during backpropagation, the gradients can often become very small likewise sigmoid function.



This plot shows that the derivatives of the tanh function are at most 1. You can observe that the derivative of the tanh function approaches 0 as the input moves towards very positive or very negative values, similar to what was explained earlier.

ReLU (Rectified linear unit) Activation function:

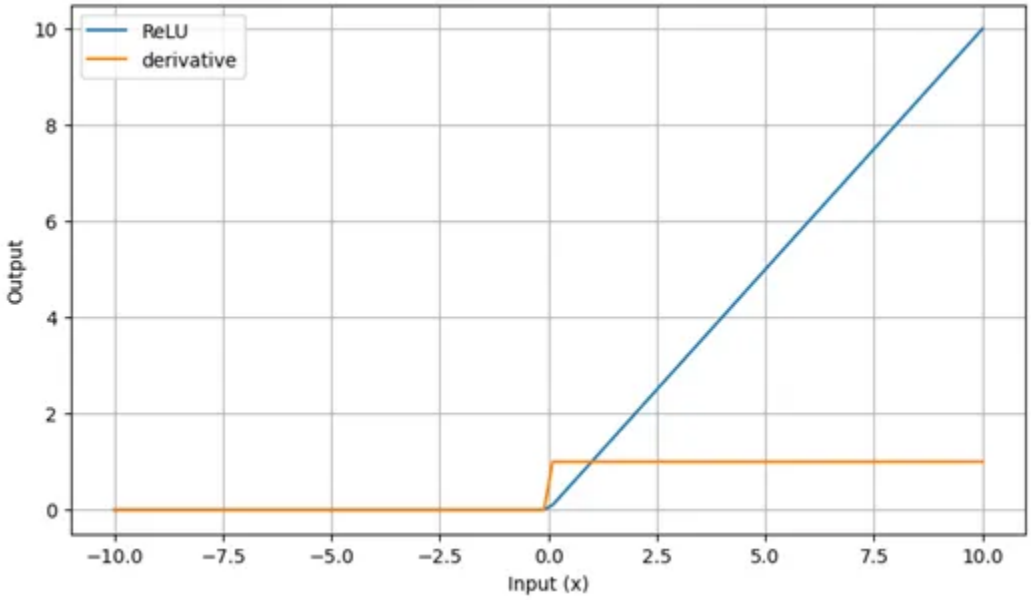
It maps inputs to outputs in the range of [0, +∞), where for inputs z >= 0, the output of the ReLU activation function is equal to z, and for inputs z < 0, the output is 0(In simple terms, It gives an output z if z is positive and 0 otherwise)

Note — ReLU is most commonly used in the hidden layers of a deep neural network.

Advantage — ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation(This sparsity also has the benefit of preventing overfitting, as it reduces the number of parameters that need to be learned). In simple words, RELU learns much faster than sigmoid and Tanh function.

Advantage — The ReLU function helps to overcome the vanishing gradient problem, a common issue in deep networks, where the gradients of the

parameters become very small during backpropagation, making it difficult to update the weights. The derivative of the ReLU function is either 0 (for x<0) or 1 (for x>=0). In other words, for positive input values, the derivative is 1, which means that gradients do not vanish. This is in contrast to sigmoid and tanh functions, which tend to approach zero as inputs become very positive.



Notice that the derivative of the ReLU function is 1 for positive inputs (left side of the graph), indicating that the gradient remains constant and non-vanishing. This property of ReLU helps to address the vanishing gradient problem, making it easier to train deep neural networks effectively.

Disadvantage: Despite its advantage in mitigating the vanishing gradient problem, ReLU does have a potential drawback known as the “dying ReLU” problem. This issue arises due to the zero gradient for negative inputs in the ReLU activation function. If a neuron’s weights get updated such that the weighted sum of the neuron’s inputs is negative, the neuron will keep outputting 0 and stop contributing to the learning process. This is because the gradient of ReLU is 0 for negative inputs, so the weights of the neuron do not get updated during the backpropagation process.

To overcome this issue, variants of the ReLU activation function, such as the leaky ReLU or the parametric ReLU have been proposed.

