UNIT 2

INTRODUCTION TO TENSORFLOW:

TensorFlow is an open-source machine learning framework developed by the Google Brain team. It is designed to facilitate the development and deployment of machine learning models, particularly neural networks. TensorFlow is one of the most popular and widely used deep learning frameworks, and it has gained significant traction in both the research and industry communities. Here are some key features and components of TensorFlow:

Tensor: The name "TensorFlow" comes from its core concept, the tensor. A tensor is a multi-dimensional array, similar to a matrix but with a higher number of dimensions. Tensors are fundamental to representing data in TensorFlow, making it well-suited for operations on multi-dimensional data, such as images, sequences, and more.

Flexibility: TensorFlow provides a flexible and comprehensive ecosystem for developing machine learning models. It supports various model types, including neural networks, decision trees, and k-means clustering, making it suitable for a wide range of applications.

Deep Learning: TensorFlow is particularly known for its strong support for deep learning, including deep neural networks (DNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). It provides high-level APIs like Keras that simplify the construction and training of deep learning models.

Scalability: TensorFlow is designed for scalability, allowing users to train and deploy models on a variety of hardware platforms, including CPUs, GPUs, and even distributed computing environments. This scalability makes it suitable for both small-scale and large-scale machine learning tasks.

Deployment: TensorFlow offers tools for deploying machine learning models to various platforms, including mobile devices and embedded systems, making it accessible for edge computing and real-time applications.

Community and Ecosystem: TensorFlow has a large and active community of developers and researchers who contribute to its growth. It also has a rich ecosystem of libraries, tools, and pre-trained models that can accelerate the development of AI and machine learning applications.

TensorFlow Serving: TensorFlow Serving is a dedicated component that simplifies the deployment of machine learning models in production. It allows models to be served as RESTful APIs, making it easier to integrate them into applications and services.

TensorBoard: TensorFlow includes a visualization tool called TensorBoard, which helps users visualize and understand the behavior of their models during training. It provides insights into model performance, graph structures, and more.

NUMERIC TENSOR:

A numeric tensor is the main data format of TensorFlow. According to the dimension, it can be divided into:

• Scalar: A single real number, such as 1.2 and 3.4, has a dimension of 0 and a shape of [].
• Vector: An ordered set of real numbers, wrapped by square brackets.
• Matrix: An ordered set of real numbers in n rows and m columns.
• Tensor: An array with dimension greater than 2. Each dimension of the tensor is also known as the axis. Generally, each dimension represents specific physical meaning.

In TensorFlow, scalars, vectors, and matrices are also collectively referred to as tensors without distinction. You need to make your own judgement based on the dimension or shape of tensors.

*Creating a Tensor x of the value [0,-2,-1],[0,1,2]:

```
#Importing required libraries:
import tensorflow as tf
import numpy as np
In[44]:
x=tf.constant([[0,-2,-1],[0,1,2]])
print(x)

Out[44]:
tf.Tensor( [[ 0 -2 -1] [ 0 1 2]], shape=(2, 3), dtype=int32)
```

***Creating a Tensor y as a tensor of zeroes with the same shape as that of tensor x.**

```
In[34]:
Y=tf.fill([2,3], 0)
print(Y)

Out[34]:
tf.Tensor( [[0 0 0] [0 0 0]], shape=(2, 3), dtype=int32)
```

***Return a Boolean tensor that yields True if x equals y element wise.**

```
In[46]:
tf.math.equal(x, Y)
Out[46]:
<tf.Tensor: shape=(2, 3), dtype=bool, numpy=
array([[ True, False, False],
 [ True, False, False]])>
```

In TensorFlow, you can create tensors in a variety of ways, such as from a Python list, from a Numpy array, or from a known distribution.

(i) **Create Tensors from Arrays and Lists:**
Numpy array and Python list are very important data containers in Python. Many data are loaded into arrays or lists before being converted to tensors. The output data of TensorFlow are also usually exported to arrays or lists, which makes them easy to use for other modules.
The tf.convert_to_tensor function can be used to create a new tensor from a Python list or Numpy array.

For example:
In [22]: # Create a tensor from a Python list
tf.convert_to_tensor([1,2.])
Out[22]:<tf.Tensor: id=86, shape=(2,), dtype=float32, numpy=array([1.,2.], dtype=float32)>
In [23]: # Create a tensor from a Numpy array
tf.convert_to_tensor(np.array([[1,2.],[3,4]]))
Out[23]:
<tf.Tensor: id=88, shape=(2, 2), dtype=float64, numpy=
array([[1., 2.], [3., 4.]])>

Note that Numpy floating-point arrays store data with 64-bit precision by default. When converting to a tensor type, the precision is tf.float64. You can convert it to tf.float32 when needed.

(ii)**Create All-0 or All-1 Tensors:**
Creating tensors with all 0s or 1s is a very common tensor initialization method. Consider linear transformation y = Wx + b. The weight matrix W can be initialised with a matrix of all 1s, and b can be initialised with a vector of all 0s. So the linear transformation changes to y = x. We can use tf.zeros() or tf.ones() to create all-zero or all-one tensors with arbitrary shapes.

***Create a vector of all 0s and all 1s:**
In [24]:
tf.zeros([]),tf.ones([])
Out[24]:
(<tf.Tensor: id=90, shape=(), dtype=float32, numpy=0.0>,<tf.Tensor: id=91, shape=(), dtype=float32, numpy=1.0>)

***Create a matrix of all 0's:**
In [26]: tf.zeros([2,2])
Out[26]:
<tf.Tensor: id=104, shape=(2, 2), dtype=float32, numpy=
array([[0., 0.],

[0., 0.]], dtype=float32)>

***Create a matrix of all 1's:**

```
In [26]: tf.ones([2,2])
Out[26]:
<tf.Tensor: id=108, shape=(2, 2), dtype=float32, numpy=
array([[1., 1.],
       [1., 1.]], dtype=float32)>
```

With tf.zeros_like and tf.ones_like, you can easily create a tensor with all 0s or 1s that is consistent with the shape of another tensor.

