NEURAL NETWORKS

PERCEPTRON MODEL:

A perceptron model, in Machine Learning, is a supervised learning algorithm of binary classifiers. A single neuron, the perceptron model detects whether any function is an input or not and classifies them in either of the classes. Representing a biological neuron in the human brain, the perceptron model or simply a perceptron acts as an artificial neuron that performs human-like brain functions. A linear ML algorithm, the perceptron conducts binary classification or two-class categorization and enables neurons to learn and register information procured from the inputs.

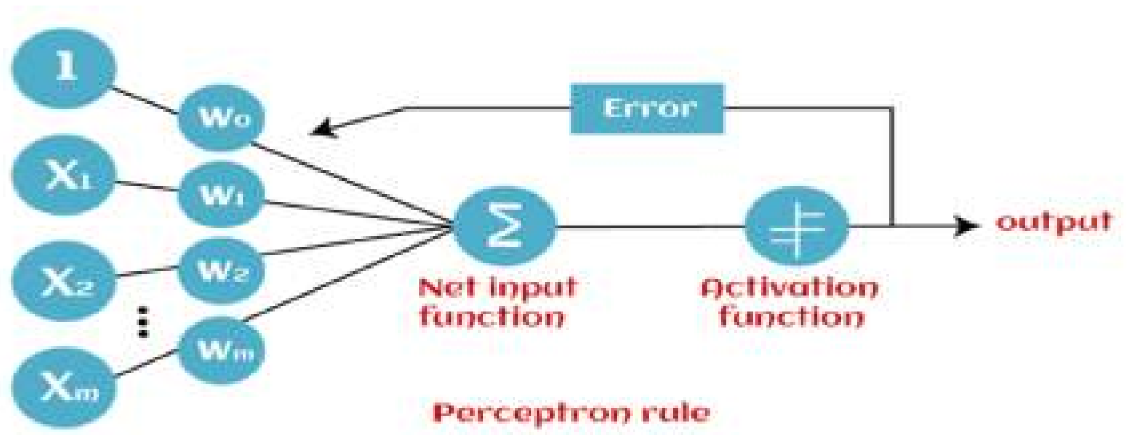
This model uses a hyperplane line that classifies two inputs and classifies them on the basis of the 2 classes that a machine learns, thus implying that the perceptron model is a linear classification model.
There are 4 constituents of a perceptron model. They are as follows-

1. Input values
2. Weights and bias
3. Net sum
4. Activation function

There are 2 types of perceptron models-

1. Single Layer Perceptron- The Single Layer perceptron is defined by its ability to linearly classify inputs. This means that this kind of model only utilizes a single hyperplane line and classifies the inputs as per the learned weights beforehand.
2. Multi-Layer Perceptron- The Multi-Layer Perceptron is defined by its ability to use layers while classifying inputs. This type is a high processing algorithm that allows machines to classify inputs using various more than one layer at the same time.

In Machine Learning, Perceptron is considered as a single-layer neural network that consists of four main parameters named input values (Input nodes), weights and Bias, net sum, and an activation function. The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the step function and is represented by 'f'.



This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (-1,1). It is important to note that the weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.

Perceptron model works in two important steps as follows:

Step-1

In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

$$\sum wi*xi = x1*w1 + x2*w2 + \ldots wn*xn$$

Add a special term called bias 'b' to this weighted sum to improve the model's performance.

$$\sum wi*xi + b$$

Step-2

In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:

$$Y = f(\sum wi*xi + b)$$

The activation function can be a step function or it can also be a signum function. After adding the activation function, the perceptron model can be used to complete the binary classification task. The step and the sign functions are discontinuous at z = 0, so the gradient descent algorithm cannot be used to optimize the parameters. In order to enable the perceptron model to automatically learn from the data, Frank Rosenblatt proposed a perceptron learning algorithm, as shown in Algorithm.

Algorithm: Perceptron Training Algorithm:

```
Initialize $w = 0, b = 0$
repeat
Randomly select a sample $(x_i, y_i)$ from training set
Calculate the output $a = sign\ (w^T x_i + b)$
```

If $a \neq y_i$:
$w' \leftarrow w + \boldsymbol{\eta} \cdot y_i \cdot x_i$
$b' \leftarrow b + \boldsymbol{\eta} \cdot y_i$
until you reach the required number of steps

Output: parameters w and b

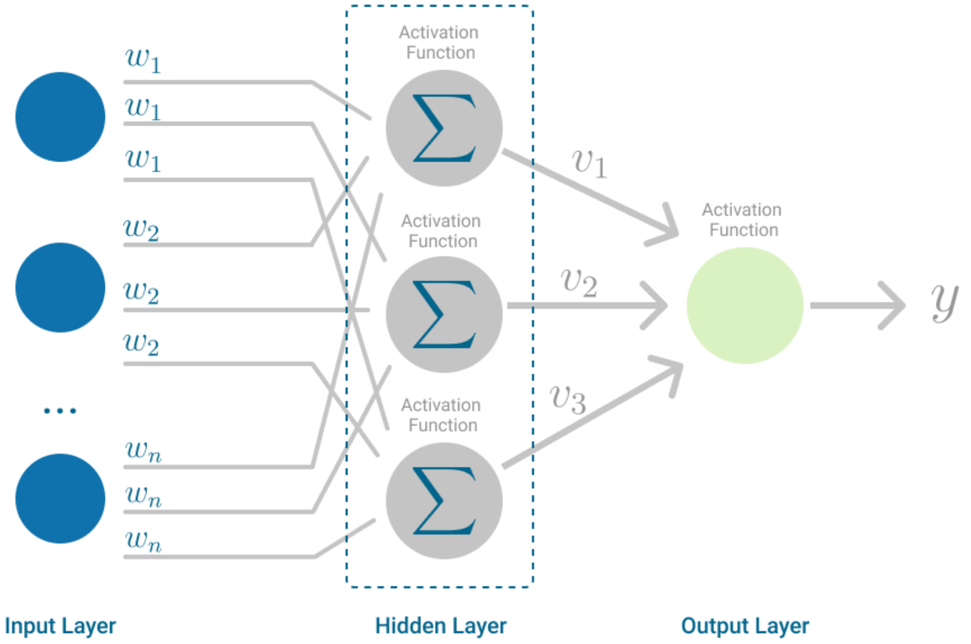
NOTE: Here η is learning rate

Advantages of Perceptron Model over Mc Culloch’s Pitts Model:

- MP Neuron Model only accepts boolean input whereas Perceptron Model can process any real input.
- Inputs aren’t weighted in MP Neuron Model, which makes this model less flexible. On the other hand, Perceptron model can take weights with respective to inputs provided.

MULTILAYER PERCEPTRON:

It is a neural network where the mapping between inputs and output is non-linear.



A Multilayer Perceptron has input and output layers, and one or more hidden layers with many neurons stacked together. And while in the Perceptron the neuron must have an activation function that imposes a threshold, like ReLU or sigmoid, neurons in a Multilayer Perceptron can use any arbitrary activation function.

Using the perceptron model, machines can learn weight coefficients that help them classify inputs. This linear binary classifier is highly effective in arranging and categorizing input data into different classes, allowing probability-based predictions and classifying items into multiple categories. Multilayer Perceptrons have the advantage of learning non-linear models and the ability to train models in real-time (online learning).

FULLY CONNECTED LAYER

The underivable nature of the perceptron model severely constrains its potential, making it only capable of solving extremely simple tasks.

We replace the activation function of the perceptron model and stack multiple neurons in parallel to achieve a multi-input and multi-output network layer structure.

As shown in Figure, two neurons are stacked in parallel, that is, two perceptrons with replaced activation functions, forming a network layer of three input nodes and two output nodes.

The first output node is:

$$o_1 = \sigma(w_{11} \cdot x_1 + w_{21} \cdot x_2 + w_{31} \cdot x_3 + b_1)$$

The output of the second node is:

$$o_2 = \sigma(w_{12} \cdot x_1 + w_{22} \cdot x_2 + w_{32} \cdot x_3 + b_2)$$

Putting them together, the output vector is $o = [o_1, o_2]$. The entire network layer can be expressed by the matrix relationship:

$$[o_1\ o_2\ ] = [x_1\ x_2\ x_3\ ]@[w_{11}\ w_{12}\ w_{21}\ w_{22}\ w_{31}\ w_{32}\ ] + [b_1\ b_2] \qquad (6\text{-}1)$$

That is:

$$O = X@W + b$$

The shape of the input matrix X is defined as [b,d_{in}], while the number of samples is b and the number of input nodes is d_{in}.

The shape of the weight matrix W is defined as [d_{in},d_{out}], while the number of output nodes is d_{out}, and the shape of the offset vector b is [d_{out}].

the output matrix O contains the output of b samples, and the shape is [b,dout].

Since each output node is connected to all input nodes, this network layer is called a fully connected layer, or a dense layer, with W as weight matrix and b is the bias vector.

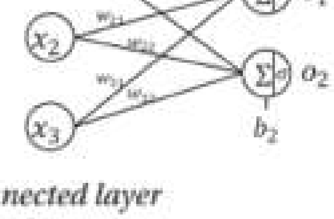


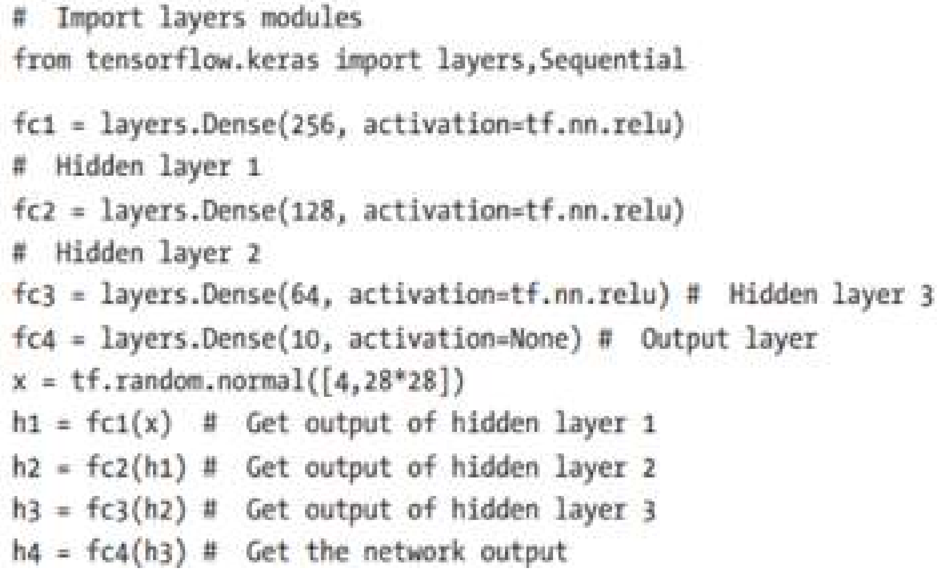
Figure 6-4. *Fully connected layer*

NEURAL NETWORK

- By stacking the fully connected layers in the above figure and ensuring that the number of output nodes of the previous layer matches the number of input nodes of the current layer, a network of any number of layers can be created, which is known as **neural networks.**
- By stacking four fully connected layers, a neural network with four layers can be obtained. Since each layer is a fully connected layer, it is called a **fully connected network**.
- Among them, the first to third fully connected layers are called **hidden layers**, and the output of the last fully connected layer is called the **output layer** of the network.
- When designing a fully connected network, the **hyperparameters** such as the configuration of the network can be set freely according to the rule of thumb, and only a few constraints need to be followed.
- For example, the number of input nodes in the first hidden layer needs to match the actual feature length of the data. The number of input layers in each layer matches the number of output nodes in the previous layer. The activation function and number of nodes in the output layer need to be set according to the specific settings of the required output.
- In general, the design of the neural network models has a greater degree of freedom.

Layer Model Implementation

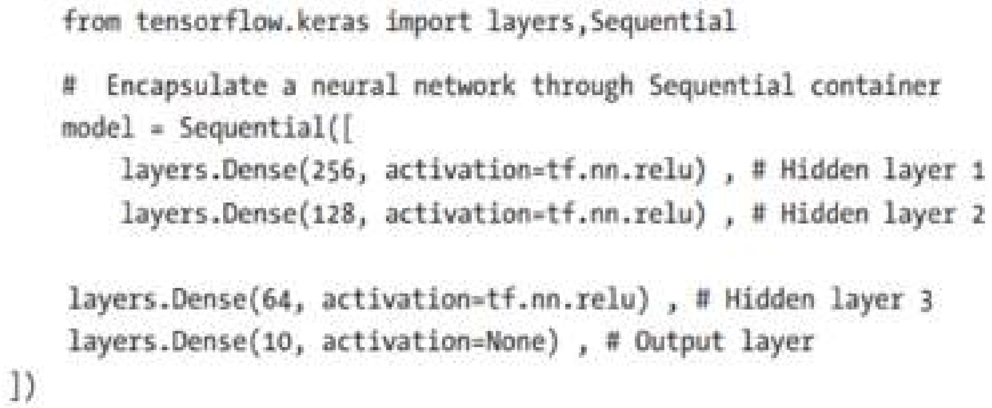
For the conventional network layer, it is more concise and efficient to implement through the layer method. First, create new network layer classes and specify the activation function types of each layer:



```
#  Import layers modules
from tensorflow.keras import layers,Sequential

fc1 = layers.Dense(256, activation=tf.nn.relu)
#  Hidden layer 1
fc2 = layers.Dense(128, activation=tf.nn.relu)
#  Hidden layer 2
fc3 = layers.Dense(64, activation=tf.nn.relu) #  Hidden layer 3
fc4 = layers.Dense(10, activation=None) #  Output layer
x = tf.random.normal([4,28*28])
h1 = fc1(x)  #  Get output of hidden layer 1
h2 = fc2(h1) #  Get output of hidden layer 2
h3 = fc3(h2) #  Get output of hidden layer 3
h4 = fc4(h3) #  Get the network output
```

For such a network where data forwards in turn, it can also be encapsulated into a network class object through the sequential container, and the forward calculation function of the class can be called once to complete the forward calculation of all layers. It is more convenient to use and is implemented as follows :



```
    from tensorflow.keras import layers,Sequential

    #  Encapsulate a neural network through Sequential container
    model = Sequential([
        layers.Dense(256, activation=tf.nn.relu) , # Hidden layer 1
        layers.Dense(128, activation=tf.nn.relu) , # Hidden layer 2

    layers.Dense(64, activation=tf.nn.relu) , # Hidden layer 3
    layers.Dense(10, activation=None) , # Output layer
])
```

In forward calculation, you only need to call the large network objects once to complete the sequential calculation of all layers:

out = model(x)

CASE STUDY:

A. You design a fully connected neural network architecture where all activations are sigmoids. You initialize the weights with large positive numbers. Is this a good idea? Explain your answer.

B. You are doing full batch gradient descent using the entire training set (not stochastic gradient descent). Is it necessary to shuffle the training data? Explain your answer**.**

C. You would like to train a dog/cat image classifier using mini-batch gradient descent. You have already split your dataset into train, dev and test sets. The classes are balanced. You realize that within the training set, the images are ordered in such a way that all the dog images come first and all the cat images come after. A friend tells you: “you absolutely need to shuffle your training set before the training procedure.” Is your friend right? Explain.

SOLU:

Fully connected neural networks (FCNNs) are a type of artificial neural network where the architecture is such that all the nodes, or neurons, in one layer are connected to the neurons in the next layer.