Softmax Activation function:

The softmax function is commonly found in the output layer of multi- class classification problems.

the sigmoid activation function, that works on calculating probability values is building block of softmax function. The output of the sigmoid function was in the range of 0 to 1, which can be thought of as probability of the positive class. But this function faces certain problems.

Let's suppose we have five output values of 0.8, 0.9, 0.7, 0.8, and 0.6, respectively. The above values don't make sense as the sum of all the classes/output probabilities should be equal to 1.

You see, the Softmax function is described as a combination of multiple sigmoids.

So Sigmoid activation function is commonly used in binary classification problems, where the output of the network represents the probability of a single class. In this case, the sigmoid activation function maps the input values to a value between 0 and 1, which can be interpreted as a probability of the positive class. However, in multi-class classification problems, where there are more than two classes, the Sigmoid activation function is not suitable as it only provides a probability for a single class. In these cases, the Softmax activation function is commonly used as it maps the input values to a probability distribution over all classes.

Example — Assume that you have three classes, meaning that there would be three neurons in the output layer. Now, suppose that your output from the neurons is [1.8, 0.9, 0.68].

Applying the softmax function over these values to give a probabilistic view will result in the following outcome: [0.58, 0.23, 0.19].

The function returns 1 for the largest probability index while it returns 0 for the other two array indexes. Here, giving full weight to index 0 and no weight to index 1 and index 2. So the output would be the class corresponding to the 1st neuron(index 0) out of three.

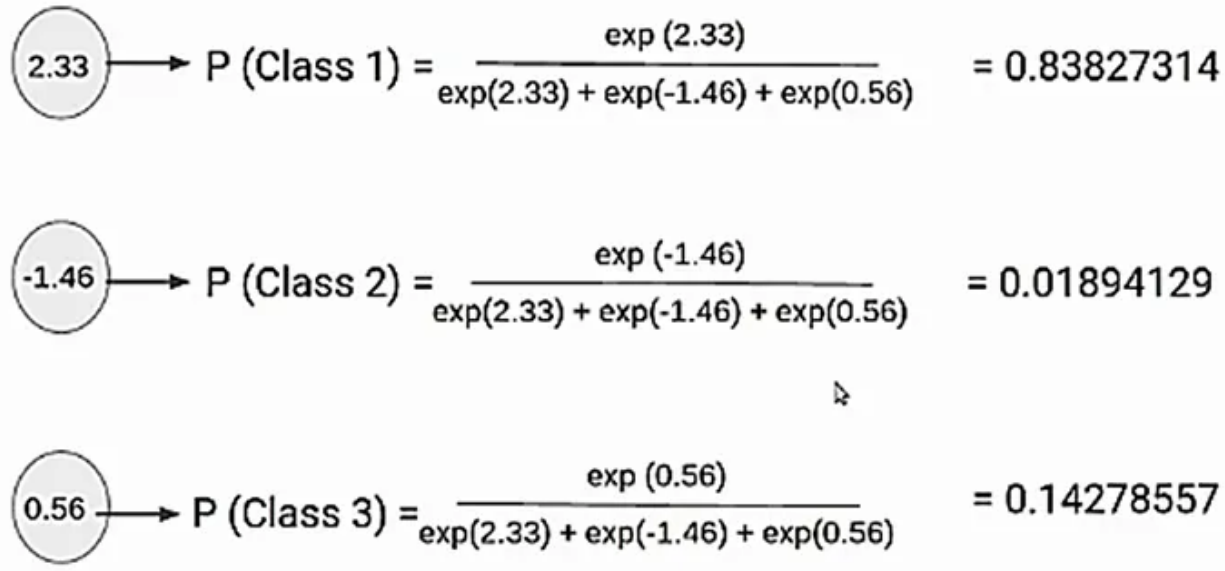
$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Where, x_i : This is the input to the activation function for class i.

$\sum_j \exp(x_j)$: This is the sum of the exponentials of all inputs for all j classes

$\text{Softmax}(x_i)$: This is the output of the Softmax activation function for class i. It represents the probability of class i, given the input x.

Example :



In this case, it is clear that the input belongs to class 1.