(iii)Create a Customized Numeric Tensor:

In addition to initializing a tensor with all 0s or 1s, sometimes it is also necessary to initialize the tensor with a specific value, such as –1. With tf.fill(shape, value), we can create a tensor with a specific numeric value, where the dimension is specified by the shape parameter. For example, here's how to create a scalar with element –1:

```
In [30]:
    tf.fill([], -1)
Out[30]:
    <tf.Tensor: id=124, shape=(), dtype=int32, numpy=-1>
```

Create a vector with all elements –1:

```
In [31]:
    tf.fill([1], -1)
Out[31]:
    <tf.Tensor: id=128,  shape=(1,), dtype=int32, numpy=array([-1])>
```

(iv)Create a Tensor from a Known Distribution:

Sometimes, it is very useful to create tensors sampled from common distributions such as normal (or Gaussian) and uniform distributions. For example, in convolutional neural networks, the convolution kernel W is usually initialized from a normal distribution to facilitate the training process. In adversarial networks, hidden variables z are generally sampled from a uniform distribution

With tf.random.normal(shape, mean=0.0, stddev=1.0), we can create a tensor with dimension defined by the shape parameter and values sampled from a normal distribution N(mean,stddev^2). For example, here's how to create a tensor from a normal distribution with mean 0 and standard deviation of 1:

```
In [33]: tf.random.normal([2,2]) # Create a 2x2 tensor from a
```

normal distribution
Out[33]:
<tf.Tensor: id=143, shape=(2, 2), dtype=float32, numpy=
array([[-0.4307344 , 0.44147003], [-0.6563149 , -0.30100572]],
dtype=float32)>

***Create a tensor from a normal distribution with mean of 1 and standard deviation of 2:**
In [34]: tf.random.normal([2,2], mean=1,stddev=2)
Out[34]:
<tf.Tensor: id=150, shape=(2, 2), dtype=float32, numpy=
array([[-2.2687864, -0.7248812],
[1.2752185, 2.8625617]], dtype=float32)>
With tf.random.uniform(shape, minval=0, maxval=None, dtype=tf. float32), we can create a uniformly distributed tensor sampled from the interval [minval,maxval). For example, here's how to create a matrix uniformly sampled from the interval [0, 1) with shape of [2, 2]:
In [35]: tf.random.uniform([2,2])
Out[35]:
<tf.Tensor: id=158, shape=(2, 2), dtype=float32, numpy=array([[0.65483284, 0.63064325], [0.008816 , 0.81437767]], dtype=float32)>
Create a matrix uniformly sampled from an interval [0, 10) with shape of [2, 2]:
In [36]: tf.random.uniform([2,2],maxval=10)
Out[36]:
<tf.Tensor: id=166, shape=(2, 2), dtype=float32, numpy=
array([[4.541913 , 0.26521802], [2.578913 , 5.126876]], dtype=float32)>

If we need to uniformly sample integers, we must specify the maxval parameter and set the data type as tf.int*.
Notice that these outputs from all random functions may be distinct. However, it does not affect the usage of these functions.

LET US NOW , Create a tensor x of the value
(29.05088806627.61298943 31.19073486, 29.35532951
0.97266006 26.67541885. 38.08450317, 20.74983215,
4.94445419. 34.45999146 29.06485367, 36.01657104
27.88236427. 20.56035233. 30.20379066, 29.51215172.
33.71149445, 28.59134293, 36.05556488, 28.66994858]
Get the indices of elements in x whose values are greater than 30 and then extract those elements.
In[1]:
import tensorflow as tf

```
array=tf.constant([29.05088806,27.61298943 ,31.19073486, 29.35532951,
0.97266006
        ,26.67541 20.74983215,4.94445419 ,34.45999146, 29.06485367,
        36.01657104,27.882364 29.51215172 ,33.71149445, 28.59134293,
        36.05556488, 28.66994858])  # Create array

In[2]:
print(array)
Out[2]:
tf.Tensor( [29.050888 27.61299 31.190735 29.35533 0.97266006 26.675419
38.084503
 20.749832 4.944454 34.45999 29.064854 36.01657 27.882364 20.560352
30.20379 29.512152 33.711494 28.591343 36.055565 28.669949 ],
shape=(20,), dtype=float32)

In[3]:
for idx, elem in enumerate(array):
        if(elem>30.0):
                tf.print(idx, elem)
Out[3]:
2 31.1907349
6 38.0845032
9 34.4599915
11 36.016571
14 30.2037907
16 33.7114944
18 36.0555649
```

APPLICATIONS OF TENSOR

APPLICATIONS OF TENSORS:

Tensors have a vast application in physics and mathematical geometry. The mathematical explanation of electromagnetism is also defined by tensors. The vector analysis acts as a primer in tensor analysis and relativity. Elasticity, quantum theory, machine learning, mechanics, relativity are all affected by tensors.

(i) SCALAR :

Scalar is a simple number with 0 dimension and a shape of []. Typical uses of scalars are the representation of error values and various metrics, such as accuracy ,precision and recall.

Consider the training curve of a model. The x-axis is the number of training steps, and the y-axis is Loss per Query. Image error change (Figure 4-1 (a)) and accuracy change (Figure 4-1 (b)), where the loss value and accuracy are scalars generated by tensor calculation.

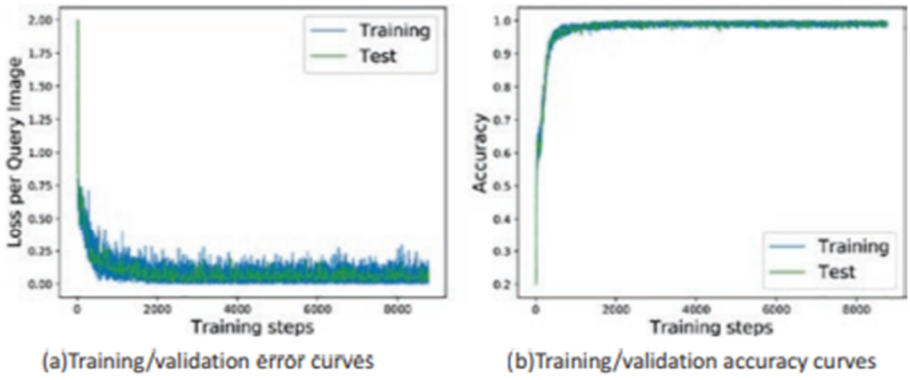


Figure 4-1. *Loss and accuracy curves*

Take the mean squared error function as an example. After tf.keras. losses.mse (or tf.keras.losses.MSE, the same function) returns the error value on each sample and finally takes the average value of the error as the error of the current batch, it automatically becomes a scalar:

In [41]:

```
out = tf.random.uniform([4,10])
# Create a model output example
y = tf.constant([2,3,2,0])
# Create a real observation
y = tf.one_hot(y, depth=10)
# one-hot encoding loss = tf.keras.losses.mse(y, out)
# Calculate MSE for each sample loss = tf.reduce_mean(loss)
```