Each individual function consists of a neuron (or a perceptron). In fully connected layers, the neuron applies a linear transformation to the input vector through a weights matrix

$$y_{jk}(x) = f\left(\sum_{i=1}^{n_H} w_{jk}x_i + w_{j0}\right)$$

Where:

xi->Input vector
wjk->weights in the matrix
wj0->Intial Bias

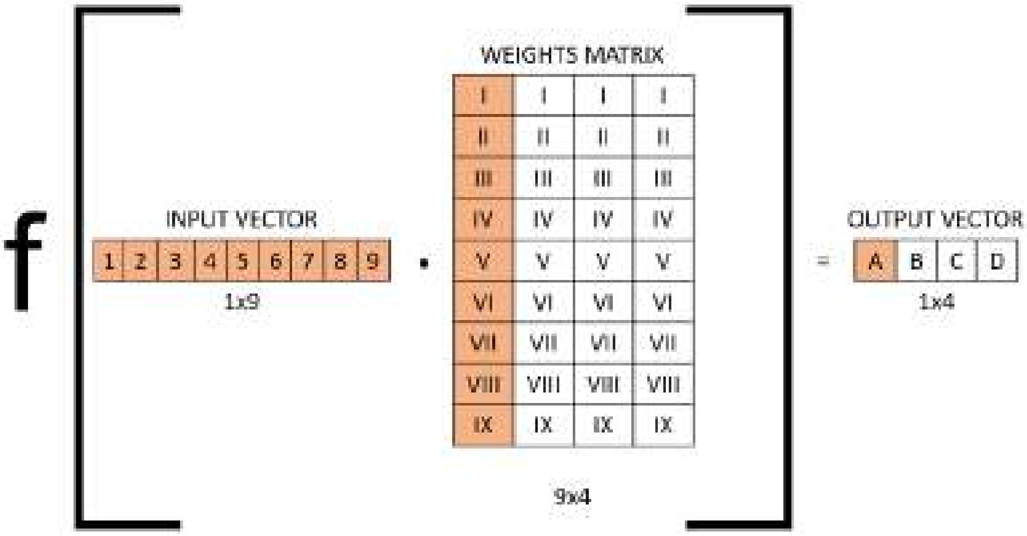
Why are fully connected layers required?

We can divide the whole neural network (for classification) into two parts:

- **Feature extraction**: In the conventional classification algorithms, like SVMs, we used to extract features from the data to make the classification work. The convolutional layers are serving the same purpose of **feature extraction**. CNNs capture better representation of data and hence we don't need to do feature engineering.
- **Classification**: After feature extraction we need to **classify the data into various classes**, this can be done using a fully connected (FC) neural network. In place of fully connected layers, we can also use a conventional classifier like **SVM**. But we generally end up adding FC layers to make the model end-to-end trainable. The fully connected layers learn a (possibly non-linear) function between the high-level features given as an output from the convolutional layers.

Visualization:

If we take as an example a layer in a FC Neural Network with an input size of 9 and an output size of 4, the operation can be visualised as follows:

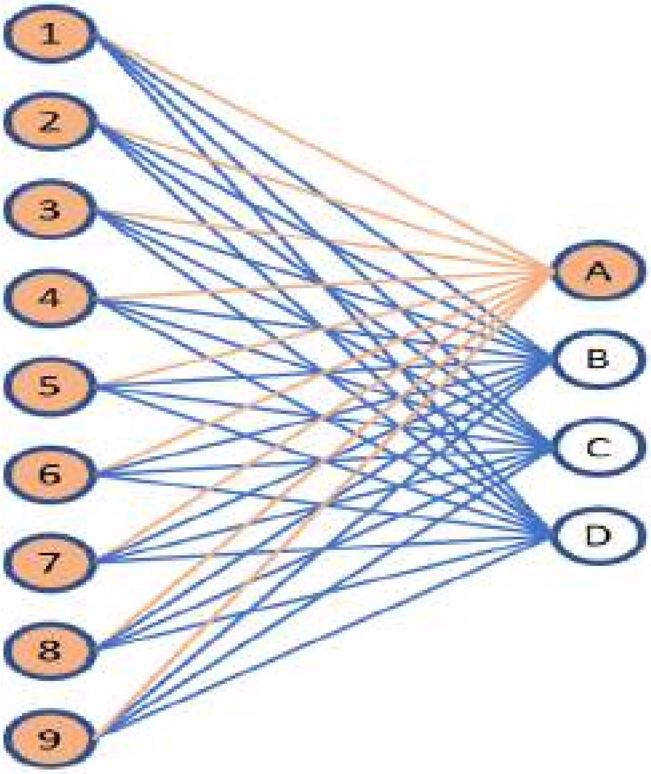


The ***activation function f*** wraps the dot product between the input of the layer and the weights matrix of that layer.

The input is a 1x9 vector, the weights matrix is a 9x4 matrix. By taking the dot product and applying the non-linear transformation with the activation function we get the output vector (1x4).

A fully connected neural network consists of a series of fully connected layers.

A fully connected layer is a function from ℝ m to ℝ n . Each output dimension depends on each input dimension. Pictorially, a fully connected layer is represented as follows in



The image above shows why we call these kinds of layers “Fully Connected” or sometimes “densely connected”.

All possible connections layer to layer are present, meaning every input of the input vector influences every output of the output vector.

A)

.Zero initialization causes the neuron to memorize the same functions almost in each iteration. Random initialization is a better choice to break the symmetry. However, initializing weight with much high or low value can result in slower optimization.

Weights should be small but not too small as it gives problems like vanishing gradient problem(vanish to 0).

B)

Shuffling training data, both before training and between epochs, helps prevent model overfitting by ensuring that batches are more representative of the entire dataset (in batch gradient descent) and that gradient updates on individual samples are independent of the sample ordering (within batches or in stochastic gradient descent); the end-result of high-quality per-epoch shuffling is better model accuracy after a set number of epochs.

C)

Suppose data is sorted in a specified order. For example a data set which is sorted base on their class. So, if you select data for training, validation, and test

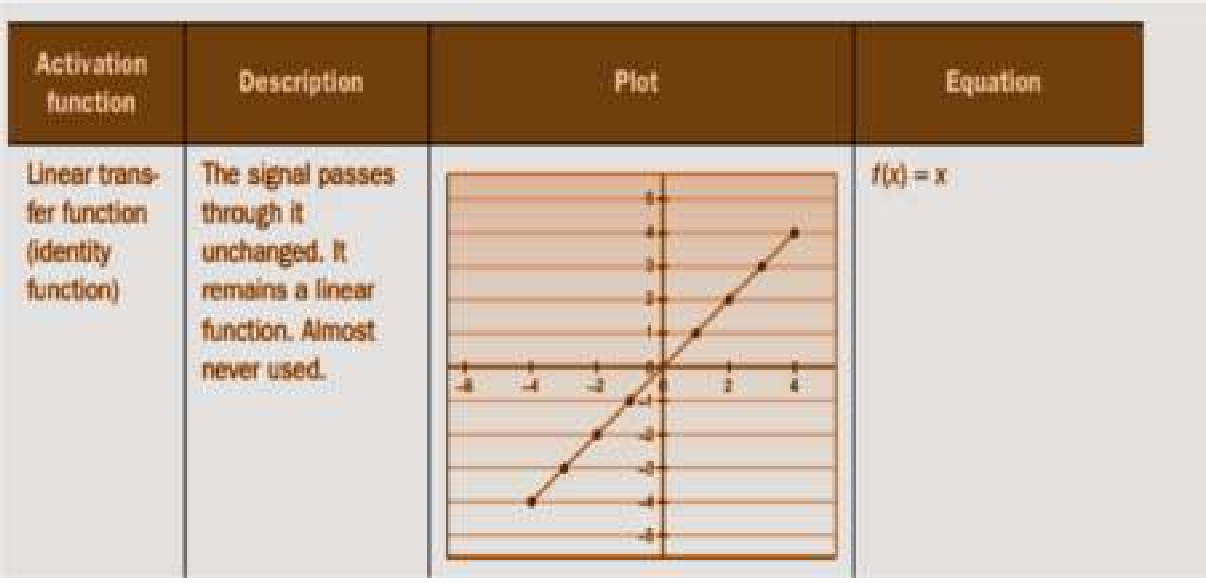
without considering this subject, you will select each class for different tasks, and it will fail the process.
Hence, to impede these kind of problems, a simple solution is shuffling the data to get different sets of training, validation, and test data

ACTIVATION FUNCTION - TYPES:

An Activation function is a deceptively small mathematical expression which decides whether a neuron fires or not. This means that the activation function suppresses the neurons whose inputs are of no significance to the overall application of the neural network. This is why neural networks require such functions which provide significant improvement in performance.
There are different types of activation functions:

1) Linear Transfer Function
2) Heaviside step function or binary classifier
3) Softmax function
4) Rectified linear unit(ReLU)
5) Leaky ReLU
6) Hyperbolic tangent function (tanh)
7) Sigmoid/logistic function

Linear Transfer Function:



Activation function	Description	Plot	Equation
Linear transfer function (identity function)	The signal passes through it unchanged. It remains a linear function. Almost never used.		$f(x) = x$

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 has a simple function with the equation:

$$f(x)=x$$

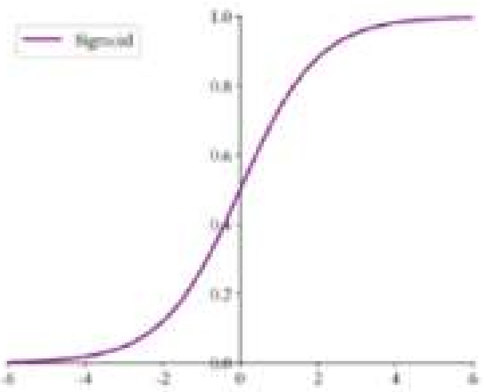
Sigmoid/Logistic Function:

The Sigmoid function is also called the logistic function, which is defined as:

Sigmoid(x) =1/1+e^-x

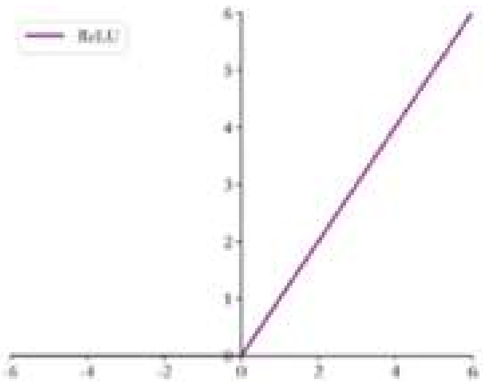
It squishes all the values to a probability between 0 and 1, which reduces extreme values or outliers. It is used to classify data into two classes

Plot:



- One of its excellent features is the ability to “compress” the input x ∈ R to an interval x ∈ (0, 1). The value of this interval is commonly used in machine learning to express the following meanings
- **Probability distribution:**The output of the interval (0, 1) matches the distribution range of probability. The output can be translated into a probability by the sigmoid function
- **Signal strength:**Usually, 0~1 can be understood as the strength of a certain signal, such as the colour intensity of the pixel: 1 represents the strongest colour of the current channel, and 0 represents the current channel without colour. It can also be used to represent the current Gate status, that is, 1 means open and 0 indicates closed.
- The Sigmoid function is continuously derivable, as shown in Figure above. The gradient descent algorithm can be directly used to optimize the network parameters.

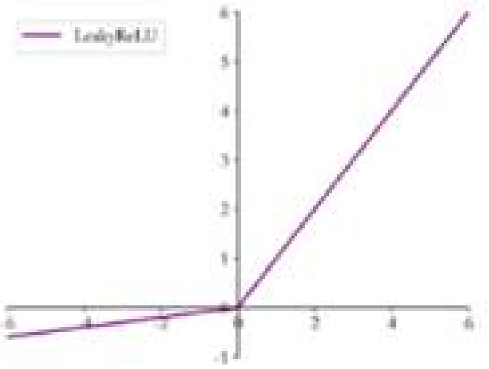
ReLU function:



Relu function activates a node only if the input is above zero. The ReLU function is defined as: ReLU (x)=max(0,x)
The function curve is shown in Figure . It can be seen that ReLU suppresses all values less than 0 to 0; for positive numbers, it outputs those directly.

Leaky ReLU:

The derivative of the ReLU function is always 0 when x<0 which may also cause gradient dispersion. To overcome this problem, the LeakyReLU function is proposed



LeakyReLU={ x , where x >=0 px x<0
where p is a small value set by users, such as 0.02.
When p = 0, the LeakyReLU function degenerates to the ReLU function.
When p ≠ 0, a small derivative value can be obtained at x

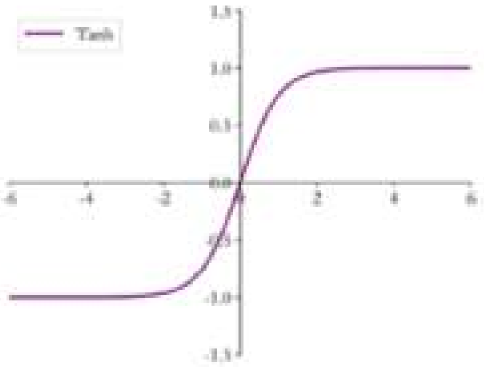
Tanh function:

The Tanh function can “compress” the input x ∈ R to an interval (−1,1), defined as:

$$tanh\ tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

$$= 2 \cdot sigmoid(2x) - 1$$

It can be seen that the Tanh activation function can be realized after zooming and translated by the Sigmoid function, as shown in Figure:



Disadvantages Of Relu Function:

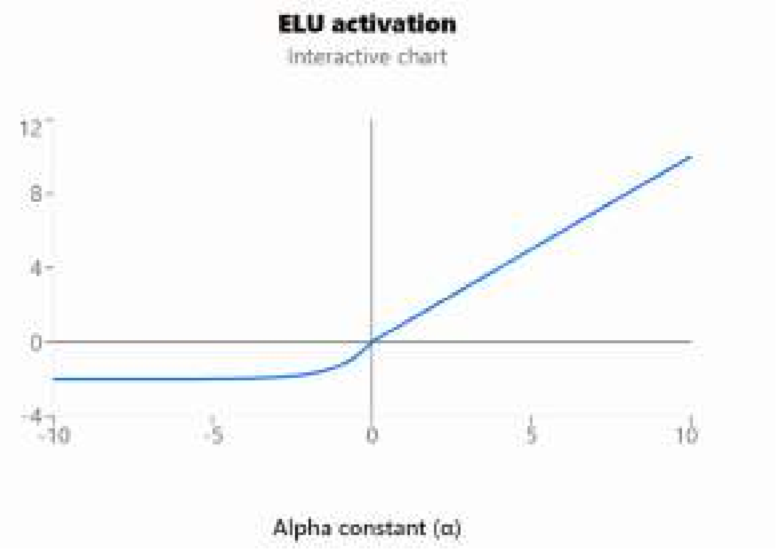
1) Exploding Gradient:

This occurs when the gradient gets accumulated, this causes a large differences in the subsequent weight updates. This as a result causes instability when converging to the global minima and causes instability in the learning too.

2) Dying ReLU:

The problem of "dead neurons" occurs when the neuron gets stuck in the negative side and constantly outputs zero. Because gradient of 0 is also 0, it's unlikely for the neuron to ever recover. This happens when the learning rate is too high or negative bias is quite large.

ELU Function:



ELU is an activation function based on ReLU that has an extra alpha constant (α) that defines function smoothness when inputs are negative. Play with an interactive example below to understand how α influences the curve for the negative part of the function.

The ELU output for positive input is the input. If the input is negative, the output curve is slightly smoothed towards the alpha constant (α). The higher the alpha constant, the more negative the output for negative inputs gets.

Advantages of ELU:

- Tend to converge faster than ReLU (because mean ELU activations are closer to zero)
- Better generalization performance than ReLU
- Fully continuous
- Fully differentiable
- Does not have a vanishing gradient's problem
- Does not have an exploding gradient problem
- Does not have a dead relu problem

NOTE: THE NEED OF A NON LINEAR MODEL IN LAYERS:

A linear model is one of the simplest models in machine learning. It has only a few parameters and can only express linear relationships. The perception and decision-making of complex brains are far more complex than a linear model. Therefore, the linear model is clearly not enough.

Complexity is the model ability to approximate complex distributions comparing a one-layer neural network model composed of a small number of neurons. Compared with the 100 billion neuron interconnection structure in the human brain, its generalization ability is obviously weaker.

Since a linear model is not feasible, we can embed a nonlinear function in the linear model and convert it to a nonlinear model. We call this nonlinear function the activation function, which is represented by

$$O= \alpha(Wx+b)$$

Here α represents a specific nonlinear activation function, such as the Sigmoid function.

We choose the ReLu function to be placed in the alpha location(α) pretty much all the time as the ReLU function only retains the positive part of function $y = x$ and sets the negative part to be zeros. It has a unilateral suppression characteristic. Although simple, the ReLU function has excellent nonlinear characteristics, easy gradient calculation, and stable training process. It is one of the most widely used activation functions for deep learning models. Here we convert the model to a nonlinear model by embedding the ReLU function

$$O=ReLU(Wx+b)$$

Hence layers in deep learning are non linear.