Python implementation:

Using activation functions in Python is straightforward. You can implement activation functions as standalone functions or use libraries like NumPy or TensorFlow that provide built-in activation functions. Here's how you can use both approaches:

Using Standalone Functions:

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def tanh(x):
```

```
    return np.tanh(x)
def relu(x):
    return np.maximum(0, x)
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)

# Example usage
input_data = np.array([0.5, -0.7, 1.2, -2.0])

sigmoid_output = sigmoid(input_data)
relu_output = relu(input_data)
tanh_output = tanh(input_data)
softmax_output = softmax(input_data)

print("Sigmoid Output:", sigmoid_output)
print("ReLU Output:", relu_output)
print("Tanh Output:", tanh_output)
print("Softmax Output:", softmax_output)
```

The output looks like this-

```
Sigmoid Output: [0.62245933 0.33181223 0.76852478 0.11920292]
ReLU Output: [0.5 0.  1.2 0. ]
Tanh Output: [ 0.46211716 -0.60436778  0.83365461 -0.96402758]
Softmax Output: [0.29437463 0.08866393 0.5927977  0.02416374]
```

Using TensorFlow:

These activation functions can be used in a neural network model through the activation argument supported by all forward layers in Keras, a popular deep learning library in Python. For example:

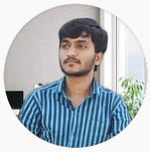
```
from tensorflow.keras import layers