Calculate the mean of MSE print(loss)
Out[41]: tf.Tensor(0.19950335, shape=(), dtype=float32)

(ii)VECTOR:
Vectors are very common in neural networks. For example, in fully connected networks and convolutional neural networks, bias tensors b are represented by vectors. As shown in Figure 4-2, a bias value is added to the output nodes of each fully connected layer, and the bias of all output nodes is represented as a vector form b = [b1,b2]^T:

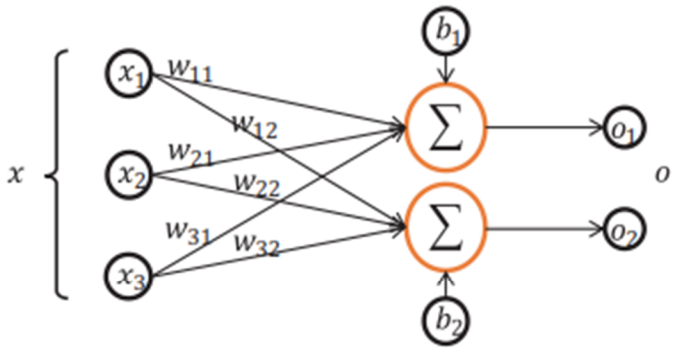


Figure 4-2. *Application of bias vectors*

Considering a network layer of two output nodes, we create a bias vector of length 2 and add back on each output node:
In [42]:
# Suppose z is the output of an activation function
z = tf.random.normal([4,2])
b = tf.zeros([2])
# Create a bias vector
z = z + b
Out[42]:
<tf.Tensor: id=245, shape=(4, 2), dtype=float32, numpy=
array([[0.6941646 , 0.4764454],
[-0.34862405, -0.26460952],
[1.5081744 , -0.6493869],
[-0.26224667, -0.78742725]], dtype=float32)>

For a network layer created through the high-level interface class Dense(), the tensors W and b are automatically created and managed by the class internally. The bias variable b can be accessed through the bias member of the fully connected layer. For example, if a linear network layer with four input nodes

and three output nodes is created, then its bias vector b should have length of 3 as follows:
In [43]:
```
fc = layers.Dense(3)
 # Create a dense layer with output length of 3
# Create W and b through build function with input nodes of 4
fc.build(input_shape=(2,4))
 fc.bias # Print bias vector
```
Out[43]:
```
<tf.Variable 'bias:0' shape=(3,) dtype=float32,
numpy=array([0., 0., 0.], dtype=float32)>
```

It can be seen that the bias member of the class is a vector of length 3 and is initialised to all 0s. This is also the default initialization scheme of the bias b. Besides, the type of the bias vector is Variable, because gradient information is needed for both W and b.

(iii)MATRIX:
A matrix is also a very common type of tensor. For example, the shape of a batch input tensor X of a fully connected layer is [b,din], where b represents the number of input samples, that is, batch size, and din represents the length of the input feature. For example, the feature length 4 and the input containing a total of two samples can be expressed as a matrix:

```
x = tf.random.normal([2,4]) # A tensor with 2 samples and 4 features
```

Let the number of output nodes of the fully connected layer be three and then the shape of its weight tensor W [4,3]. We can directly implement a network layer using the tensors X, W and vector b. The code is as follows:
In [44]:
```
w = tf.ones([4,3])
b = tf.zeros([3])
o = x@w+b # @ means matrix multiplication
```
Out[44]: <tf.Tensor: id=291, shape=(2, 3), dtype=float32, numpy=
array([[2.3506963, 2.3506963, 2.3506963], [-1.1724043, -1.1724043, -1.1724043]], dtype=float32)

In the preceding code, both X and W are matrices. The preceding code implements a linear transformation network layer, and the activation function is empty. In general, the network layer $\sigma(X @ W + b)$ is called a fully connected layer, which can be directly implemented by the Dense() class in TensorFlow.

In particular, when the activation function σ is empty, the fully connected layer is also called a linear layer. We can create a network layer with four input nodes and three output nodes through the Dense() class and view its weight matrix W through the kernel member of the fully connected layer:
In [45]:

```
 fc = layers.Dense(3)
# Create fully-connected layer with 3 output nodes
fc.build(input_shape=(2,4))
# Define the input nodes to be 4
 fc.kernel # Check kernel matrix W
```

Out[45]:

```
<tf.Variable 'kernel:0' shape=(4, 3) dtype=float32, numpy=array([[ 0.06468129,
-0.5146048 , -0.12036425], [ 0.71618867, -0.01442951, -0.5891943 ], [-
0.03011459, 0.578704 , 0.7245046 ], [ 0.73894167, -0.21171576, 0.4820758 ]],
dtype=float32)>
```

(iv)Three-Dimensional Tensor:
A typical application of a three-dimensional tensor is to represent a sequence signal. Its format is:

X=[b, sequence length, feature length]

where the number of sequence signals is b, sequence length represents the number of sampling points or steps in the time dimension, and feature length represents the feature length of each point.

In order to facilitate the processing of strings by neural networks, words are generally encoded into vectors of fixed length through the embedding layer. For example, "a" is encoded as a vector of length 3. Then two sentences with equal length (each sentence has five words) can be expressed as a three-dimensional tensor with shape of [2,5,3], where 2 represents the number of sentences, 5 represents the number of words, and 3 represents the length of the encoded word vector. We demonstrate how to represent sentences through the IMDB dataset as follows:
In [46]:

```
# Load IMDB dataset
 from tensorflow import keras
 (x_train,y_train),(x_test,y_test)=keras.datasets.imdb.load_
data(num_words=10000)
 # Convert each sentence to length of 80 words
 x_train = keras.preprocessing.sequence.pad_sequences(x_ train,maxlen=80)
 x_train.shape
```

Out [46]: (25000, 80)

We can see that the shape of the x_train is [25000,80], where 25000 represents the number of sentences, 80 represents a total of 80 words in each sentence, and each word is represented by a numeric encoding method. Next, we use the layers.Embedding function to convert each numeric encoded word into a vector of length 100:

```
In [47]: # Create Embedding layer with 100 output length
embedding=layers.Embedding(10000, 100)
# Convert numeric encoded words to word vectors
 out = embedding(x_train) out.shape
Out[47]: TensorShape([25000, 80, 100])
```

Through the embedding layer, the shape of the sentence tensor becomes [25000,80,100], where 100 represents that each word is encoded as a vector of length 100.