GRADIENT OF SIGMOID & TANH ACTIVATION FUNCTIONS:

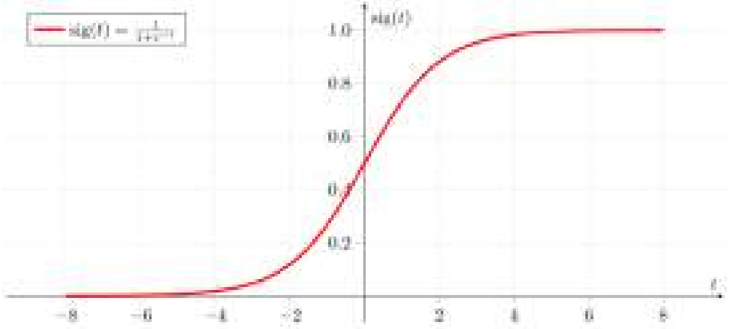
Sigmoid Activation Function :

The Sigmoid function is also called the logistic function, which is defined as:

$$Sigmoid(x) \triangleq \frac{1}{1+e^{-x}}$$

One of its excellent features is the ability to “compress” the input $x \in R$ to an interval $x \in (0, 1)$. The value of this interval is commonly used in machine learning to express the following meanings:

- **Probability distribution** The output of the interval (0, 1) matches the distribution range of probability. The output can be translated into a probability by the Sigmoid function
- **Signal strength** Usually, 0~1 can be understood as the strength of a certain signal, such as the color intensity of the pixel: 1 represents the strongest color of the current channel, and 0 represents the current channel without color. It can also be used to represent the current Gate status, that is, 1 means open and 0 indicates closed.
- Let’s take a look at the graph of the sigmoid function.



Okay, so let’s start deriving the sigmoid function!

$$
\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}\left(1+\mathrm{e}^{-x}\right)^{-1} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x) \cdot (1-\sigma(x))
\end{aligned}
$$

Gradient for sigmoid: $\sigma(x) * (1 - \sigma(x))$

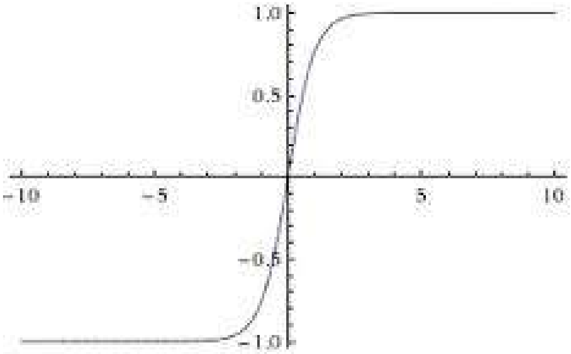
Tanh Activation Function :

The Tanh function can “compress” the input $x \in R$ to an interval (−1, 1), defined as:

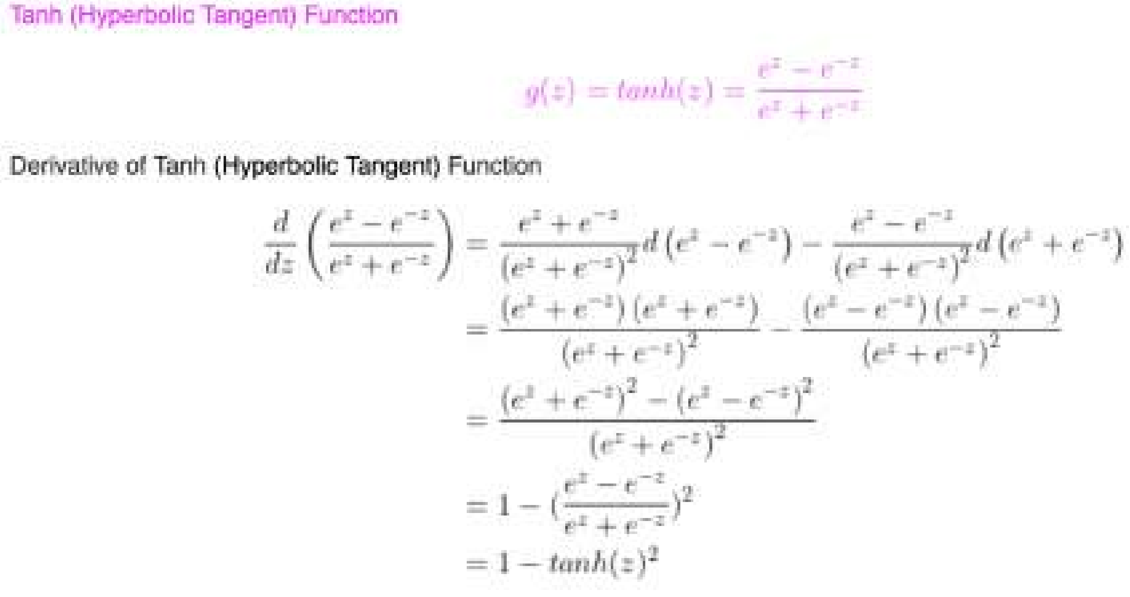
$$tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

= 2Sigmoid(2x)-1

- tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped).
- The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.
- The function is **differentiable**.
- The function is **monotonic** while its **derivative is not monotonic**.
- The tanh function is mainly used for classification between two classes.



Okay, so let’s start deriving the tanh function!



Tanh (Hyperbolic Tangent) Function

$$g(z) = tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Derivative of Tanh (Hyperbolic Tangent) Function

$$\begin{aligned}
\frac{d}{dz}\left(\frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}\right) &= \frac{e^{z} + e^{-z}}{(e^{z} + e^{-z})^2} d\left(e^{z} - e^{-z}\right) - \frac{e^{z} - e^{-z}}{(e^{z} + e^{-z})^2} d\left(e^{z} + e^{-z}\right) \\
&= \frac{(e^{z} + e^{-z})(e^{z} + e^{-z})}{(e^{z} + e^{-z})^2} - \frac{(e^{z} - e^{-z})(e^{z} - e^{-z})}{(e^{z} + e^{-z})^2} \\
&= \frac{(e^{z} + e^{-z})^2 - (e^{z} - e^{-z})^2}{(e^{z} + e^{-z})^2} \\
&= 1 - (\frac{e^{z} - e^{-z}}{e^{z} + e^{-z}})^2 \\
&= 1 - tanh(z)^2
\end{aligned}$$

Gradient for tanh: 1 − tanh2(z)

CASE STUDY : Assume that before training your neural network the setting is:

(1) The data is zero centered.

(2) All weights are initialized independently with mean 0 and variance 0.001.

(3) The biases are all initialized to 0.

(4) Learning rate is small and cannot be tuned.

Explain which activation function between tanh and sigmoid is likely to lead to a higher gradient during the first update

SOLU : We know the gradients of sigmoid and tanh activation functions, which are:

- Gradient for sigmoid: σ(z) ∗ (1 − σ(z))
- Gradient for tanh: 1 − tanh2(z)

We just have to substitute the ***z*** value(0) into respective functions.

During initialization, the expected value of ***z*** is 0.
Derivative of σ w.r.t ***z*** evaluated at zero = 0.5 * 0.5 = 0.25.
Derivative of tanh w.r.t ***z*** evaluated at zero = 1.
Hence, tanh has higher gradient magnitude during first update.

DESIGN OF OUTPUT LAYER:

Let's discuss the design of the last layer of network in particular. In addition to all hidden layers, it completes the functions of dimensional transformation and feature extraction, and it is also used as an output layer.

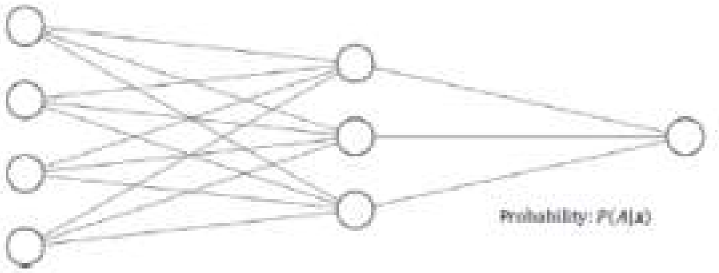
It is necessary to decide whether to use the activation function and what type of activation function to use according to the specific tasks. We will classify the discussions based on the range of output values.

[0, 1] Interval:

It is also common for output values to belong to interval [0, 1], such as image generation, and binary classification problems. The binary classification network with single output node looks like:

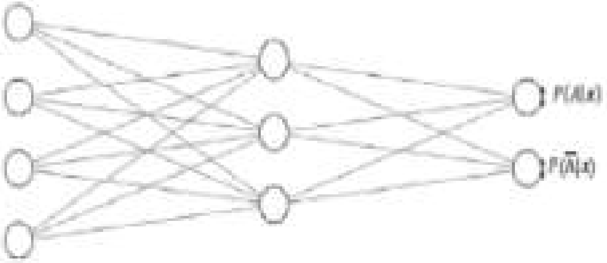
$$P(x)=o$$

$$P(x)=1-o$$



In this case, you only need to add the Sigmoid function after the value of the output layer to translate the output into a probability value.

Below figure shows the output layer of the binary classification network is two nodes.



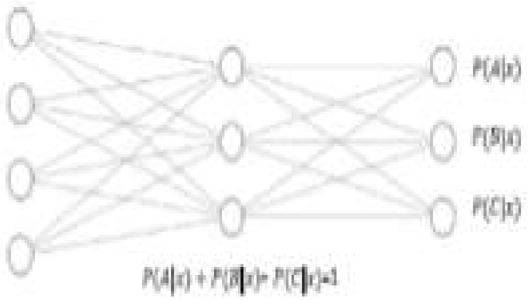
The output value of the first node represents the probability of the occurrence of event A *P*(*x*), and the output value of the second node represents the probability of the occurrence of the opposite event *P*(*x*). The function can only compress a single value to the interval (0, 1) and does not consider the relationship between the two node values. We hope that in addition to satisfy $o_i \in [0, 1]$, they can satisfy the constraint that the sum of probabilities is 1:

$$\sum_i o_i = 1$$

[0,1] Interval with Sum 1:

For cases that the output value $o_i \in [0, 1]$, and the sum of all output values is 1, it is the most common problem with multi-classification. As shown in Figure each output node of the output layer represents a category.

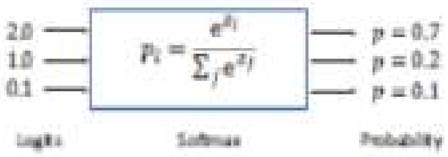
The network structure in the figure is used to handle three classification tasks. The output value distribution of the three nodes represents the probability that the current sample belongs to category A, B, and C: *P*(*x*), *P*(*B*| *x*), and *P*(*C*| *x*). Because the sample in the multi-classification problem can only belong to one of the categories, so the sum of the probabilities of all categories should be 1.



This can be achieved by adding a Softmax function to the output layer. The Softmax function is defined as:

$$Softmax(z_i) \triangleq \frac{e^{z_i}}{\sum_{j=1}^{d_{out}} e^{z_j}}$$

The Softmax function can not only map the output value to the interval [0, 1] but also satisfy the characteristic that the sum of all output values is 1.



```
z = tf.constant([2.,1.,0.1])
tf.nn.softmax(z)
Out[12]:
<tf.Tensor: id=19, shape=(3,), dtype=float32, numpy=array([0.6590012, 0.242433 , 0.0985659],
dtype=float32)>
```

(-1, 1) Interval

If you want the range of output values to be distributed in intervals (−1, 1), you can simply use the tanh activation function:

I

```
x = tf.linspace(-6.,6.,10)
tf.tanh(x)
Out[15]:
<tf.Tensor: id=264, shape=(10,), dtype=float32, numpy= array([-0.9999877 , -0.99982315, -0.997458
, -0.9640276 ,-0.58278286, 0.5827831 , 0.9640276 , 0.997458 , 0.99982315,0.9999877 ],
dtype=float32)>
```

The design of the output layer has a certain flexibility, which can be designed according to the actual application scenario, and make full use of the characteristics of the existing activation function.

ERROR CALCULATION:

- After building the model structure, the next step is to select the appropriate error function to calculate the error.
- Common error functions are mean square error, cross-entropy, KL divergence, and hinge loss.
- Among them, the mean square error function and cross-entropy function are more common in deep learning.
- The mean square error function is mainly used for regression problems, and the cross-entropy function is mainly used for classification problem.

Mean Square Error Function

Mean square error (MSE) function maps the output vector and the true vector to two points in the Cartesian coordinate system, by calculating the Euclidean distance between these two points (to be precise, the square of Euclidean distance) to measure the difference between the two vectors:

$$MSE(y,o) \triangleq \frac{1}{d_{out}} \sum_{i=1}^{d_{out}} (y_i - o_i)^2$$

The value of MSE is always greater than or equal to 0. When the MSE function reaches the minimum value of 0, the output is equal to the true label, and the parameters of the neural network reach the optimal state.

```
o = tf.random.normal([2,10]) # Network output
y_onehot = tf.constant([1,3]) # Real label
y_onehot = tf.one_hot(y_onehot, depth=10)
loss = keras.losses.MSE(y_onehot, o) # Calculate MSE
loss
Out[16]:
<tf.Tensor: id=27, shape=(2,), dtype=float32,
numpy=array([0.779179 , 1.6585705], dtype=float32)>
```

You need to average again in the sample dimension to obtain the mean square error of the average sample. The implementation is as follows:

```
loss = tf.reduce_mean(loss)
loss
Out[17]:
<tf.Tensor: id=30, shape=(), dtype=float32, numpy=1.2188747>
```

It can also be implemented in layer mode. The corresponding class is keras.losses.MeanSquaredError().
Like other classes, the __call__ function can be called to complete the forward calculation. The code is as follows:

```
criteon = keras.losses.MeanSquaredError()
loss = criteon(y_onehot,o)
loss
Out[18]:
<tf.Tensor: id=54, shape=(), dtype=float32, numpy=1.2188747>
```

Cross-Entropy Error Function:

Calculating the cross-entropy error function in a neural network involves computing the loss between the predicted values (often probabilities) generated by the model and the true labels or target values. As mentioned earlier, there are two common variants of cross-entropy loss: binary cross-entropy and categorical cross-entropy.

Binary Cross-Entropy (Binary Log Loss):

For binary classification problems, where there are only two classes (0 and 1), the binary cross-entropy loss is used. Given a true label y (0 or 1) and a predicted probability y^ (a value between 0 and 1), the binary cross-entropy loss is calculated as follows:

$$L(y,y^{\wedge})=-(y\cdot\log(y^{\wedge})+(1-y)\cdot\log(1-y^{\wedge}))$$

Where, L(y, y^) is the binary cross-entropy loss.

y is the true label (0 or 1).

y^is the predicted probability of belonging to class 1.

To calculate this loss for a batch of samples, you typically average the individual losses.

Categorical Cross-Entropy (Multi-Class Log Loss):

For multi-class classification problems, where there are more than two classes, the categorical cross-entropy loss is used. Given a true label y (a one-hot encoded vector) and predicted class probabilities y^ (a vector of predicted probabilities), the categorical cross-entropy loss is calculated as follows:

$$L(y,y^{\wedge})=-\sum_{i=1}^{N}y_i\cdot\log(y^{\wedge}_i)$$

Where, L(y, y^) is the categorical cross-entropy loss.

y is a one-hot encoded vector representing the true class.

y^is a vector of predicted class probabilities for each class.

N is the number of classes.

To calculate this loss for a batch of samples, you typically average the individual losses.