model = Sequential()
model.add(layers.Dense(64, activation='relu'))
```

In this case, ‘relu’ is the string identifier for the ReLU activation function. You can replace ‘relu’ with ‘sigmoid’, ‘tanh’, or ‘softmax’ to use those activation functions.

Final Note: It’s a Wrap, but the Learning Journey Continues! Stay ‘Activated’ and Keep Exploring!

Did you find this article informative and engaging? If so, go ahead, give that clap button a friendly tap, and let’s spread the knowledge together! Your claps mean a lot and help others discover this content.

Deep Learning | Machine Learning | Artificial Intelligence | Activation Functions | Data Science

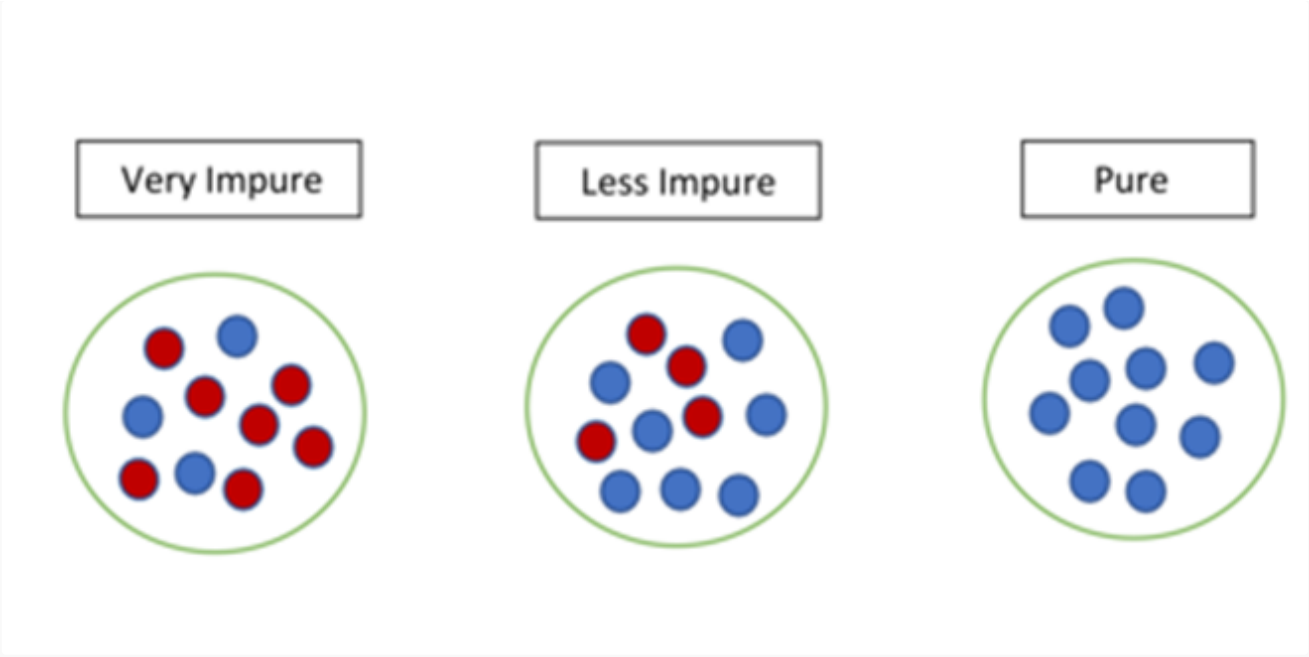


Written by om pramod

181 Followers

AIML enthusiast

More from om pramod

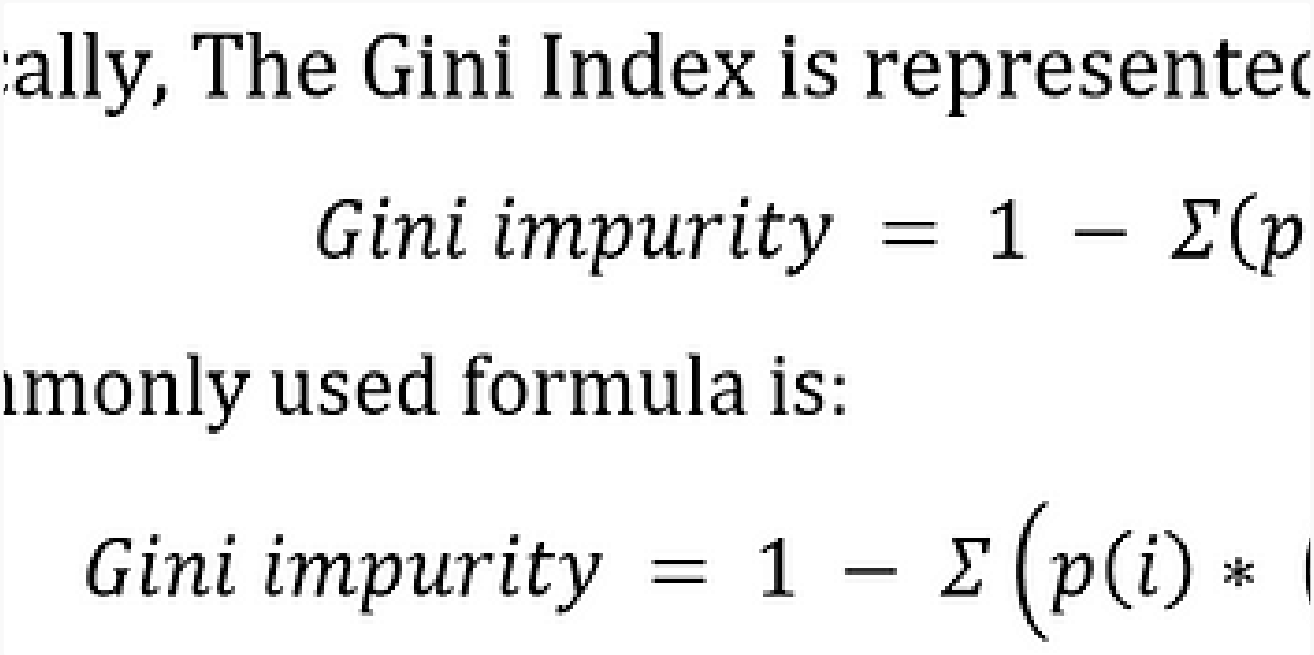


om pramod

Decision Trees

Part 2: Information Gain

Jan 29, 2023 850 3



om pramod

Decision Trees

Part 4: Gini Index

Jan 29, 2023 767 1

- Learning rate in optimization algorithms (e.g. gradient descent)
- Choice of optimization algorithm (e.g., gradient descent, stochastic gradient descent, or Adam optimizer)
- Choice of activation function in a neural network (nn) layer (e.g. Sigmoid, ReLU, Tanh)
- The choice of cost or loss function the model will use
- Number of hidden layers in a nn
- Number of activation units in each layer
- The drop-out rate in nn (dropout probability)