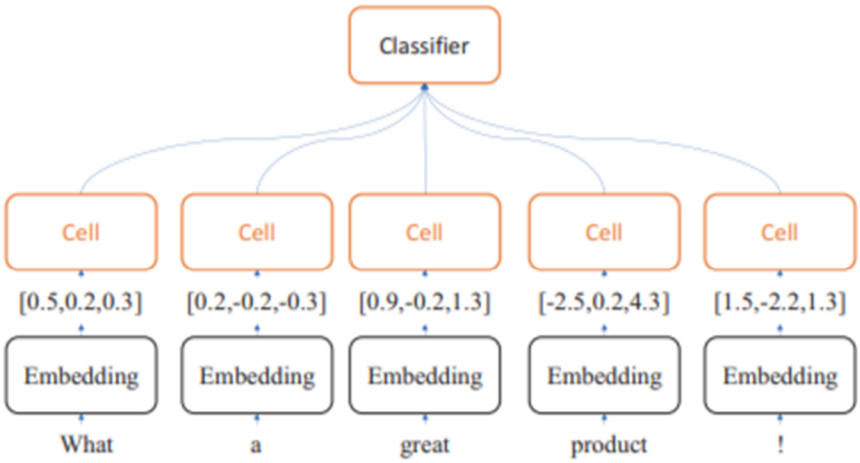


Figure 4-3. *Sentiment classification network*

For a sequence signal with one feature, such as the price of a product within 60 days, only one scalar is required to represent the product price, so the price change of two products can be expressed using a tensor of shape [2,60]. In order to facilitate the uniform format, the price change can also be expressed as a tensor of shape [2,60,1], where 1 represents the feature length of 1.

(iv)Four-Dimensional Tensor:

Most times we only use tensors with dimension less than five. For larger dimension tensors, such as five-dimensional tensor representation in meta learning, a similar principle can be applied. Four-dimensional tensors are widely used in convolutional neural networks. They are used to save feature maps. The format is generally defined as :

[b,h,w,c]

where b indicates the number of input samples; h and w represent the height and width of the feature map, respectively; and c is the number of channels. Some deep learning frameworks also use the format of [b, c,h,w], such as PyTorch. Image data is a type of feature map. A color image with three channels of RGB contains h rows and w columns of pixels. Each point requires three values to represent the color intensity of the RGB channel, so a picture can be expressed using a tensor of shape [h,w, 3]. As shown in Figure 4-4, the top picture represents the original image, which contains the intensity information of the three lower channels

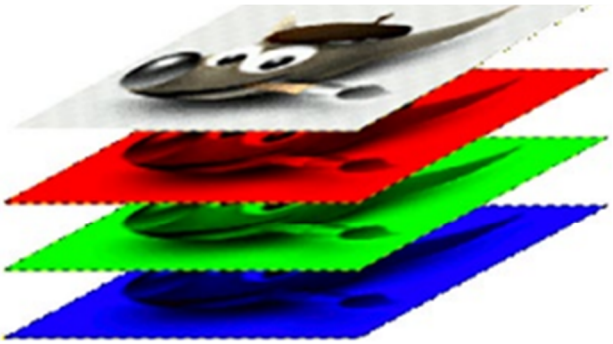


Figure 4-4. *Feature maps of RGB images*

In neural networks, multiple inputs are generally calculated in parallel to improve the computation efficiency, so the tensor of b pictures can be expressed as [b,h,w, 3]:

```
In [48]: # Create 4 32x32 color images
 x = tf.random.normal([4,32,32,3])
# Create convolutional layer
 layer = layers.Conv2D(16,kernel_size=3)
out = layer(x) out.shape
Out[48]: TensorShape([4, 30, 30, 16])
```

The convolution kernel tensor is also a four-dimensional tensor, which can be accessed through the kernel member variable: In [49]: layer.kernel.shape
Out[49]: TensorShape([3, 3, 3, 16])

BROADCASTING IN TENSORFLOW:

Broadcasting is a lightweight tensor copying method, which logically expands the shape of the tensor data, but only performs the actual storage copy operation when needed.The core idea of the broadcasting mechanism is universality. That is, the same data can be generally suitable for other locations.

For all dimensions of length 1, broadcasting has the same effect as tf.tile. The difference is that tf.tile creates a new tensor by performing the copy IO

operation. Broadcasting does not immediately copy the data; instead, it will logically change the shape of the tensor, so that the view becomes the copied shape. Broadcasting will use the optimization methods of the deep learning framework to avoid the actual copying of data and complete the logical operations.
The broadcasting mechanism saves a lot of computational resources. It is recommended to use broadcasting as much as possible in the calculation process to improve efficiency.

Continuing to consider the preceding example Y = X @ W + b, the shape of X @ W is [2, 3], and the shape of b is [3]. We can manually complete the copy data operation by combining tf.expand_dims and tf.tile, that is, transform b to shape [2, 3] and then add it to X @ W. But in fact, it is also correct to add X @ W directly to b with shape [3], for example:

```
x = tf.random.normal([2,4])
w = tf.random.normal([4,3])
 b = tf.random.normal([3])
y = x@w+b # Add tensors with different shapes directly
```

The preceding addition does not throw a logical error. This is because it automatically calls the broadcasting function tf.broadcast_to(x, new_ shape), expanding the shape of b to [2,3]. The preceding operation is equivalent to:

```
y = x@w + tf.broadcast_to(b,[2,3])
```

In other words, when the operator + encounters two tensors with inconsistent shapes, it will automatically consider expanding the two tensors to a consistent shape and then call tf.add to complete the tensor addition operation. By automatically calling tf.broadcast_to(b, [2,3]), it not only achieves the purpose of increasing dimension but also avoids the expensive computational cost of actually copying the data.

Before verifying universality, we need to align the tensor shape to the right first and then perform universality check: for a dimension of length 1, by default this data is generally suitable for other positions in the current dimension; for dimensions that do not exist, after adding a new dimension, the default current data is also universally applicable to the new dimension, so that it can be expanded into a tensor shape of any number of dimensions.

NUMERIC PRECISION:

In TensorFlow, as in most programming and machine learning libraries, numeric precision plays a crucial role in determining the accuracy and behavior of machine learning models. TensorFlow provides various options for specifying and controlling numeric precision, which is particularly important when working with neural networks and deep learning models. Here are some key aspects of numeric precision in TensorFlow:

Data Types: TensorFlow supports various data types for representing numerical values. The most common data types for representing real numbers are tf.float32 and tf.float64. The choice of data type affects the precision and memory usage of your computations. For example, tf.float32 represents numbers with 32-bit precision, while tf.float64 uses 64-bit precision, offering higher precision but consuming more memory.