Here's an example of how to set the loss function in Keras:

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

model = Sequential([
    Dense(units=64, activation='relu', input_dim=input_dim),
    Dense(units=1, activation='sigmoid')  # For binary classification
])

model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
```

For multi-class classification, you would typically use 'categorical_crossentropy' as the loss function and adapt your model architecture accordingly.

During training, the optimization algorithm minimizes the chosen loss function, which means it adjusts the model's parameters to make the predicted values (probabilities) closer to the true labels.

TYPES OF NEURAL NETWORKS:

There are various types of neural networks, each designed to address specific types of machine learning tasks. Here are some of the most common types of neural networks:

Feedforward Neural Network (FNN):

- Also known as Multi-Layer Perceptrons (MLP).
- The simplest form of neural network.
- Consists of an input layer, one or more hidden layers, and an output layer.
- Used for tasks like classification and regression.

Convolutional Neural Network (CNN):

- Specifically designed for processing grid-like data, such as images.
- Employs convolutional layers to automatically learn spatial hierarchies of features.
- Widely used in image classification, object detection, and image segmentation.

Recurrent Neural Network (RNN):

- Designed for sequential data and time-series analysis.
- Contains recurrent layers that maintain hidden states to capture temporal dependencies.
- Suitable for tasks like natural language processing, speech recognition, and time-series prediction.

Long Short-Term Memory (LSTM):

- A type of RNN with improved ability to capture long-term dependencies.
- Utilizes memory cells and gates to control the flow of information.
- Excellent for sequential tasks where context over long sequences is essential.

Gated Recurrent Unit (GRU):

- Similar to LSTM but with a simpler architecture.
- Uses gating mechanisms to control information flow.
- Offers a balance between performance and complexity compared to LSTM.

Autoencoder (AE):

- Unsupervised learning neural network used for dimensionality reduction and feature learning.
- Comprises an encoder to reduce input data dimensions and a decoder to reconstruct the original data.
- Used in image denoising, anomaly detection, and recommendation systems.
-

Variational Autoencoder (VAE):

- An extension of autoencoders with probabilistic properties.
- Encourages the model to generate data points similar to those in the training dataset.
- Commonly used in generating new data samples and data representation learning.

Generative Adversarial Network (GAN):

- Comprises a generator network and a discriminator network.
- Trains by having the generator and discriminator compete against each other.
- Used for generating synthetic data, image-to-image translation, and style transfer.

Radial Basis Function Network (RBFN):

- Utilizes radial basis functions as activation functions.
- Suitable for interpolation, approximation, and function approximation tasks.

Self-Organizing Maps (SOM):

- Used for clustering and dimensionality reduction.
- Organizes data points in a low-dimensional grid while preserving topological relationships.

Residual Neural Network (ResNet):

- Addresses the vanishing gradient problem by using skip connections.
- Enables the training of extremely deep neural networks.
- Commonly used in image recognition tasks.

Siamese Network:

- Designed for tasks involving similarity or dissimilarity comparisons.
- Consists of two identical subnetworks with shared weights.
- Often used in face recognition and signature verification.

Transformers:

- Introduced in the field of natural language processing (NLP).
- Utilizes attention mechanisms to capture contextual information.
- The basis for models like BERT, GPT, and T5 for various NLP tasks.

Graph Convolutional Neural Network (Graph CNN or GCN):

- Designed for processing graph-structured data.
- Utilizes graph convolutional layers to propagate information between connected nodes in a graph.
- Used in tasks such as node classification, link prediction, and graph classification in areas like social network analysis and recommendation systems.

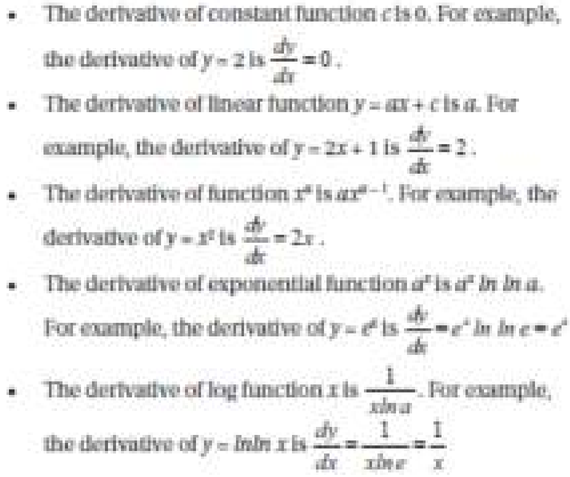
Attention Mechanism:

- Not a standalone network architecture but a mechanism integrated into various neural networks.
- Introduced in models like Transformers.
- Allows the model to focus on different parts of the input sequence when making predictions.
- Essential for capturing long-range dependencies in sequential data.
- Used in natural language processing for tasks like machine translation, text summarization, and question-answering.

These are some of the fundamental types of neural networks, and there are many more specialized architectures and variations tailored to specific applications and research areas. The choice of the neural network architecture depends on the nature of the problem you want to solve.

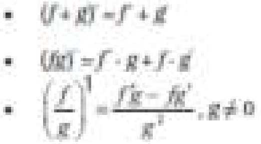
DERIVATIVES AND GRADIENTS:

- Neural network model expressions are usually very complex, and the model parameters can reach tens or hundreds of millions of levels.
- Almost all neural network optimization problems rely on deep learning frameworks to automatically calculate the gradient of network parameters and then use gradient descent to iteratively optimize the network parameters until the performance meets the requirement.
- The main algorithms implemented in deep learning frameworks are back propagation and gradient descent algorithms. So understanding the principles of these two algorithms is helpful to understand the role of deep learning frameworks.
- Before introducing the back propagation algorithm of the multilayer neural network, we first introduce the common attributes of the derivative, the gradient derivation of the common activation function, and the loss function and then derive the gradient propagation law of the multilayer neural network.
- COMMON DERIVATIVES:

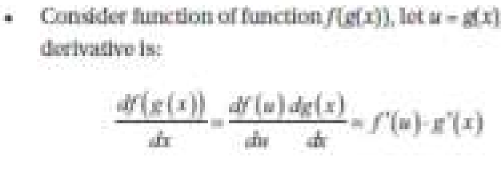


- The derivative of constant function c is 0. For example, the derivative of $y = 2$ is $\frac{dy}{dx} = 0$.
- The derivative of linear function $y = ax + c$ is a. For example, the derivative of $y = 2x + 1$ is $\frac{dy}{dx} = 2$.
- The derivative of function x^a is ax^{a-1}. For example, the derivative of $y = x^2$ is $\frac{dy}{dx} = 2x$.
- The derivative of exponential function a^x is $a^x \ln \ln a$. For example, the derivative of $y = e^x$ is $\frac{dy}{dx} = e^x \ln \ln e = e^x$
- The derivative of log function x is $\frac{1}{x \ln a}$. For example, the derivative of $y = \ln \ln x$ is $\frac{dy}{dx} = \frac{1}{x \ln e} = \frac{1}{x}$

- COMMON PROPERTIES OF DERIVATIVES:



- $(f + g)' = f' + g'$
- $(fg)' = f' \cdot g + f \cdot g'$
- $\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}, g \neq 0$

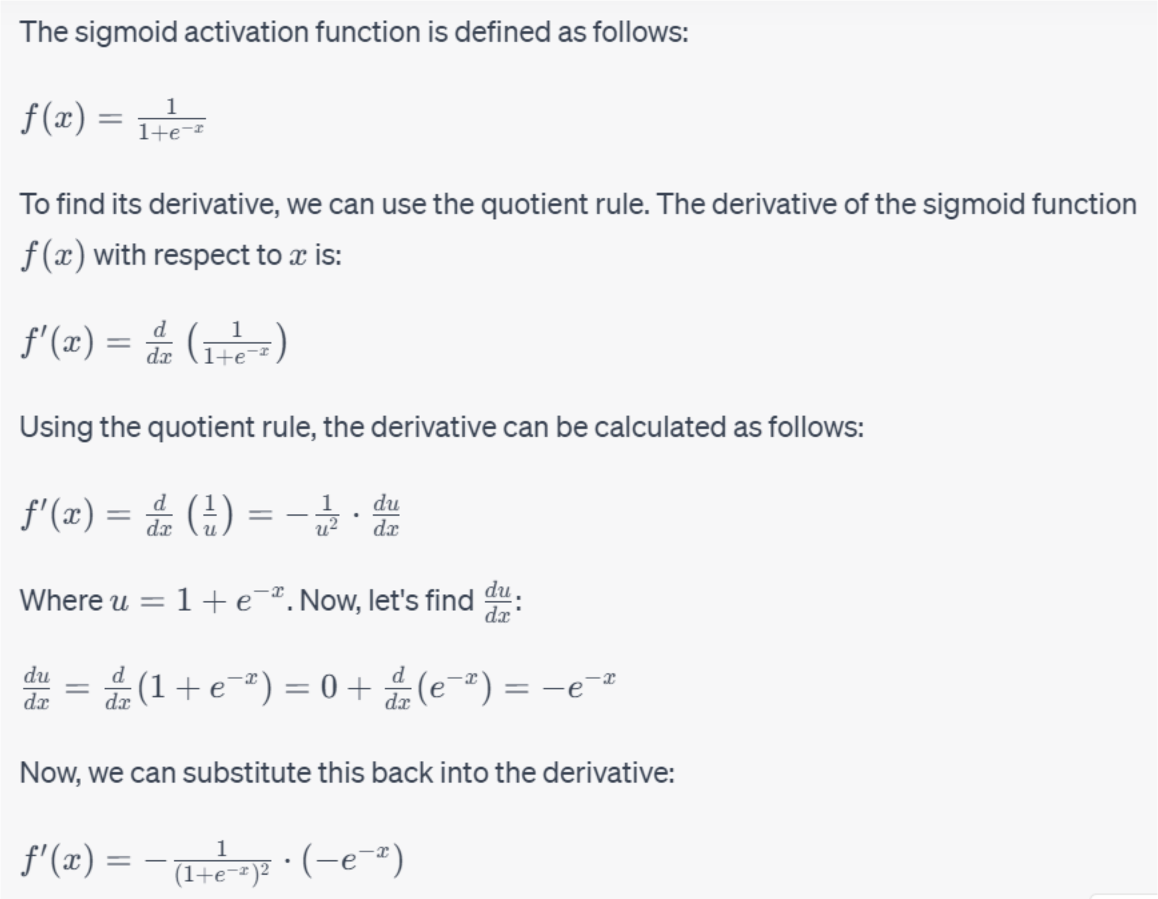


- Consider function of function $f(g(x))$, let $u = g(x)$, the derivative is:

$$\frac{df(g(x))}{dx} = \frac{df(u)}{du}\frac{dg(x)}{dx} = f'(u) \cdot g'(x)$$

DERIVATIVE OF ACTIVATION FUNCTIONS:

DERIVATIVE OF SIGMOID ACTIVATION FUNCTION:



The sigmoid activation function is defined as follows:

$$f(x) = \frac{1}{1+e^{-x}}$$

To find its derivative, we can use the quotient rule. The derivative of the sigmoid function $f(x)$ with respect to x is:

$$f'(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right)$$

Using the quotient rule, the derivative can be calculated as follows:

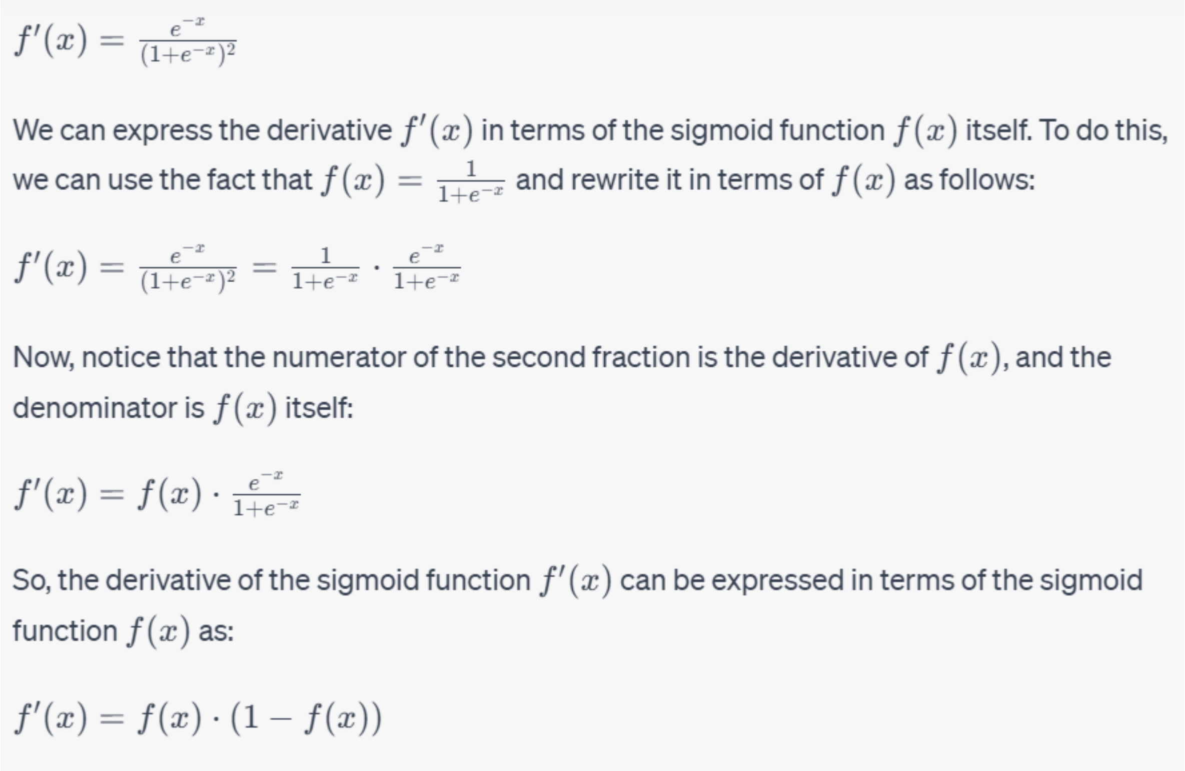
$$f'(x) = \frac{d}{dx}\left(\frac{1}{u}\right) = -\frac{1}{u^2} \cdot \frac{du}{dx}$$

Where $u = 1 + e^{-x}$. Now, let's find $\frac{du}{dx}$:

$$\frac{du}{dx} = \frac{d}{dx}(1 + e^{-x}) = 0 + \frac{d}{dx}(e^{-x}) = -e^{-x}$$

Now, we can substitute this back into the derivative:

$$f'(x) = -\frac{1}{(1+e^{-x})^2} \cdot (-e^{-x})$$



$$f'(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

We can express the derivative $f'(x)$ in terms of the sigmoid function $f(x)$ itself. To do this, we can use the fact that $f(x) = \frac{1}{1+e^{-x}}$ and rewrite it in terms of $f(x)$ as follows:

$$f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}}$$

Now, notice that the numerator of the second fraction is the derivative of $f(x)$, and the denominator is $f(x)$ itself:

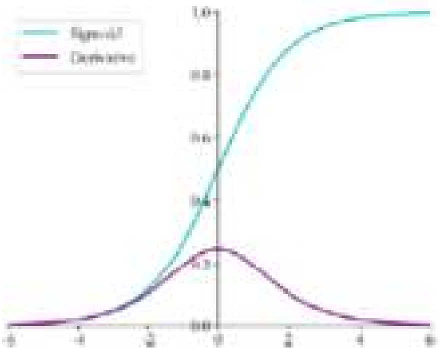
$$f'(x) = f(x) \cdot \frac{e^{-x}}{1+e^{-x}}$$

So, the derivative of the sigmoid function $f'(x)$ can be expressed in terms of the sigmoid function $f(x)$ as:

$$f'(x) = f(x) \cdot (1 - f(x))$$

$$= \sigma(1-\sigma)$$

It can be seen that the derivative expression of the Sigmoid function can finally be expressed as a simple operation of the output value of the activation function. Using this property, we can calculate its derivate by caching the output value of the Sigmoid function of each layer in the gradient calculation of the neural network.



Sigmoid function and its derivative

NumPy code:

```
import numpy as np # import numpy library
def sigmoid(x): # implement sigmoid function
        return 1 / (1 + np.exp(-x))
def derivative(x): # calculate derivative of sigmoid
# Using the derived expression of the derivatives
        return sigmoid(x)*(1-sigmoid(x))
```

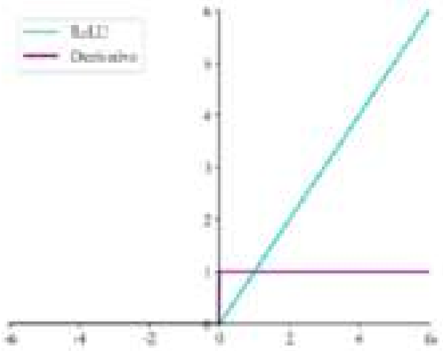
DERIVATIVE OF ReLU ACTIVATION FUNCTION:

- Before the ReLU function was widely used, the activation function in neural networks was mostly Sigmoid. However, the Sigmoid function was prone to gradient dispersion. (When the number of layers of the network increased, because the gradient values become very small, the parameters of the network cannot be effectively updated. As a result, deeper neural networks cannot be trained, resulting in the research of neural networks staying at the shallow level.)
- With the introduction of the ReLU function, the phenomenon of gradient dispersion is well alleviated, and the number
- of layers of the neural network can reach deeper layers.
- The expression of the ReLU function: $ReLU\ (x) = max(0,x)$

- The derivation of its derivative is very simple:

  $d/dx\ [ReLU] = 1$ For $x >= 0$ and $d/dx\ [ReLU] = 0$ For $x < 0$

- It can be seen that the derivative calculation of the ReLU function is simple. When x is greater than or equal to zero, the derivative value is always 1.
- In the process of back propagation, it will neither amplify the gradient, causing gradient exploding, nor shrink the gradient, causing gradient vanishing phenomenon.
- The derivative curve of the ReLU function is shown in Figure:



ReLU function and its derivative

NumPy code:

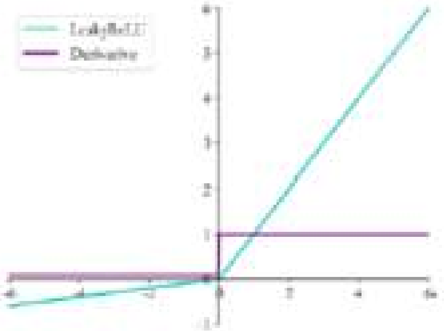
```
def derivative(x): # Derivative of ReLU
        d = np.array(x, copy=True)
        d[x < 0] = 0
        d[x >= 0] = 1
        return d
```

DERIVATIVE OF LEAKY ReLU ACTIVATION FUNCTION:

- The expression of LeakyReLU function:
  LeakyReLU =x for x >= 0 and *LeakyReLU* =px *for x*< 0

- Its derivative can be derived as:
  d/dx[LeakyReLU] = 1 for *x* >= 0 and *d/dx[LeakyReLU] =px* for x< 0

- It's different from the ReLU function because when x is less than zero, the derivative value of the LeakyReLU function is not 0, but a constant p, which is generally set to a smaller value, such as 0.01 or 0.02.
- The derivative curve of the LeakyReLU function is shown in Figure:



LeakyReLU function and its derivative

NumPy Code:

```
def derivative(x, p): # p is the slope of negative part of
LeakyReLU
        dx = np.ones_like(x) # initialize a vector with 1
        dx[x < 0] = p # set negative part to p
        return dx
```

DERIVATIVE OF TANH ACTIVATION FUNCTION:

$$tanh\left(x\right)=\frac{\left(e^{x}-e^{-x}\right)}{\left(e^{x}+e^{-x}\right)}$$

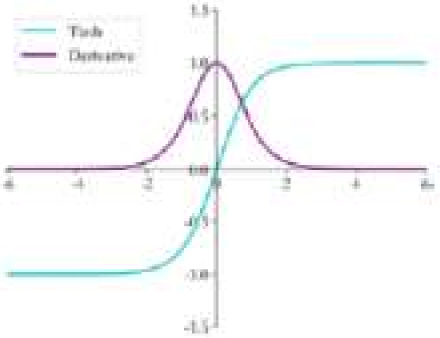
$$=2\cdot sigmoid\left(2x\right)-1$$

Its derivative expression is:

$$\frac{d}{dx}tanh\ tanh\left(x\right)=\frac{\left(e^{x}+e^{-x}\right)\left(e^{x}+e^{-x}\right)-\left(e^{x}-e^{-x}\right)\left(e^{x}-e^{-x}\right)}{\left(e^{x}+e^{-x}\right)^{2}}$$

$$=1-\frac{\left(e^{x}-e^{-x}\right)^{2}}{\left(e^{x}+e^{-x}\right)^{2}}=1-\left(x\right)$$

The Tanh function and its derivative curve are shown in Figure



NumPy code:

```
def sigmoid(x): # sigmoid function
        return 1 / (1 + np.exp(-x))
def tanh(x): # tanh function
        return 2*sigmoid(2*x) - 1
def derivative(x): # derivative of tanh
        return 1-tanh(x)**2
```

BACK PROPOGATION ALGORITHM

GRADIENT OF LOSS FUNCTION:

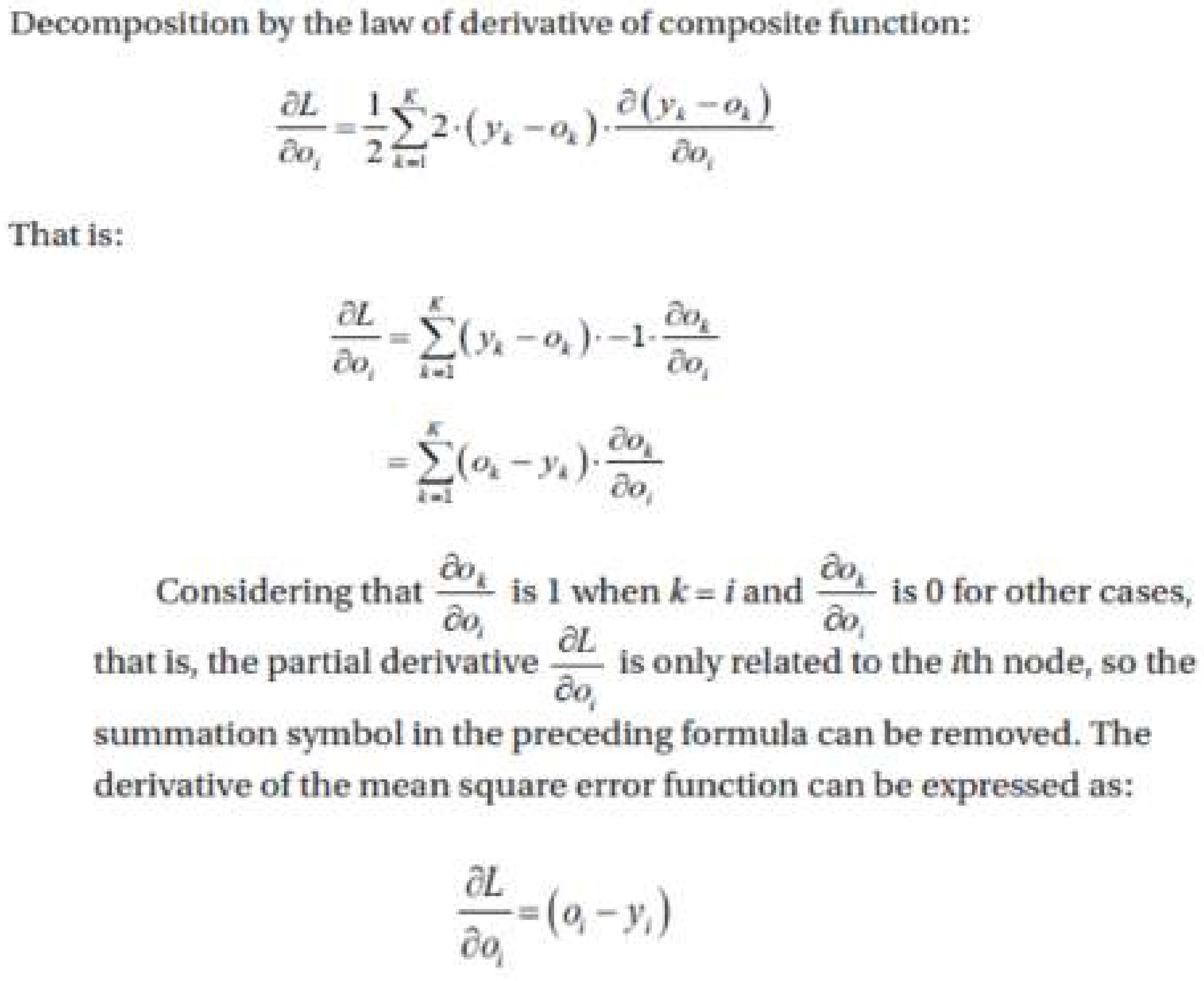
The loss function is a method of evaluating how well your machine learning algorithm models your featured data set.
In other words, loss functions are a measurement of how good your model is in terms of predicting the expected outcome.
Loss function refer to the training process that uses back-propagation to minimize the error between the actual and predicted outcome.

Here we mainly derive the gradient expressions of the mean square error loss function and the cross-entropy loss function.The mean square error loss function expression is:

$$L = \frac{1}{2}\sum_{k=1}^{K}(y_k - o_k)^2$$

Then its partial derivative is:

$$\frac{\partial L}{\partial o_i} = \frac{1}{2}\sum_{k=1}^{K}\frac{\partial}{\partial o_i}(y_k - o_k)^2$$



Decomposition by the law of derivative of composite function:

$$\frac{\partial L}{\partial o_i} = \frac{1}{2}\sum_{k=1}^{K}2\cdot(y_k - o_k)\cdot\frac{\partial(y_k - o_k)}{\partial o_i}$$

That is:

$$\frac{\partial L}{\partial o_i} = \sum_{k=1}^{K}(y_k - o_k)\cdot -1\cdot\frac{\partial o_k}{\partial o_i}$$

$$= \sum_{k=1}^{K}(o_k - y_k)\cdot\frac{\partial o_k}{\partial o_i}$$

Considering that $\frac{\partial o_k}{\partial o_i}$ is 1 when $k = i$ and $\frac{\partial o_k}{\partial o_i}$ is 0 for other cases, that is, the partial derivative $\frac{\partial L}{\partial o_i}$ is only related to the *i*th node, so the summation symbol in the preceding formula can be removed. The derivative of the mean square error function can be expressed as:

$$\frac{\partial L}{\partial o_i} = (o_i - y_i)$$

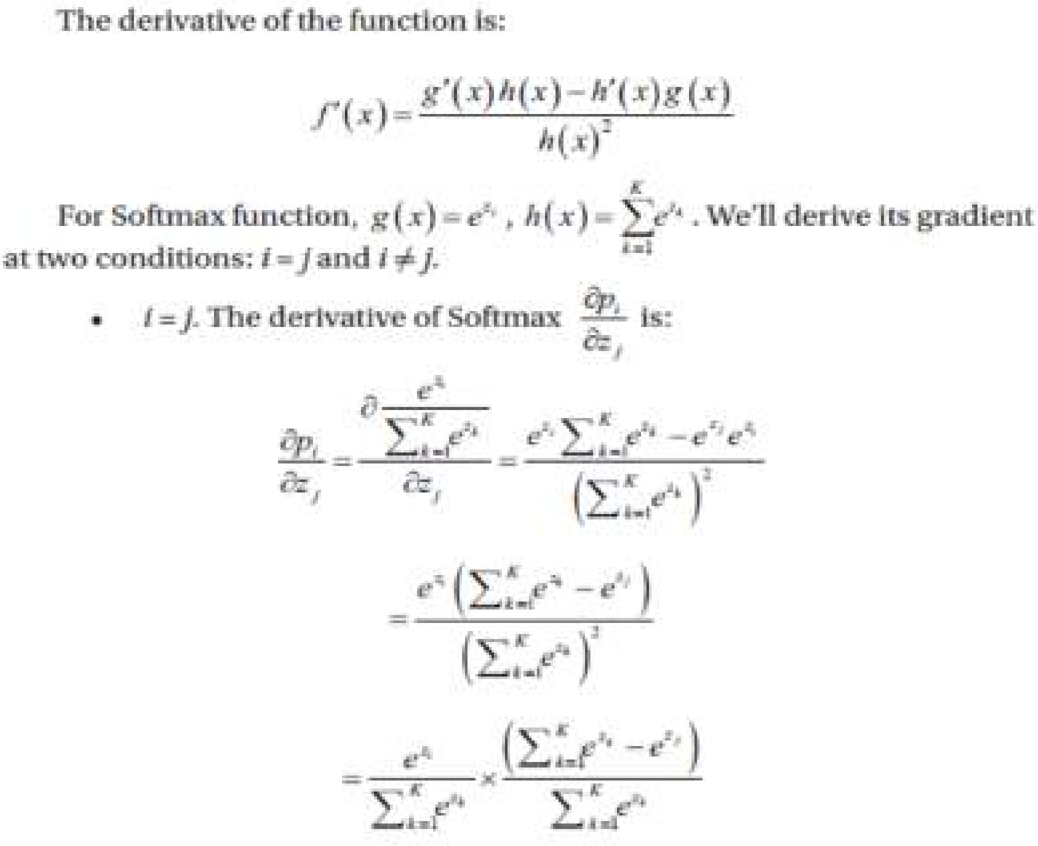
GRADIENT OF CROSS ENTROPY FUNCTION:

When calculating the cross-entropy loss function, the Softmax function and the cross-entropy function are generally implemented in a unified manner.
We first derive the gradient of the Softmax function, and then derive the gradient of the cross-entropy function.
Gradient of Softmax: The expression of Softmax:

$$p_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

We know that if

$$f(x) = \frac{g(x)}{h(x)}$$



The derivative of the function is:

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$$

For Softmax function, $g(x) = e^{z_i}$, $h(x) = \sum_{k=1}^{K} e^{z_k}$. We'll derive its gradient at two conditions: $i = j$ and $i \neq j$.

- $i = j$. The derivative of Softmax $\frac{\partial p_i}{\partial z_j}$ is:

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}}{\partial z_j} = \frac{e^{z_i} \sum_{k=1}^{K} e^{z_k} - e^{z_j} e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2}$$

$$= \frac{e^{z_i}\left(\sum_{k=1}^{K} e^{z_k} - e^{z_j}\right)}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2}$$

$$= \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \times \frac{\left(\sum_{k=1}^{K} e^{z_k} - e^{z_j}\right)}{\sum_{k=1}^{K} e^{z_k}}$$

The preceding expression is the multiplication of p_i and $1 - p_j$, and $p_i = p_j$. So when $i = j$, the derivative of Softmax $\frac{\partial p_i}{\partial z_j}$ is:

$$\frac{\partial p_i}{\partial z_j} = p_i\left(1 - p_j\right), i = j$$

- $i \neq j$. Extend the Softmax function:

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}}{\partial z_j} = \frac{0 - e^{z_j} e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2}$$

$$= \frac{-e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \times \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

That is:

$$\frac{\partial p_i}{\partial z_j} = -p_j \cdot p_i$$

Gradient of cross-entropy function Consider the expression of the cross-entropy loss function:

$$L = -\sum_k y_k \, log \, log\left(p_k\right)$$

Here we directly derive the partial derivative of the final loss value *L* to the logits variable z_i of the network output, which expands to:

$$\frac{\partial L}{\partial z_i} = -\sum_k y_k \frac{\partial \, log \, log\left(p_k\right)}{\partial z_i}$$

Decompose the composite function *log log h* into:

$$= -\sum_k y_k \frac{\partial \, log \, log\left(p_k\right)}{\partial p_k} \cdot \frac{\partial p_k}{\partial z_i}$$

That is:

$$= -\sum_k y_k \frac{1}{p_k} \cdot \frac{\partial p_k}{\partial z_i}$$

where $\frac{\partial p_k}{\partial z_i}$ is the partial derivative of the Softmax function that we have derived.

Split the summation symbol into the two cases: $k = i$ and $k \neq i$, and substitute the expression of $\frac{\partial p_k}{\partial z_i}$, we can get:

$$\frac{\partial L}{\partial z_i} = -y_i(1-p_i) - \sum_{k \neq i} y_k \frac{1}{p_k}(-p_k \cdot p_i)$$

$$= -y_i(1-p_i) + \sum_{k \neq i} y_k \cdot p_i$$

$$= -y_i + y_i p_i + \sum_{k \neq i} y_k \cdot p_i$$

That is:

$$\frac{\partial L}{\partial z_i} = p_i\left(y_i + \sum_{k \neq i} y_k\right) - y_i$$

In particular, the one-hot encoding method for the label in the classification problem has the following relationship:

$$\sum_k y_k = 1$$

$$y_i + \sum_{k \neq i} y_k = 1$$

Therefore, the partial derivative of cross-entropy can be further simplified to:

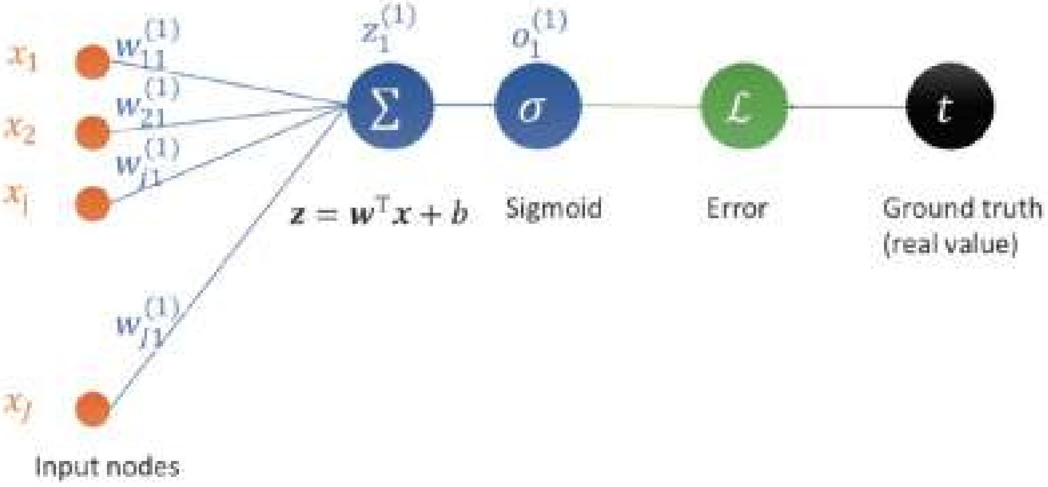
$$\frac{\partial L}{\partial z_i} = p_i - y_i$$

GRADIENT OF FULLY CONNECTED NETWORK:

GRADIENT OF SINGLE NEURON:

- For a neuron model using Sigmoid activation function, its mathematical model can be written as:

  O(1)=sigma(w(1)^T+b(1))
- The superscript of the variable represents the number of layers. For example, o(1) represents the output of the first layer and x is the input of the network
- We take the partial derivative derivation ∂L/ ∂wj1 of the weight parameter wj1 as an example
- The number of input nodes is J. The weight connection from the input of the jth node to the output o(1) is denoted as wj1 (1) , where the superscript indicates the number of layers to which the weight parameter belongs, and the subscript indicates the starting node number and the ending node number of the current connection.
- For example, the subscript j1 indicates the jth node of the previous layer to the first node of the current layer.
- The variable before the activation function σ is called z1(1) , and the variable after the activation function σ is called o1(1) . Because there is only one output node, so o1(1)=o(1)=o .
- The error value L is calculated by the error function between the output and the real label



- The loss can be expressed as:

  L =1/2(o1(1) - t)^2 = ½(o(1) - t)^2
- Among them, t is the real label value. Adding 1/2 does not affect the direction of the gradient, and the calculation is simpler
- We take the weight variable wj1 of the jth (j ∈ [1,J]) node as an example and consider the partial derivative of the loss function L:

$$\partial L/\partial wj1 = (o1 - t)\ \partial o1/\partial wj1$$

- Considering o1 = σ(z1) and the derivative of the Sigmoid function is σ′ = σ(1 − σ), we have:

$$\frac{\partial L}{\partial w_{j1}} = (o_1 - t)\frac{\partial \sigma(z_1)}{\partial w_{j1}}$$

$$= (o_1 - t)\sigma(z_1)(1-\sigma(z_1))\frac{\partial z_1^{(1)}}{\partial w_{j1}}$$

Write $\sigma(z_1)$ *as* o_1:

$$\frac{\partial L}{\partial w_{j1}} = (o_1 - t)o_1(1-o_1)\frac{\partial z_1^{(1)}}{\partial w_{j1}}$$

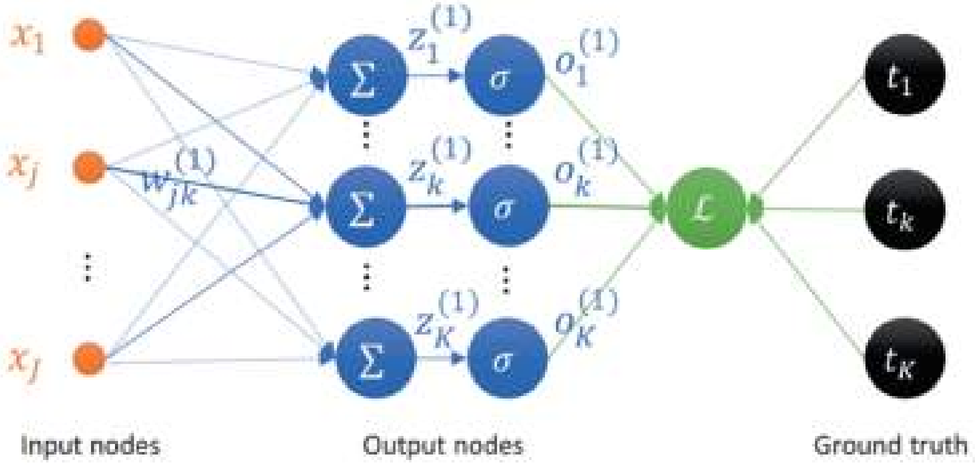
Consider $\frac{\partial z_1^{(1)}}{\partial w_{j1}} = x_j$, we have:

$$\frac{\partial L}{\partial w_{j1}} = (o_1 - t)o_1(1-o_1)x_j$$

- It can be seen from the preceding formula that the partial derivative of the error to the weight wj1 is only related to the output value o1, the true value t, and the input xj connected to the current weight

GRADIENT OF FULLY CONNECTED LAYER

- We generalize the single neuron model to a single-layer network of fully connected layers, as shown in Figure .
- The input layer obtains the output vector o(1) through a fully connected layer and calculates the mean square error with the real label vector t.
- The number of input nodes is J, and the number of output nodes is K.



- The multi-output fully connected network layer model differs from the single neuron model in that it has many more output nodes o1(1) ,o2(1) ,o3(1) ,ok(1) …… , and each output node corresponds to a real label t1, t2, …, tK. wjk is the connection weight of the jth input node and the kth output node. The mean square error can be expressed as:

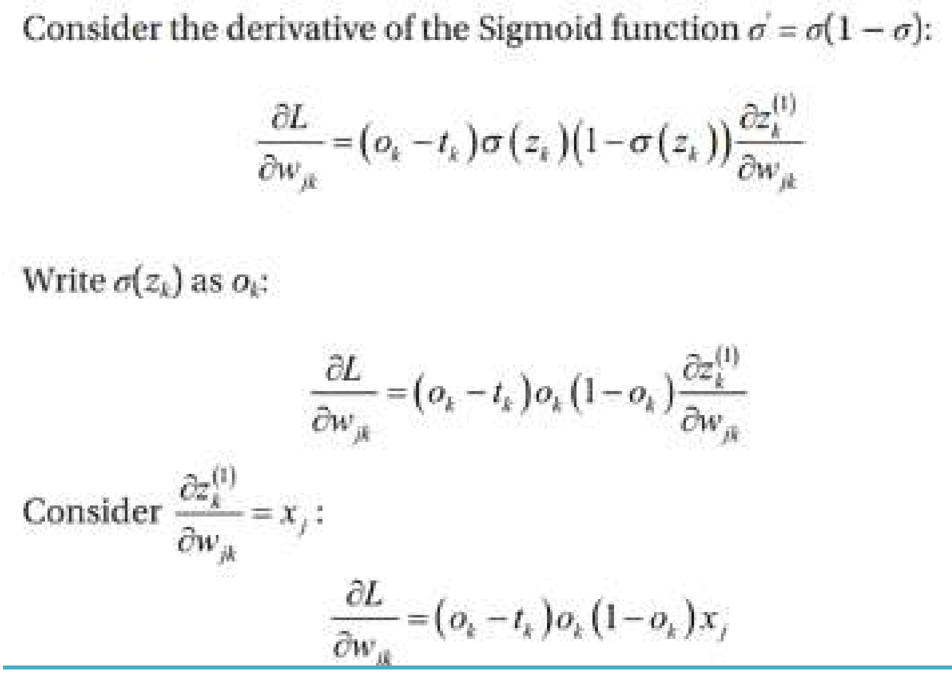
- $$L=\frac{1}{2}\sum_{i=1}^{K}\left(o_i^{(1)}-t_i\right)^2$$

- Since ∂L / ∂wjk is only associated with node ok (1) , the summation symbol in the preceding formula can be removed, that is, i = k:

- $$\frac{\partial L}{\partial w_{jk}}=\left(o_k-t_k\right)\frac{\partial o_k}{\partial w_{jk}}$$

- Substitute ok = σ(zk):

- $$\frac{\partial L}{\partial w_{jk}}=\left(o_k-t_k\right)\frac{\partial\sigma\left(z_k\right)}{\partial w_{jk}}$$



Consider the derivative of the Sigmoid function $\sigma'=\sigma(1-\sigma)$:

$$\frac{\partial L}{\partial w_{jk}}=\left(o_k-t_k\right)\sigma\left(z_k\right)\left(1-\sigma\left(z_k\right)\right)\frac{\partial z_k^{(1)}}{\partial w_{jk}}$$

Write $\sigma(z_k)$ as o_k:

$$\frac{\partial L}{\partial w_{jk}}=\left(o_k-t_k\right)o_k\left(1-o_k\right)\frac{\partial z_k^{(1)}}{\partial w_{jk}}$$

Consider $\frac{\partial z_k^{(1)}}{\partial w_{jk}}=x_j$:

- $$\frac{\partial L}{\partial w_{jk}}=\left(o_k-t_k\right)o_k\left(1-o_k\right)x_j$$

- It can be seen that the partial derivative of wjk is only related to the output node ok (1) of the current connection, the label tk (1) of the corresponding true, and the corresponding input node xj.

$$\text{Let } \delta_k = (o_k - t_k)o_k(1 - o_k),\ \frac{\partial L}{\partial w_{jk}} \text{ becomes:}$$

$$\frac{\partial L}{\partial w_{jk}} = \delta_k x_j$$

- The variable δk characterizes a certain characteristic of the error gradient propagation of the end node of the connection line.

- After using the representation δk, the partial derivative ∂L/ ∂wjk is only related to the start node xj and the end node δk of the current connection.

- Now that the gradient propagation method of the single-layer neural network (i.e., the output layer) has been derived, next we try to derive the gradient propagation method of the penultimate layer.

- After completing the propagation derivation of the penultimate layer, similarly, the gradient propagation mode of all hidden layers can be derived cyclically to obtain gradient calculation expressions of all layer parameters.

CHAIN RULE:

The chain rule is a fundamental concept in calculus that allows you to find the derivative of a composite function. In other words, it helps you calculate the rate of change of a function that is composed of two or more functions nested inside each other. The chain rule is especially important in calculus when dealing with functions where one quantity depends on another, which in turn depends on another, and so on.

Mathematically, the chain rule is stated as follows:

If you have a composite function y = f(g(x)), where:

y is the final output or dependent variable,

f(u) is the outer function,

g(x) is the inner function, and u = g(x), Then, the derivative of y with respect to x, denoted as dy/dx, is calculated as:

$dy/dx = (df/du) * (du/dx)$

In words, the chain rule says that to find the derivative of the composite function y = f(g(x)), you multiply the derivative of the outer function f(u) with respect to its variable u by the derivative of the inner function g(x) with respect to x.

Here's a more concrete example:

Let $y = f(u) = u^2$, and $u = g(x) = 3x - 1$. We want to find dy/dx.

Find df/du: $df/du = 2u$.

Find du/dx: $du/dx = 3$.

Now, apply the chain rule:

$dy/dx = (df/du) * (du/dx)$

$dy/dx = (2u) * (3)$

Since u = g(x), we can substitute u back in:

$dy/dx = (2(3x - 1)) * 3$

$dy/dx = 6(3x - 1)$

So, $dy/dx = 18x - 6$.

The chain rule is a powerful tool that enables you to find derivatives of complex functions by breaking them down into simpler functions and considering how changes in the inner function affect the outer function. It's a fundamental concept in calculus and is widely used in various fields of mathematics and science.

CHAIN RULE IN GRADIENT PROPOGATION:

In the context of machine learning and neural networks, the chain rule plays a crucial role in gradient propagation during the training process. Gradient propagation is the process of computing gradients (derivatives) of a composite function, such as a neural network, with respect to its parameters. The chain rule is used to calculate these gradients efficiently, allowing the model to update its parameters during training through optimization algorithms like gradient descent.

Let's break down how the chain rule is used in gradient propagation:

Neural Network Forward Pass:

During the forward pass of a neural network, input data is processed through multiple layers. Each layer applies an activation function to its input and produces an output. This process can be represented as a sequence of functions:

Input data: x

Layer 1: $h_1 = f_1(W_1 * x + b_1)$

Layer 2: $h_2 = f_2(W_2 * h_1 + b_2)$

...

Output layer: $y = f_o(W_o * h_k + b_o)$

Here, each layer has its own weights (W) and biases (b), and f_1, f_2, ..., f_o are activation functions.

Computing Loss:

The neural network makes predictions (y) based on the input data, and the predictions are compared to the actual target values to compute a loss function (e.g., Mean Squared Error or Cross-Entropy Loss).

Gradient Calculation - Backpropagation:

The goal of training is to minimize the loss. To do this, you need to calculate the gradients of the loss with respect to the parameters (weights and biases) in each layer of the network. This is where the chain rule comes into play.

Starting from the output layer and moving backward through the network (hence the term "backpropagation"), you calculate the gradients layer by layer using the chain rule. The key steps are as follows:

Compute the gradient of the loss with respect to the output layer's inputs.

Propagate this gradient backward through each layer, multiplying it by the gradient of the layer's inputs with respect to its parameters, using the chain rule.

Mathematically, the chain rule is applied as follows:

$\partial Loss/\partial W_o = \partial Loss/\partial y * \partial y/\partial(W_o * h_k + b_o)$

$\partial Loss/\partial b_o = \partial Loss/\partial y * \partial y/\partial(W_o * h_k + b_o)$

$\partial Loss/\partial W_2 = \partial Loss/\partial y * \partial y/\partial(W_2 * h_1 + b_2) * \partial h_1/\partial(W_2 * h_1 + b_2) * \partial(W_2 * h_1 + b_2)/\partial W_2$

$\partial Loss/\partial b_2 = \partial Loss/\partial y * \partial y/\partial(W_2 * h_1 + b_2) * \partial h_1/\partial(W_2 * h_1 + b_2) * \partial(W_2 * h_1 + b_2)/\partial b_2$

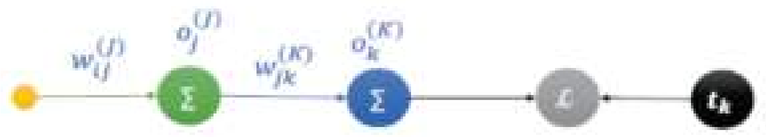
...

The chain rule is repeatedly applied for each layer to calculate the gradients.

Updating Parameters:

Once you have computed the gradients of the loss with respect to the parameters, you can use them to update the parameters through an optimization algorithm like gradient descent. The goal is to adjust the parameters to minimize the loss, thereby improving the model's performance.

In summary, the chain rule is a fundamental concept in gradient propagation within neural networks. It enables the efficient calculation of gradients for each layer, allowing the network to learn and adapt its parameters during training. This process is critical for the successful training of machine learning models, including deep neural networks.



). *Gradient propagation illustration*

BACK PROPOGATION ALGORITHM:

Backpropagation, short for "backward propagation of errors," is an algorithm used to train artificial neural networks, including deep learning models. It is a key component of the training process and is responsible for updating the model's weights to minimize the error between predicted and actual values. Here is a step-by-step explanation of the backpropagation algorithm:

Step 1: Initialize Weights and Biases

Initialize the weights and biases of the neural network. These values are typically initialized randomly.

Step 2: Forward Pass

Input data is fed forward through the network layer by layer, from the input layer to the output layer.

For each layer:

Calculate the weighted sum of inputs for each neuron in the layer.

Apply the activation function to the weighted sum to get the output of each neuron.

Step 3: Compute Loss

Calculate the loss (error) between the predicted output and the actual target values using a suitable loss function (e.g., Mean Squared Error for regression or Cross-Entropy Loss for classification).

Step 4: Backward Pass (Backpropagation)

Compute the gradient of the loss with respect to the output layer's inputs. This gradient measures how much a small change in the output of the network affects the loss.

$\partial Loss/\partial output\_layer\_inputs$

Propagate this gradient backward through the network to calculate the gradients of the loss with respect to the weights and biases of each layer. This is done using the chain rule.

$\partial Loss/\partial weights$ and $\partial Loss/\partial biases$ for each layer

Update the weights and biases of each layer using an optimization algorithm like gradient descent. The goal is to adjust these parameters in a way that minimizes the loss.

Step 5: Repeat

Repeat steps 2-4 for a fixed number of iterations (epochs) or until the loss converges to a satisfactory level.

Step 6: Evaluate Model

After training, evaluate the model's performance on a separate validation dataset or test dataset to assess its generalization ability.

Step 7: Use the Model

Once the model is trained and evaluated, it can be used for making predictions on new, unseen data.

This was simple process… lets understand it in detail

Backpropagation is the core of how neural networks learn. Up until this point, you learned

that training a neural network typically happens by the repetition of the following three steps:

• Feedforward: get the linear combination (weighted sum), and apply the activation

function to get the output prediction ($\hat{y}$):

$\hat{y} = \sigma \cdot W(3) \cdot \sigma \cdot W(2) \cdot \sigma \cdot W(1) \cdot (x)$

• Compare the prediction with the label to calculate the error or loss function:

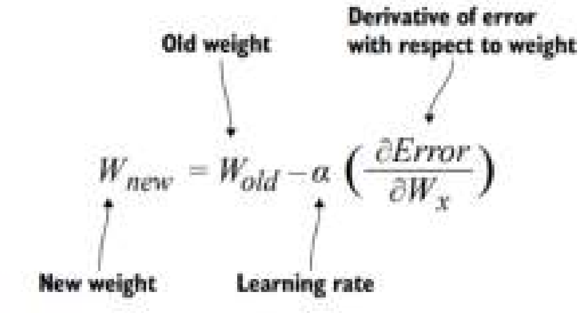
$E(W, b) = |\hat{y}i - yi|$

• Use a gradient descent optimization algorithm to compute the Δw that optimizes the

error function:

$\Delta wi = -\alpha \, dE/dw$

- Backpropagate the Δw through the network to update the weights:


Derivative of error with respect to weight
Old weight
$W_{new} = W_{old} - \alpha \left(\frac{\partial Error}{\partial W_x} \right)$
New weight
Learning rate

Backpropagation, or backward pass, means propagating derivatives of the error with respect to each specific weight dE/dWi .

from the last layer (output) back to the first layer (inputs) to adjust weights. By propagating the change in weights Δw backward from the prediction node (yˆ) all the way through the hidden layers and back to the input layer, the weights get updated:

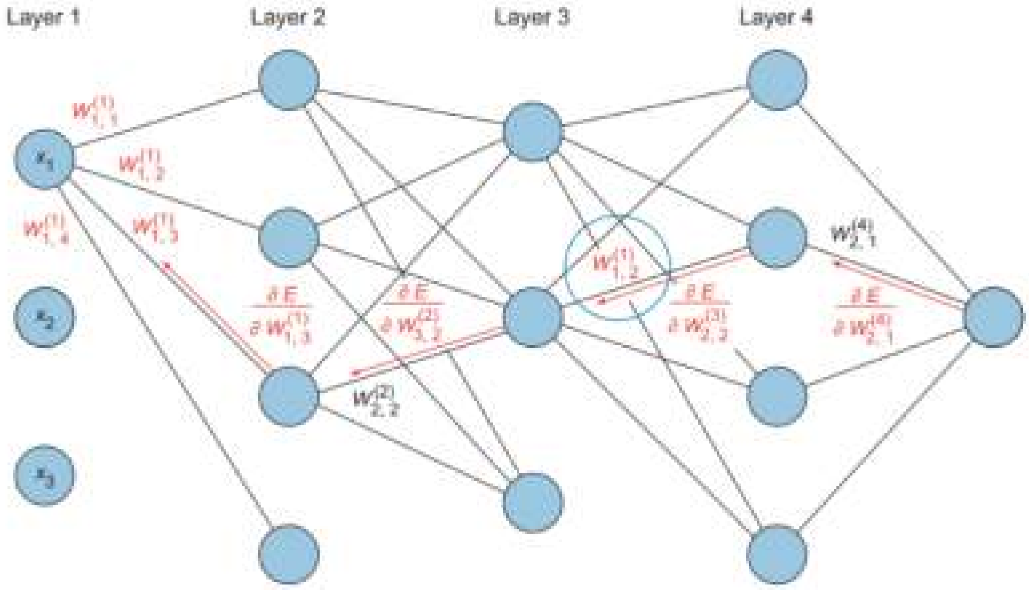
(wnext – step = wcurrent + Δw)

This will take the error one step down the error mountain. Then the cycle starts again (steps 1 to 3) to update the weights and take the error another step down, until we get to the minimum error.

Backpropagation might sound clearer when we have only one weight. We simply adjust the weight by adding the Δw to the old weight

wnew = w – α dE/dwi.

But it gets complicated when we have a multilayer perceptron (MLP) network with many weight variables. To make this clearer, consider the scenario in figure:


Layer 1
Layer 2
Layer 3
Layer 4
x_1
x_2
x_3
$w_{1,1}^{(1)}$
$w_{1,2}^{(1)}$
$w_{1,4}^{(1)}$
$w_{1,3}^{(1)}$
$\frac{\partial E}{\partial w_{1,3}^{(1)}}$
$\frac{\partial E}{\partial w_{3,2}^{(2)}}$
$w_{2,2}^{(2)}$
$w_{1,2}^{(1)}$
$\frac{\partial E}{\partial w_{2,2}^{(3)}}$
$w_{2,1}^{(4)}$
$\frac{\partial E}{\partial w_{2,1}^{(4)}}$

How do we compute the change of the total error with respect to dE/dw13 ?

How much will the total error change when we change the parameter w13?

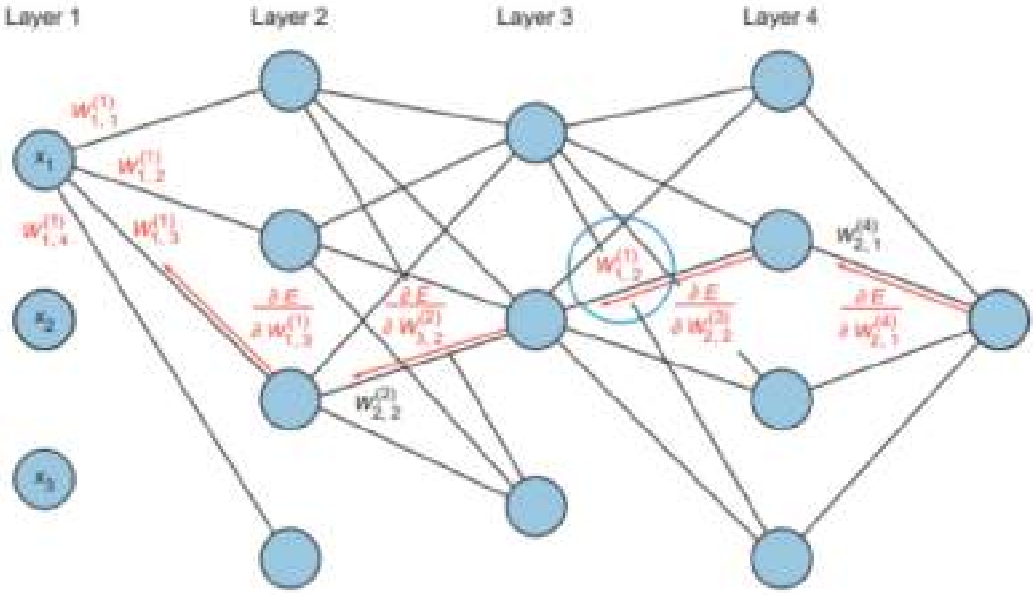
how to compute by applying the derivative rules on the error function.

That is straightforward because w21 is directly connected to the error function. But to compute the derivatives of the total error with respect to the weights all the way back to the input, we need a calculus rule called the chain rule.

Let’s apply the chain rule to calculate the derivative

of the error with respect to the third weight on the first input w1,3 (1) , where the (1) means layer 1, and w1,3 means node number 1 and weight number 3:

$$\frac{dE}{dw^{(1)}_{1,3}} = \frac{dE}{dw^{(4)}_{2,1}} \times \frac{dw^{(4)}_{2,1}}{dw^{(3)}_{2,2}} \times \frac{dw^{(3)}_{2,2}}{dw^{(2)}_{3,2}} \times \frac{dw^{(2)}_{3,2}}{dw^{(1)}_{1,3}}$$


Layer 1
Layer 2
Layer 3
Layer 4
x_1
x_2
x_3
$W^{(1)}_{1,1}$
$W^{(1)}_{1,2}$
$W^{(1)}_{1,4}$
$W^{(1)}_{1,3}$
$\frac{\partial E}{\partial W^{(1)}_{1,3}}$
$\frac{\partial E}{\partial W^{(2)}_{3,2}}$
$W^{(2)}_{2,2}$
$W^{(1)}_{1,2}$
$\frac{\partial E}{\partial W^{(3)}_{2,2}}$
$W^{(4)}_{2,1}$
$\frac{\partial E}{\partial W^{(4)}_{2,1}}$

The error back propagated to the

edge w1,3 (1) = effect of error on edge 4 · effect on edge 3 ·effect on edge 2 · effect on target edge .

Thus the backpropagation technique is used by neural networks to update the weights to solve

the best fit problem.

HANDS ON:

This example will demonstrate a single-layer neural network for a simple regression problem using backpropagation.

Let's start with a simple neural network with one neuron and one input feature. We'll implement forward and backward passes for training.