- Number of iterations (epochs) in training a nn
- Number of clusters in a clustering task
- Kernel or filter size in convolutional layers

om pramod

Model Parameters and Hyperparameters in machine learning

In machine learning, parameters and hyperparameters refer to different types of variables that are used to control the behaviour of the...

Jan 14, 2023 658

P(Oi | q1,..qT,O1,...OT) = P(Oi |qi)

Here's what this means:

- `P(Oi | q1,..qT,O1,...OT)` is the joint probability of observing `Oi` given the states `q1,..qT` and the observations `O1,...OT`.
- `P(Oi |qi)` is the probability of observing `Oi` given the state `qi`.

The equation says that the probability of observing `Oi` depends only on the state `qi` that produced the observation `Oi`, and not on any other states or observations. This is a fundamental assumption in HMMs and is what allows them to model systems where the current state affects the current observation but past states and observations do not

om pramod

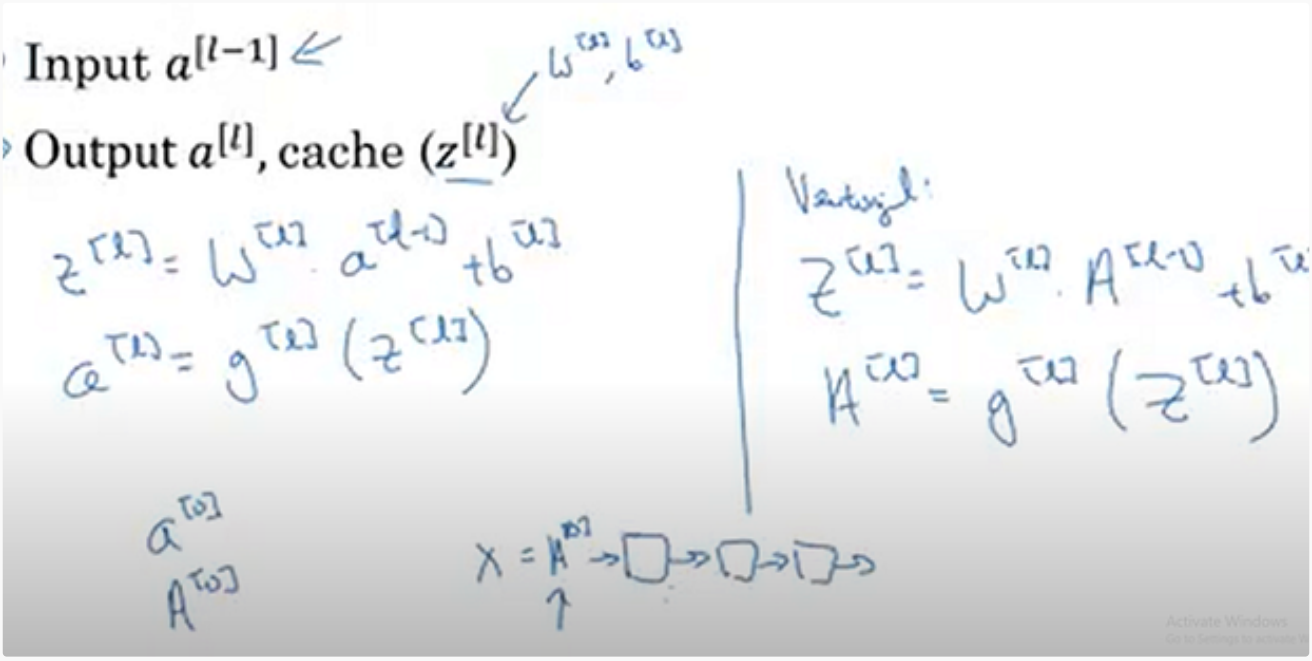
Hidden Markov Models: The Secret Sauce in Natural Language Processing

Part 1: Understanding the Basics: Terminologies and Architecture of Hidden Markov Models

Dec 30, 2023 627

See all from om pramod

Recommended from Medium



Coursesteach

Deep Learning (Part 34)-Forward and Backward Propagation

Chapter5:Deep Neural Network

Apr 8 5

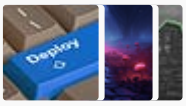