Mixed Precision: In some deep learning applications, mixed-precision training is used to strike a balance between speed and precision. Mixed-precision training typically involves using 16-bit floating-point numbers (tf.float16) for most computations while maintaining some critical parts of the model in higher precision (e.g., tf.float32) to avoid loss of information during training. This approach can significantly accelerate training while preserving model accuracy.

Quantization: Quantization is a technique used to reduce the memory and computational requirements of deep learning models by representing weights and activations using lower-precision integers instead of floating-point numbers. TensorFlow provides tools and libraries for quantizing models to various levels of precision, such as 8-bit integers.

Custom Numeric Precision: TensorFlow allows you to define custom data types and numeric precision settings for specialized use cases. This can be particularly useful when working with hardware accelerators like GPUs or TPUs, where different numeric precision formats may be supported.

Numerical Stability: TensorFlow includes functions and techniques to enhance numerical stability in deep learning models. This includes using specialized activation functions (e.g., ReLU variants), weight initialization methods, and gradient clipping to mitigate issues like vanishing gradients and exploding gradients that can affect model training.

Hardware Considerations: The choice of hardware (CPU, GPU, TPU) can impact the numeric precision available for computations. For example, GPUs and TPUs often support hardware-accelerated operations in lower-precision formats like tf.float16.

OPERATIONS ON TENSORS:

Operations on tensors refer to various mathematical operations that can be performed on tensors, which are multi-dimensional arrays of numbers. These operations are fundamental in fields such as mathematics, physics, computer science, and machine learning. Some common operations on tensors include:

- Addition and Subtraction: Tensors of the same shape can be added or subtracted element-wise.

- Scalar Multiplication: Multiplying a tensor by a scalar involves multiplying every element of the tensor by that scalar.

- Element-wise Multiplication: Also known as the Hadamard product, this operation involves multiplying tensors element-wise. The tensors involved must have the same shape.

- Matrix Multiplication: Involves multiplying two tensors to produce a new tensor according to specific rules. For example, the dot product of two vectors or the matrix product of two matrices.

- Transpose: This operation flips a tensor over its diagonal, meaning the rows become columns and vice versa.

- Reshaping: Changing the shape of a tensor while keeping the same number of elements. For example, converting a 3x3 matrix into a 1x9 vector.

- Reduction Operations: Operations that reduce the dimensionality of a tensor by aggregating its elements. Examples include summing along specific axes, finding the mean, maximum, minimum, etc.

- Tensor Contraction: This operation involves summing products of elements along specified axes of two tensors.

- Tensor Decomposition: Breaking down a tensor into a combination of simpler tensors. For instance, Singular Value Decomposition (SVD) or Eigenvalue Decomposition.

- Outer Product: This operation results in a tensor that is the direct product of two tensors, typically used to generate higher-rank tensors.

- Tensor Product: A generalization of the outer product that combines elements of two tensors to create a new tensor.

These operations are crucial in various mathematical contexts, including linear algebra, calculus, differential geometry, and in practical applications such as signal processing, image processing, deep learning, and physics simulations.

Lets see a few of them in detail….

Tensors

A Tensor is a multi-dimensional array. Similar to NumPy ndarray objects, tf.Tensor objects have a data type and a shape. Additionally, tf.Tensors can reside in accelerator memory (like a GPU). TensorFlow offers a rich library of operations (for example, tf.math.add, tf.linalg.matmul, and tf.linalg.inv) that consume and produce tf.Tensors. These operations automatically convert built-in Python types.

In[1]:

```
import tensorflow as tf
a = tf.constant([1, 2, 3, 4])  # Create tensor a
b = tf.constant([10, 21, 35, 46])  # Create tensor b
```

Operations on Tensors

1)Addition

Function:

tf.add(x, y, name=None) Adds two tensors

In[2]:

```
print("Shape of a:",a.shape) #shape of tensor a
print("Shape of b:",b.shape) #shape of tensor b
```

Out[2]:

```
Shape of a: (4,)
Shape of b: (4,)
```

Int[3]:

```
 x=tf.add(a,b) #Addition of tensors a and b
tf.print("Addition of two tensors: ",x)
```

Out[3]:

```
 Addition of two tensors: [11 23 38 50]
```

2)Subtraction

Function:
subtract(x, y, name=None) Subtracts two tensors

In[4]:
```
 x=tf.subtract(a,b)#Subtraction
 y=tf.subtract(b,a)
 tf.print("Subtraction of two tensors: ",x)
tf.print("Subtraction of two tensors: ",y)
```

Out[4]:
Subtraction of two tensors: [-9 -19 -32 -42]
Subtraction of two tensors: [9 19 32 42]

3)Multiplication
Function:
multiply(x, y, name=None) Multiplies two tensors

In[5]:
```
x=tf.multiply(a,b) #Multiplication of tensors
tf.print("Multiplication of two tensors: ",x)
```
Out[5]:
Multiplication of two tensors: [10 42 105 184]

4)Division
Function:
divide(x, y, name=None) Divides the elements of two tensors
(or)
div(x, y, name=None) Divides the elements of two tensors

In[6]:
```
 x=tf.divide(b,a)#Division of two tensors
tf.print("Division of two tensors: ",x)
```

Out[6]:
Division of two tensors: [10 10.5 11.666666666666666 11.5]

When operating on floating-point values, div and divide produce the same result. But for integer division, divide returns a floating-point result, and div returns an integer result. The following code demonstrates the difference between them:
```
a = tf.constant([3, 3, 3])
b = tf.constant([2, 2, 2])
```

```
div1 = tf.divide(a, b)          # [ 1.5 1.5 1.5 ]

div2 = a / b                    # [ 1.5 1.5 1.5 ]

div3 = tf.div(a, b)             # [ 1 1 1 ]

div4 = a // b                   # [ 1 1 1 ]
```

The div function and the / operator both perform element-wise division. In contrast, the divide function performs Python-style division.