```
import numpy as np


# Define the sigmoid activation function and its derivative

def sigmoid(x):

    return 1 / (1 + np.exp(-x))
```

```
def sigmoid_derivative(x):
    return x * (1 - x)


# Define the neural network class
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases with random values
        self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

    def forward(self, x):
        # Forward pass
        self.hidden_input = np.dot(x, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)
        self.output = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        return self.output

    def backward(self, x, y, output):
        # Backpropagation
        self.output_error = y - output
        self.output_delta = self.output_error
        self.hidden_error = self.output_delta.dot(self.weights_hidden_output.T)
        self.hidden_delta = self.hidden_error * sigmoid_derivative(self.hidden_output)
        self.weights_hidden_output += self.hidden_output.T.dot(self.output_delta)
        self.weights_input_hidden += x.T.dot(self.hidden_delta)
        self.bias_output += np.sum(self.output_delta, axis=0)
        self.bias_hidden += np.sum(self.hidden_delta, axis=0)

    def train(self, x, y, epochs):
        for _ in range(epochs):
            output = self.forward(x)
            self.backward(x, y, output)

if __name__ == "__main__":
    # Define the dataset
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0], [1], [1], [0]])

    # Create and train the neural network
    input_size = 2
    hidden_size = 4
    output_size = 1
    nn = NeuralNetwork(input_size, hidden_size, output_size)

    epochs = 10000
    nn.train(X, y, epochs)

    # Test the trained network
    for i in range(len(X)):
        prediction = nn.forward(X[i])
        print("Input:", X[i], "Actual:", y[i], "Predicted:", prediction)
```

This code defines a simple neural network with one hidden layer and uses the sigmoid activation function. It trains the network using the XOR dataset.

EXAMPLE 2:

Here's an example of a four-layer fully connected neural network implemented in Python for binary classification using backpropagation. This network has two input nodes, three hidden layers with 20, 50, and 25 nodes respectively, and two output nodes for binary classification:

```
import numpy as np


# Define the sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):
    return x * (1 - x)


# Define the neural network class
class NeuralNetwork:
    def __init__(self, input_size, hidden_sizes, output_size):
        self.input_size = input_size
        self.hidden_sizes = hidden_sizes
        self.output_size = output_size
        self.num_layers = len(hidden_sizes) + 2  # Including input and output layers


        # Initialize weights and biases with random values
        self.weights = [np.random.rand(input_size, hidden_sizes[0])]
        self.biases = [np.zeros((1, hidden_sizes[0]))]


        for i in range(len(hidden_sizes) - 1):
```

```
        self.weights.append(np.random.rand(hidden_sizes[i], hidden_sizes[i+1]))
        self.biases.append(np.zeros((1, hidden_sizes[i+1])))

    self.weights.append(np.random.rand(hidden_sizes[-1], output_size))
    self.biases.append(np.zeros((1, output_size)))

def forward(self, x):
    self.layer_outputs = []
    input_layer = x

    # Forward pass through hidden layers
    for i in range(self.num_layers - 1):
        weighted_sum = np.dot(input_layer, self.weights[i]) + self.biases[i]
        layer_output = sigmoid(weighted_sum)
        self.layer_outputs.append(layer_output)
        input_layer = layer_output

    return input_layer

def backward(self, x, y, output):
    # Backpropagation
    deltas = [None] * self.num_layers
    error = y - output
    delta = error * sigmoid_derivative(output)
    deltas[-1] = delta

    # Calculate deltas for hidden layers
```

```
        for i in range(self.num_layers - 2, 0, -1):
            error = deltas[i + 1].dot(self.weights[i].T)
            delta = error * sigmoid_derivative(self.layer_outputs[i - 1])
            deltas[i] = delta

        # Update weights and biases
        for i in range(self.num_layers - 1):
            self.weights[i] += self.layer_outputs[i - 1].T.dot(deltas[i])
            self.biases[i] += np.sum(deltas[i], axis=0)

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            for i in range(len(X)):
                x = X[i]
                target = y[i]
                output = self.forward(x)
                self.backward(x, target, output)

    def predict(self, x):
        return self.forward(x)

if __name__ == "__main__":
    # Define the dataset
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0, 1], [1, 0], [1, 0], [0, 1]])  # One-hot encoded labels

    # Create and train the neural network
```

```
input_size = 2

hidden_sizes = [20, 50, 25]

output_size = 2

learning_rate = 0.1

epochs = 10000


nn = NeuralNetwork(input_size, hidden_sizes, output_size)

nn.train(X, y, epochs, learning_rate)


# Test the trained network

for i in range(len(X)):

    prediction = nn.predict(X[i])

    print("Input:", X[i], "Actual:", y[i], "Predicted:", prediction)
```

KERAS

INTRODUCTION:

- Keras is an open-source neural network computing library mainly developed in the Python language.
- It was originally written by François Chollet.
- It is designed as a highly modular and extensible high-level neural network interface, so that users can quickly complete model building and training without excessive professional knowledge.
- The Keras library is divided into a frontend and a backend. The backend generally calls the existing deep learning framework to implement the underlying operations, such as Theano, CNTK, and TensorFlow.
- The frontend interface is a set of unified interface functions abstracted by Keras. Users can easily switch between different backend operations through Keras.
- Since 2017, most components of Keras have been integrated into the TensorFlow framework.
- In 2019, Keras was officially identified as the only high-level interface API for TensorFlow 2, replacing the high-level interfaces such as tf.layers included in the TensorFlow 1. In other words, now you can only use the Keras interface to complete TensorFlow layer model building and training. In TensorFlow 2, Keras is implemented in the tf.keras submodule.

COMMON FUNCTIONAL MODULES:

- Keras is a popular deep learning framework that provides a high-level API for building and training neural networks. It offers a variety of common functional modules that you can use to construct neural network architectures.
- Keras provides a series of high-level neural network-related classes and functions, such as classic dataset loading function, network layer class, model container, loss function class, optimizer class, and classic model class.

Common Network Layer Classes:

- For the common neural network layer, we can use the tensor mode of the underlying interface functions to achieve, which are generally included in the tf.nn module.
- For common network layers, we generally use the layer method to complete the model construction. A large number of common network layers are provided in the tf.keras.layers namespace, such as fully connected layers, activation function layers, pooling layers, convolutional layers, and recurrent neural network layers.
- For these network layer classes, you only need to specify the relevant parameters of the network layer at the time of creation and use the __call__ method to complete the forward calculation. When using the __call__ method, Keras will automatically call the forward propagation logic of each layer, which is generally implemented in the call function of the class.

Examples:

Dense Layer: The Dense layer is a fully connected layer where every neuron in the layer is connected to every neuron in the previous layer. It's commonly used in feed forward neural networks and can be used for tasks like image classification and regression.

```
from keras.layers import Dense
dense_layer = Dense(units=64, activation='relu')
```

Convolutional Layer: The Conv2D and Conv3D layers are used for 2D and 3D convolution operations, respectively. They are essential for tasks like image and video analysis.

```
from keras.layers import Conv2D
conv_layer = Conv2D(filters=32, kernel_size=(3, 3), activation='relu')
```

Network Container:

For common networks, we need to manually call the class instance of each layer to complete the forward propagation operation. When the network layer becomes deeper, this part of the code appears very bloated. Multiple network layers can be encapsulated into a large network model through the network container Sequential provided by Keras. Only the instance of the network model needs to be called once to complete the sequential propagation operation of the data from the first layer to the last layer.

For example, the two-layer fully connected network with a separate activation function layer can be encapsulated as a network through the Sequential container.