Irene Markelic

Mastering the Softmax Function: Understanding its Derivative with a Step-by-Step Example

This article focuses on obtaining the derivative of the softmax function by means of a simple example. It assumes that the reader is...

Apr 1 10 1

Lists



Predictive Modeling w/ Python

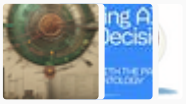
20 stories · 1498 saves

Natural Language Processing

1681 stories · 1255 saves

Practical Guides to Machine Learning

10 stories · 1827 saves



data science and AI

40 stories · 234 saves

Sanjay Dutta

Understanding Glorot and He Initialization: A Guide for Students

When training deep neural networks, the initialization of the weights can significantly affect how well and how quickly the model learns...

Jun 9 9

Neeraj Nayan

Binary Cross Entropy — Machine Learning

Binary cross-entropy (BCE), is a loss function commonly used in binary classification tasks, particularly in machine learning algorithms...

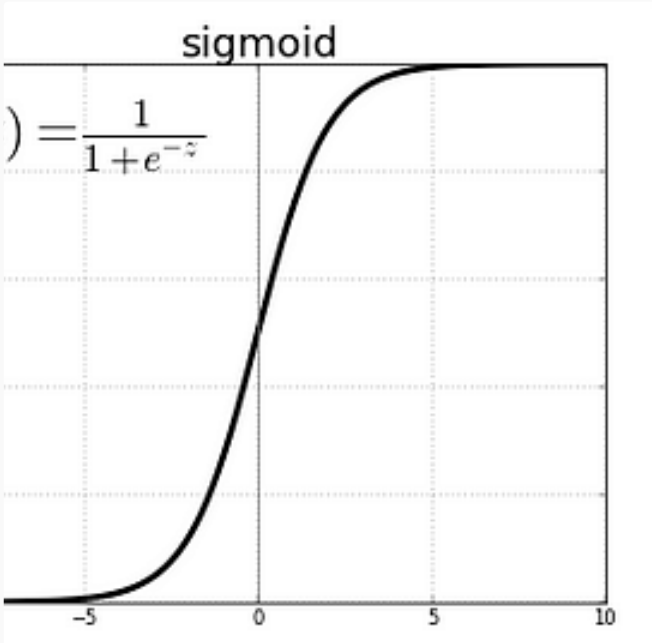
Apr 24

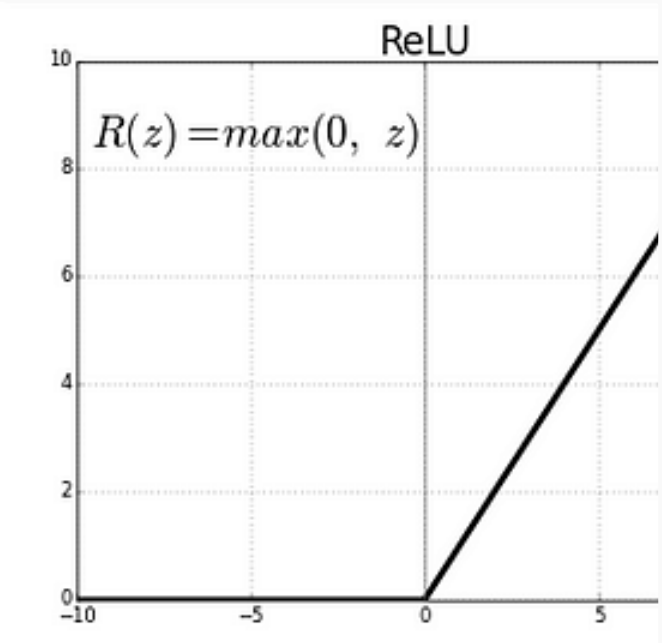
om pramod

Autoencoders Explained

Part 3: KL-divergence

Jun 16 350





Shivansh Srivastava

Understanding the Difference Between ReLU and Sigmoid Activation Functions in Deep Learning

In the realm of deep learning, activation functions play a crucial role in shaping the behavior and performance of neural networks. Two...

Apr 18

See more recommendations