5)Logarithm of a tensor(tf.log(x))

Computes natural logarithm of x element-wise.
Tensor x must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

In[7]:

```
a=tf.constant([0,0.5,1.2,2.5]) #dtype=float
x=tf.math.log(a)
tf.print(x)
```

Out[7]:

```
[-inf -0.693147182 0.182321593 0.91629076]
```

6)Exponent of a tensor (tf.exp(x))

Computes exponential of x element-wise.
Tensor x Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

In[8]:

```
a=tf.constant([0,0.5,1.2,2.5])
x=tf.math.exp(a)
tf.print(x)
```

Out[8]:

```
[1 1.64872122 3.320117 12.1824942] Double-click (or enter) to edit
```

Applications can perform identical operations by using regular Python operators, such as +, -, *, /, and //.

MERGING AND SPLITTING:

Merging means combining multiple tensors into one tensor in a certain dimension. Taking the data of a school's gradebooks as an example, tensor A is used to save the gradebooks of classes 1–4. There are 35 students in each class with a total of eight subjects. The shape of tensor A is [4,35,8]. Similarly, tensor B keeps the gradebooks of the other six classes, with a shape of [6,35,8]. By merging these two gradebooks, you can get the gradebooks of all the classes in the school, recorded as tensor C, and the corresponding shape should be [10,35,8], where 10 represents ten classes, 35 represents 35 students, and 8 represents eight subjects.

Tensors can be merged using concatenate and stack operations. The concatenate operation does not generate new dimensions. It only merges along existing dimensions. But the stack operation creates new dimensions. Whether to use the concatenate or stack operation to merge tensors depends on whether a new dimension needs to be created for a specific scene. We will discuss both of them in the following session.

8) Concatenate: In TensorFlow, tensors can be concatenated using the tf.concat(tensors, axis) function, where the first parameter holds a list of tensors that need to be merged and the second parameter specifies the dimensional index on which to merge. Back to the preceding example, we merge the gradebooks in the class dimension. Here, the index number of the class dimension is 0, that is, axis = 0. The code for merging A and B is as follows:

In[0]:

```
import tensorflow as tf
 a = tf.random.normal([4,35,8]) # Create gradebook A
b = tf.random.normal([6,35,8]) # Create gradebook B
tf.concat([a,b],axis=0) #Merging A and B along axis 0
```

Out[0]:

```
[[ 1.0221779 , -1.6211283 , 1.2535826 , ..., 0.8566991 , -0.72349036, -
1.1157284 ], [-1.2414012 , -1.8060911 , 0.75162935, ..., 0.3864265 , 1.7944626
, -0.22267966], [ 0.6155348 , 0.29014412, -1.2273517 , ..., -1.2903832 , -
0.18366472, 1.1815189 ], ..., [ 0.24838325, -0.73552257, 0.1266093 , ...,
1.0176892 , -1.2254806 , 0.30242777], [-0.31192377, -2.3388796 , -0.645616 ,
..., 1.4724126 , 0.40083158, -0.4169775 ], [-0.46578857, 1.282099 ,
0.40268826, ..., 1.1217413 , 1.2570121 , -0.36919504]], ..., [[ 0.04562129, -
1.7338026 , -0.47456488, ..., 1.5377275 , 0.4686792 , 0.49222776], [
0.87741333, -1.8917205 , 1.1468201 , ..., 0.1872781 , -1.9547023 , -1.3492054
```

], [-1.2246968 , -0.77289253, -1.6376779 , ..., 0.5537161 , 0.4848342 , 0.5912451], ..., [1.192786 , -0.13226394, 0.4575644 , ..., -1.5059961 , 2.73987 , -0.7606196], [-0.8590228 , -0.1044452 , -0.9715867 , ..., -0.9215654 , 1.5691227 , -0.5617567], [0.606453 , 0.49316478, -0.6383934 , ..., -1.3212531 , -0.62109023, 0.1684509]], [[-0.02599278, -0.67362314, 0.48570326, ..., 0.22644693, 0.9934903 , -2.5620646], [0.1159848 , -0.47915196, -0.6811871 , ..., 0.09434994, 0.3029398 , 0.42174685], [0.10794386, 0.46578902, -2.4382775 , ..., 0.770606 , -0.76425624, -1.3866335], ..., [0.5851059 , -0.06696272, 1.4475126 , ..., 0.34095022, -0.13524076, -1.0530555], [-1.4368769 , -0.82656825, -0.6933661], ..., [0.5489716 , 0.54459846, -0.8586318 , ..., -0.5403502 , -0.61418784, 0.23939401], [-0.72123253, 0.7499539 , -0.63613904, ..., 0.78031933, -0.98876697, -0.56911653], [1.1187263 , 0.5863637 , -0.04054144, ..., 0.31619748, 0.47991168, 0.5674796]]], dtype=float32)>0.73706305, -1.3118304 , ..., 0.32239807, 1.0951602 , -1.5393324], [-0.07207929, 0.00580749, 0.22784917, ..., 0.81736535, -0.89541644, 0.7397061]], [[0.6770822 , -0.74941754, 0.8807446 , ..., -0.5339431 , 1.5411109 , 0.15334469], [1.0478606 , 1.9049921 , 0.48189902, ..., 0.35219425, 1.9818558 , 0.2982525], [-2.1503694 , 0.05650509, -1.7292447 , ..., 0.37240797,]]

In addition to the class dimension, we can also merge tensors in other dimensions. Consider that tensor A saves the first four subjects' scores of all students in all classes, with shape [10,35,4] and tensor B saves the remaining 4 subjects' scores, with shape [10,35,4]. We can get the total gradebook tensor by merging A and B as in the following:

In [2]:
```
a = tf.random.normal([10,35,4])
b = tf.random.normal([10,35,4])
tf.concat([a,b],axis=2) # Merge along the last dimension
```

Out[2]:
```
<tf.Tensor: id=28, shape=(10, 35, 8), dtype=float32, numpy=
array([[[-5.13509691e-01, -1.79707789e+00, 6.50747120e-01, ...,
 2.58447856e-01, 8.47878829e-02, 4.13468748e-01],
 [-1.17108583e+00, 1.93961406e+00, 1.27830813e-02, ...,]
```

Syntactically, the concatenate operation can be performed on any dimension. The only constraint is that the length of the non-merging dimension must be the same. For example, the tensors with shape [4,32,8] and shape [6,35,8] cannot be directly merged in the class dimension, because the length of the number of students' dimension is not the same – one is 32 and the other is 35, for example:
In [3]:

```
a = tf.random.normal([4,32,8])
 b = tf.random.normal([6,35,8])
 tf.concat([a,b],axis=0) # Illegal merge.
 Second dimension is different.
 Out[3]: InvalidArgumentError:
ConcatOp : Dimensions of inputs should match: shape[0] = [4,32,8] vs.
shape[1] = [6,35,8]
 [Op:ConcatV2] name: concat
```

Suppose We get the gradebook tensor of the entire school with the shape of [10,35,8]. Now we need to cut the data into ten tensors in he class dimension, and each tensor holds the gradebook data of the corresponding class using tensor split operation.Explain the procedure.