```
from tensorflow.keras import layers, Sequential
network = Sequential([
layers.Dense(3, activation=None), # Fully-connected layer
without activation function
layers.ReLU(),# activation function layer
layers.Dense(2, activation=None), # Fully-connected layer
without activation function
layers.ReLU() # activation function layer
])
x = tf.random.normal([4,3])
out = network(x)
```

The Sequential container can also continue to add a new network layer through the add() method to dynamically create a network:

```
layers_num = 2
network = Sequential([]) # Create an empty container
for _ in range(layers_num):
network.add(layers.Dense(3)) # add fully-connected layer
network.add(layers.ReLU())# add activation layer
network.build(input_shape=(4, 4))
network.summary()
```

When we encapsulate multiple network layers through Sequential container, the parameter list of each layer will be automatically incorporated into the Sequential container.

MODEL CONFIGURATION, TRAINING AND TESTING IN KERAS

In Keras, you can create, configure, train, and test a neural network model by following a series of steps. Below is an overview of these steps:

1. **Model Creation and Configuration:**

Import necessary libraries and modules from Keras.

Define the model architecture, including layers, activation functions, and input/output shapes.

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

model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=feature_dim))
model.add(Dense(units=num_classes, activation='softmax'))
```

2. **Compile the Model:**

After defining the model, you need to configure its learning process by specifying the optimizer, loss function, and evaluation metrics.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

3. **Data Preparation:**

Prepare your training and testing data by loading, preprocessing, and splitting it into input and target (label) datasets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2,
random_state=42)
```

4. **Model Training:**

Train the model using the fit method. Pass in your training data (features and labels), batch size, number of epochs, and validation data if necessary.

```
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_test, y_test)).
```

5. **Model Evaluation:**

After training, you can evaluate the model's performance on the test dataset using the evaluate method.

```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}")
```

6. **Making Predictions:**

   Use the trained model to make predictions on new or unseen data.

   ```
   predictions = model.predict(X_new_data)
   ```

7. **Save and Load Models (Optional):**

   You can save your trained model to disk for later use and load it when needed.

   ```
   model.save("my_model.h5") loaded_model =
   keras.models.load_model("my_model.h5")
   ```

8. **Visualization (Optional):**

   You can visualize training history and performance using libraries like Matplotlib.

   ```
   import matplotlib.pyplot as plt plt.plot(history.history['loss'], label='Training Loss')
   plt.plot(history.history['val_loss'], label='Validation Loss') plt.legend() plt.show()
   ```

MODEL SAVING AND LOADING

1. **TENSOR METHOD:**

   In TensorFlow, you can save and load models using the tf.saved_model method. This method provides a standard way to save and load models, making it compatible with TensorFlow Serving and other TensorFlow-based deployment environments. Here's how you can save and load a model using the tf.saved_model method:

Saving a Model:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Create a simple Sequential model
model = Sequential([
    Dense(units=64, activation='relu', input_dim=feature_dim),
    Dense(units=num_classes, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_test, y_test))

# Save the model using the tf.saved_model method
model.save('my_model')
 # 'my_model' is the directory where the model will be saved
```

Loading a Model:

```
import tensorflow as tf

# Load the model using the tf.saved_model.load method
loaded_model = tf.saved_model.load('my_model')

# Make predictions using the loaded model
predictions = loaded_model(X_new_data)
 # X_new_data is your new data for inference.
```

In this example:

We first create a simple Sequential model and train it.

We save the model using model.save('my_model'), which will create a directory named 'my_model' containing the saved model artifacts.

To load the model, we use tf.saved_model.load('my_model'). This will load the model and return a callable object that you can use for making predictions.

Note that the model will be saved as a TensorFlow SavedModel, which is a directory containing the model's architecture, weights, and other metadata. This format is designed for easy deployment and compatibility with TensorFlow Serving.

Make sure to replace 'my_model' with the path where you want to save or load your model. You can also specify different versions of the model by using different directory names when saving.

2. USING NETWORK METHOD:

Let’s introduce a method that does not require network source files and only needs model parameter files to recover the network model.The model structure and model parameters can be saved to the path file through the Model.save(path) function, and the network structure and network parameters can be restored through keras.models.load_model(path) without the need for network source files .

First, save the MNIST handwritten digital picture recognition model to a file, and delete the network object:

```
# Save model and parameters to a file
network.save('model.h5')
print('saved total model.')
del network # Delete the network
```

The structure and state of the network can be recovered through the model.h5 file, and there is no need to create network objects in advance.

The code is as follows:

```
# Recover the model and parameters from a file
network = keras.models.load_model('model.h5')
```

In addition to storing model parameters, the model. h5 file should also save network structure information. You can directly recover the network object from the file without creating a model in advance.

3. SAVED MODEL METHOD:

TensorFlow is favored by the industry, not only because of the excellent neural network layer API support, but also because it has powerful ecosystem, including mobile and web support. When the model needs to be deployed to other platforms, the SavedModel method proposed by TensorFlow is platform-independent.

By tf.saved_model.save(network, path), the model can be saved to the path directory as follows:

```
# Save model and parameters to a file
tf.saved_model.save(network, 'model-savedmodel')
print('saving savedmodel.')
del network # Delete network object
```

After recovering the model instance, we complete the calculation of the test accuracy rate and achieve the following:

```
print('load savedmodel from file.')
# Recover network and parameter from files
network = tf.saved_model.load('model-savedmodel')
# Accuracy metrics
acc_meter = metrics.CategoricalAccuracy()
for x,y in ds_val: # Loop through test dataset
pred = network(x) # Forward calculation
acc_meter.update_state(y_true=y, y_pred=pred)
# Update stats
# Print accuracy
print("Test Accuracy:%f" % acc_meter.result())
```

CUSTOM NETWORK LAYERS:

In Keras, you can create custom neural network architectures by subclassing the keras.Model class. This approach allows you to define your own forward pass logic and create highly customized network architectures.

Here's how to create a custom neural network in Keras:

#Import the necessary libraries:

```
import tensorflow as tf
from tensorflow.keras.layers import Layer
```

Define a custom layer or set of layers. You can do this by creating a class that inherits from keras.layers.Layer. Implement the **__init__ method** to define layer parameters and the call method to specify the layer's forward pass.

For example, here's how you can define a custom layer:

```
class CustomLayer(Layer):
    def __init__(self, num_units, activation='relu'):
        super(CustomLayer, self).__init__()
        self.num_units = num_units
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape):
        # Define layer variables and weights here
        self.kernel = self.add_weight("kernel", shape=[input_shape[-1], self.num_units])
        self.bias = self.add_weight("bias", shape=[self.num_units])

    def call(self, inputs):
        # Define the layer's forward pass logic
        return self.activation(tf.matmul(inputs, self.kernel) + self.bias)
```

Create a custom model by subclassing keras.Model and define the architecture by composing custom layers.

```
class CustomModel(tf.keras.Model):
    def __init__(self):
        super(CustomModel, self).__init__()
        self.layer1 = CustomLayer(num_units=64, activation='relu')
        self.layer2 = CustomLayer(num_units=10, activation='softmax')

    def call(self, inputs):
        x = self.layer1(inputs)
        return self.layer2(x)
```

Compile the custom model with an optimizer, loss function, and metrics.

```
model = CustomModel()
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Train the model using your custom data and fit it to your training data:

```
model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(X_val,
y_val))
```

Evaluate and use the custom model just like any other Keras model:

```
loss, accuracy = model.evaluate(X_test, y_test)
predictions = model.predict(X_new_data)
```

MODEL ZOO:

For commonly used network models, such as ResNet and VGG, you do not need to manually create them. They can be implemented directly with the keras.applications submodule with a line of code. At the same time, you can also load pre-trained models by setting the weights parameters.

For an example, use the Keras model zoo to load the pre-trained ResNet50 network by ImageNet. The code is as follows:

```
# Load ImageNet pre-trained network. Exclude the last layer.
resnet = keras.applications.ResNet50(weights='imagenet',include_top=False)
resnet.summary()
# test the output
x = tf.random.normal([4,224,224,3])
out = resnet(x) # get output
out.shape
```

For a specific task, we need to set a custom number output nodes. Taking 100 classification tasks as an example, we rebuild a new network based on ResNet50. Create a new pooling layer (the pooling
layer here can be understood as a function of downsampling in high and wide dimensions) and reduce the features dimension from [*b*, 7, 7, 2048] to [*b*, 2048] as in the following.

```
# New pooling layer
global_average_layer = layers.GlobalAveragePooling2D()
# Use last layer's output as this layer's input
x = tf.random.normal([4,7,7,2048])
# Use pooling layer to reduce dimension from [4,7,7,2048] to
[4,1,1,2048],and squeeze to [4,2048]
out = global_average_layer(x)
print(out.shape)
Out[6]: (4, 2048)
```

Finally, create a new fully connected layer and set the number of output nodes to 100. The code is as follows:

In [7]:

```
# New fully connected layer
fc = layers.Dense(100)
# Use last layer's output as this layer's input
x = tf.random.normal([4,2048])
out = fc(x)
print(out.shape)
```

```
Out[7]: (4, 100)
```

After creating a pre-trained ResNet50 feature sub-network, a new pooling layer, and a fully connected layer, we re-use the Sequential container to encapsulate a new network:

```
# Build a new network using previous layers
mynet = Sequential([resnet, global_average_layer, fc])
mynet.summary()
```

You can see the structure information of the new network model is:

```
Layer (type) Output
Shape Param Number
=================================================================
resnet50 (Model) (None, None, None, 2048) 23587712
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048) 0
_________________________________________________________________
dense_4 (Dense) (None, 100) 204900
=================================================================
Total params: 23,792,612
Trainable params: 23,739,492
Non-trainable params: 53,120
```

By setting resnet.trainable = False, you can choose to freeze the network parameters of the ResNet part and only train the newly created network layer, so that the network model training can be completed quickly and efficiently. Of course, you can also update all the parameters of the network.

<u>METRICS:</u>

In Keras, metrics are used to evaluate the performance of machine learning or deep learning models. Metrics provide quantitative measures that help assess how well a model is performing on a given task, such as classification or regression. Keras provides a wide range of built-in metrics that you can use during model training and evaluation. Here are some common metrics available in Keras:

- Accuracy ('accuracy'): Measures the proportion of correctly classified samples in classification tasks. It's commonly used for classification problems with balanced classes.

- Precision ('precision'): Calculates the ratio of true positives to the sum of true positives and false positives. It is used to assess how many of the positively predicted instances were actually positive. Useful in situations where false positives are costly.

- Recall ('recall') or Sensitivity ('sensitivity') or True Positive Rate ('tp_rate'): Measures the ratio of true positives to the sum of true positives and false negatives. It quantifies how many of the actual positive instances were correctly predicted as positive. Important when missing positive cases is costly.

- F1-Score ('f1'): Harmonic mean of precision and recall. Useful when there is an imbalance between classes and you want to balance precision and recall.

- AUC-ROC ('roc_auc'): Area under the Receiver Operating Characteristic (ROC) curve. It's used to evaluate binary classification models and measures the model's ability to distinguish between positive and negative samples.

- Mean Absolute Error ('mae'): Measures the average absolute difference between predicted and actual values in regression tasks. It is robust to outliers.

- Mean Squared Error ('mse'): Measures the average squared difference between predicted and actual values in regression tasks. It penalizes larger errors more than MAE.

- Root Mean Squared Error ('rmse'): The square root of the MSE, which provides an error metric in the same units as the target variable.

- Mean Absolute Percentage Error ('mape'): Measures the average percentage difference between predicted and actual values in regression tasks. Useful when you want to assess the percentage error.

- Cosine Proximity ('cosine_proximity'): Measures the cosine similarity between predicted and actual values. Often used in recommendation systems.

- Cohen's Kappa ('kappa'): Measures the agreement between predicted and actual values while accounting for chance agreement. Useful for classification tasks when the class distribution is imbalanced.

- Top-K Categorical Accuracy ('top_k_categorical_accuracy'): Measures the accuracy of the top-k predictions in multi-class classification. It's often used when you care about whether the correct class is within the top-k predictions.

You can choose one or more of these metrics when compiling your Keras model using the model.compile() method.

CREATE A METRIC CONTAINER, WRITE DATA, READ STATISTICAL DATA, CLEAR THE CONTAINER, ACCURACY METRIC

In Keras, you can create a custom metric container, write data to it, read statistical data from it, clear the container, and implement an example metric like accuracy. Here's how you can do that:

Create a Metric Container:

You can create a custom metric container by subclassing keras.metrics.Metric and implementing the necessary methods.

```
import tensorflow as tf
from tensorflow.keras.metrics import Metric

class CustomMetric(Metric):
    def __init__(self, name='custom_metric', **kwargs):
        super(CustomMetric, self).__init__(name=name, **kwargs)
        self.values = self.add_weight(name='values', initializer='zeros')
        self.count = self.add_weight(name='count', initializer='zeros')
    def update_state(self, y_true, y_pred, sample_weight=None):
        # Custom metric logic: Update the values and count
        values = ...  # Calculate your custom metric value here
        self.values.assign_add(tf.reduce_sum(values))
        self.count.assign_add(tf.cast(tf.shape(y_true)[0], dtype=tf.float32))

    def result(self):
        # Calculate and return the final metric value
        return self.values / self.count
```

Write Data to the Metric Container:
To use your custom metric, you can instantiate it and then update its state by calling update_state() during model training or evaluation.

```
custom_metric = CustomMetric()

# During training or evaluation loop:
for batch in dataset:
    predictions = model(batch['input'])
    custom_metric.update_state(batch['target'], predictions)
```

Read Statistical Data:
You can read the statistical data from your custom metric by calling the result() method.

```
metric_value = custom_metric.result().numpy()
print(f'Custom Metric Value: {metric_value:.4f}')
```

Clear the Metric Container:
You can clear the metric container's state by calling the reset_states() method.

```
custom_metric.reset_states()
```

Accuracy Metric Example:
Here's how you can implement the accuracy metric as an example:

```
class AccuracyMetric(Metric):
    def __init__(self, name='accuracy', **kwargs):
        super(AccuracyMetric, self).__init__(name=name, **kwargs)
        self.correct_predictions = self.add_weight(name='correct_predictions', 
initializer='zeros')
        self.total_samples = self.add_weight(name='total_samples', initializer='zeros')

    def update_state(self, y_true, y_pred, sample_weight=None):
        # Calculate the number of correct predictions
        correct = tf.cast(tf.equal(tf.argmax(y_true, axis=-1), tf.argmax(y_pred, axis=-1)), 
dtype=tf.float32)
        self.correct_predictions.assign_add(tf.reduce_sum(correct))
        self.total_samples.assign_add(tf.cast(tf.shape(y_true)[0], dtype=tf.float32))

    def result(self):
        # Calculate and return accuracy
        return self.correct_predictions / self.total_samples
```

VISUALISATION IN KERAS : MODEL SIDE AND BROWSER SIDE:

Visualizing the performance and architecture of a Keras model can be done on both the model side (inside your Python script) and the browser side (using tools like TensorBoard). Here's how you can perform visualization in both contexts:

Model Side Visualization:

Plot Training History:

You can plot the training history of your Keras model directly within your Python script using libraries like Matplotlib. This allows you to visualize metrics like loss and accuracy during training.

```
import matplotlib.pyplot as plt
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_val, y_val))

# Plot training history
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.show()
```

Model Summary:

You can print a summary of your model's architecture to the console to understand its structure and the number of trainable parameters.

```
model.summary()
```

Browser Side Visualization (TensorBoard):

TensorBoard is a powerful tool provided by TensorFlow that allows you to visualize various aspects of your model, such as training metrics, model architecture, and even custom metrics. Here's how to use TensorBoard for visualization:

Install TensorFlow (if not already installed):

Make sure you have TensorFlow installed. You can install it using pip:

```
pip install tensorflow
```

Logging Metrics to TensorBoard: During model training, you can log metrics to TensorBoard using a Keras callback.
Here's an example:

```
from tensorflow.keras.callbacks import TensorBoard
tensorboard_callback = TensorBoard(log_dir='./logs', histogram_freq=1)
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_val, y_val), callbacks=[tensorboard_callback])
```

Start TensorBoard:

In your terminal, navigate to the directory where you're running your Python script and use the following command to start TensorBoard:

```
tensorboard --logdir=./logs
```

This command will start TensorBoard and provide a URL you can open in your browser to access the visualization.

Access TensorBoard in Your Browser:

Open a web browser and go to the URL provided by TensorBoard (typically, it's http://localhost:6006). Here, you can view various visualizations, including training metrics, model architecture, and more.

By combining model-side visualization in your Python script with browser-side visualization using TensorBoard, you can gain a comprehensive understanding of your Keras model's performance and architecture during training and evaluation.