9) Splitting:The inverse process of the merge operation is split, which splits a tensor into multiple tensors.

Let's continue the gradebook example. We get the gradebook tensor of the entire school with shape of [10,35,8]. Now we need to cut the data into ten tensors in the class dimension, and each tensor holds the gradebook data of the corresponding class.

tf.split(x, num_or_ size_splits, axis) can be used to complete the tensor split operation. The meaning of the parameters in the function is as follows:

- x: The tensor to be split.
- num_or_size_splits: Cutting scheme. When num_or_ size_splits is a single value, such as 10, it means that the tensor x is cut into ten parts with equal length. When num_or_size_splits is a list, each element of the list represents the length of each part. For example, num_or_size_splits=[2, 4,2,2] means that the tensor is cut into four parts, with the length of each part as 2, 4, 2, and 2.
- axis: Specifies the dimension index of the split. Now we cut the total gradebook tensor into ten pieces as follows:

```
In [8]:
x = tf.random.normal([10,35,8])
# Cut into 10 pieces with equal length
result = tf.split(x, num_or_size_splits=10, axis=0)
len(result) # Return a list with 10 tensors of equal length

Out[8]: 10
```

We can view the shape of a tensor after cutting, and it should be all gradebook data of one class with shape of [1, 35, 8]:

```
In [9]:
result[0] # Check the first class gradebook

Out[9]:
<tf.Tensor: id=136, shape=(1, 35, 8),
dtype=float32, numpy=
array([[[-1.7786729 , 0.2970506 , 0.02983334, 1.3970423 ,
 1.315918 , -0.79110134, -0.8501629 , -1.5549672 ],
 [ 0.5398711 , 0.21478991, -0.08685189, 0.7730989 ,...
```

It can be seen that the shape of the first class tensor is [1,35,8], which still has the class dimension.

Let's perform unequal length cutting.
For example, split the data into four parts with each length as [4, 2, 2, 2] for each part:

```
In [10]: x = tf.random.normal([10,35,8])
# Split tensor into 4 parts
result = tf.split(x, num_or_size_splits=[4,2,2,2] ,axis=0)
len(result)

Out[10]: 4
```

Check the shape of the first split tensor. According to our splitting scheme, it should contain the gradebooks of four classes. The shape should be [4,35,8]:

```
In [10]: result[0]
Out[10]: <tf.Tensor: id=155, shape=(4, 35, 8),
dtype=float32, numpy=
array([[[-6.95693314e-01, 3.01393479e-01, 1.33964568e-01, ...,
```

In particular, if we want to divide one certain dimension by a length of 1, we can use the **tf.unstack(x, axis)** function. This method is a special case of tf.split. The splitting length is fixed as 1. We only need to specify the index number of the splitting dimension.
For example, unstack the total gradebook tensor in the class dimension:

In [11]: x = tf.random.normal([10,35,8])
result = tf.unstack(x,axis=0)
len(result) # Return a list with 10 tensors

Out[11]: 10

View the shape of the split tensor:
In [12]: result[0] # The first class tensor

Out[12]: <tf.Tensor: id=166, shape=(35, 8),
dtype=float32, numpy=
array([[-0.2034383 , 1.1851563 , 0.25327438,
-0.10160723, 2.094969 ,
 -0.8571669 , -0.48985648, 0.55798006],...

It can be seen that after splitting through tf.unstack, the split tensor shape becomes [35, 8], that is, the class dimension disappears, which is different from tf.split

It can be seen that after splitting through tf.unstack, the split tensor shape becomes [35, 8], that is, the class dimension disappears, which is different from tf.split.

5.2 Common Statistics

During the neural network calculations, various statistical attributes need to be computed, such as maximum, minimum, mean, and norm. Because tensors usually contain a lot of data, it is easier to infer the distribution of tensor values by obtaining the statistical information of these tensors.

5.2.1 Norm

Norm is a measure of the "length" of a vector. It can be generalized to tensors. In neural networks, it is often used to represent the tensor weight and the gradient magnitude. Commonly used norms are:

- L1 norm, defined as the sum of the absolute values of all the elements of the vector:

$$||x||_1 = \sum_i |x_i|$$

- L2 norm, defined as the root sum of the squares of all the elements of the vector:

$$||x||_2 = \sqrt{\sum_i |x_i|^2}$$

- ∞ norm, defined as the maximum of the absolute values of all elements of a vector:

$$||x||_\infty = max_i\left(|x_i|\right)$$

For matrices and tensors, the preceding formulas can also be used after flattening the matrices and tensors into a vector. In TensorFlow, the tf.norm(x, ord) function can be used to solve the L1, L2, and ∞norms, where the parameter ord is specified as 1, 1, and np.inf for L1, L2, and ∞ norms, respectively:

```
In [13]: x = tf.ones([2,2])
tf.norm(x,ord=1) # L1 norm
Out[13]: <tf.Tensor: id=183, shape=(), dtype=float32,
numpy=4.0>
In [14]: tf.norm(x,ord=2) # L2 norm
Out[14]: <tf.Tensor: id=189, shape=(), dtype=float32,
numpy=2.0>
In [15]: import numpy as np
tf.norm(x,ord=np.inf) # ∞ norm
Out[15]: <tf.Tensor: id=194, shape=(), dtype=float32,
numpy=1.0>
```

5.2.2 Max, Min, Mean, and Sum

The tf.reduce_max, tf.reduce_min, tf.reduce_mean, and tf.reduce_sum functions can be used to get the maximum, minimum, mean, and sum of tensors in a certain dimension or in all dimensions.

Consider a tensor of shape [4, 10], where the first dimension represents the number of samples and the second dimension represents the probability that the current sample belongs to each of the ten categories. The maximum value of each sample's probability can be obtained through the tf.reduce_max function:

```
In [16]: x = tf.random.normal([4,10])
tf.reduce_max(x,axis=1) # get maximum value at 2nd dimension
```

```
Out[16]:<tf.Tensor: id=203, shape=(4,), dtype=float32,
numpy=array([1.2410722 , 0.88495886, 1.4170984 , 0.9550192 ],
dtype=float32)>
```

The preceding code returns a vector of length 4, which represents the maximum probability value of each sample. Similarly, we can find the minimum value of the probability for each sample as follows:

```
In [17]: tf.reduce_min(x,axis=1) # get the minimum value at 2nd
dimension
Out[17]:<tf.Tensor: id=206, shape=(4,), dtype=float32,
numpy=array([-0.27862206, -2.4480672 , -1.9983795 , -1.5287997 ],
dtype=float32)>
```

Find the mean probabilities of each sample:

```
In [18]: tf.reduce_mean(x,axis=1)
Out[18]:<tf.Tensor: id=209, shape=(4,), dtype=float32,
numpy=array([ 0.39526337, -0.17684573, -0.148988  ,
-0.43544054], dtype=float32)>
```

When the axis parameter is not specified, the tf.reduce_* functions will find the maximum, minimum, mean, and sum of all the data:

```
In [19]:x = tf.random.normal([4,10])
tf.reduce_max(x),tf.reduce_min(x),tf.reduce_mean(x)
Out [19]: (<tf.Tensor: id=218, shape=(), dtype=float32,
numpy=1.8653786>,
 <tf.Tensor: id=220, shape=(), dtype=float32,
numpy=-1.9751656>,
 <tf.Tensor: id=222, shape=(), dtype=float32,
numpy=0.014772797>)
```

When solving the error function, the error of each sample can be obtained through the MSE function, and the average error of the sample needs to be calculated. Here we can use tf.reduce_mean function as follows:

```
In [20]:
out = tf.random.normal([4,10]) # Simulate output
y = tf.constant([1,2,2,0]) # Real labels
y = tf.one_hot(y,depth=10) # One-hot encoding
loss = keras.losses.mse(y,out) # Calculate loss of each sample
loss = tf.reduce_mean(loss) # Calculate mean loss
loss
Out[20]:
<tf.Tensor: id=241, shape=(), dtype=float32, numpy=1.1921183>
```

Similar to the tf.reduce_mean function, the sum function tf.reduce_sum(x, axis) can calculate the sum of all features of the tensor on the corresponding axis:

```
In [21]:out = tf.random.normal([4,10])
tf.reduce_sum(out,axis=-1) # Calculate sum along the last
dimension
Out[21]:<tf.Tensor: id=303, shape=(4,), dtype=float32,
numpy=array([-0.588144 ,  2.2382064,  2.1582587,  4.962141 ],
dtype=float32)>
```

In addition, to obtain the maximum or minimum value of the tensor, we sometimes also want to obtain the corresponding position index. For example, for the classification tasks, we need to know the position index of the maximum probability, which is usually used as the prediction category. Considering the classification problem with ten categories, we get the output tensor with shape [2, 10], where 2 represents two samples and 10 indicates the probability of belonging to ten categories. Since the position index of the element represents the probability that the current

sample belongs to this category, we often use the index corresponding to the largest probability as the predicted category.

```
In [22]:out = tf.random.normal([2,10])
out = tf.nn.softmax(out, axis=1) # Use softmax to convert to
probability
out
Out[22]:<tf.Tensor: id=257, shape=(2, 10),
dtype=float32, numpy=
array([[0.18773547, 0.1510464 , 0.09431915, 0.13652141, 0.06579739,
        0.02033597, 0.06067333, 0.0666793 , 0.14594753, 0.07094406],
       [0.5092072 , 0.03887136, 0.0390687 , 0.01911005, 0.03850609,
        0.03442522, 0.08060656, 0.10171875, 0.08244187, 0.05604421]],
       dtype=float32)>
```

Taking the first sample as an example, it can be seen that the index with the highest probability (0.1877) is 0. Because the probability on each index represents the probability that the sample belongs to this category, the probability that the first sample belongs to class 0 is the largest. Therefore, the first sample should most likely belong to class 0. This is a typical application where the index number of the maximum needs to be solved.

We can use tf.argmax(x, axis) and tf.argmin(x, axis) to find the index of the maximum and minimum values of x on the axis parameter. For example:

```
In [23]:pred = tf.argmax(out, axis=1)
pred
Out[23]:<tf.Tensor: id=262, shape=(2,), dtype=int64,
numpy=array([0, 0], dtype=int64)>
```

It can be seen that the maximum probability of the two samples appears on index 0, so it is most likely that they both belong to category 0. We can use category 0 as the predicted category for the two samples.

5.3 Tensor Comparison

In order to get the classification metrics such as accuracy, it is generally necessary to compare the prediction result with the real label. Considering the prediction results of 100 samples, the predicted category can be obtained through tf.argmax.

```
In [24]:out = tf.random.normal([100,10])
out = tf.nn.softmax(out, axis=1) # Convert to probability
pred = tf.argmax(out, axis=1) # Find corresponding category
Out[24]:<tf.Tensor: id=272, shape=(100,), dtype=int64, numpy=
array([0, 6, 4, 3, 6, 8, 6, 3, 7, 9, 5, 7, 3, 7, 1, 5, 6, 1, 2,
       9, 0, 6,
       5, 4, 9, 5, 6, 4, 6, 0, 8, 4, 7, 3, 4, 7, 4, 1, 2, 4,
       9, 4,...
```

The pred variable holds the predicted category of the 100 samples. We compare them with the true labels to get a boolean tensor representing whether each sample predicts the correct one. The tf.equal(a, b) (or tf.math.equal(a, b), which is equivalent) function can compare whether the two tensors are equal, for example:

```
In [25]: # Simiulate the true labels
y = tf.random.uniform([100],dtype=tf.int64,maxval=10)
Out[25]:<tf.Tensor: id=281, shape=(100,), dtype=int64, numpy=
array([0, 9, 8, 4, 9, 7, 2, 7, 6, 7, 3, 4, 2, 6, 5, 0, 9, 4, 5,
       8, 4, 2,
       5, 5, 5, 3, 8, 5, 2, 0, 3, 6, 0, 7, 1, 1, 7, 0, 6, 1, 2,
       1, 3, ...
In [26]:out = tf.equal(pred,y) # Compare true and prediction
Out[26]:<tf.Tensor: id=288, shape=(100,), dtype=bool, numpy=
array([False, False, False, False, True, False, False, False, False,
       False, False, False, False, False, True, False,
       False, True,...
```

The tf.equal function returns the comparison result as a boolean tensor. We only need to count the number of True elements to get the correct number of predictions. In order to achieve this, we first convert the boolean type to an integer tensor, that is, True corresponds to 1, and False corresponds to 0, and then sum the number of 1 to get the number of True elements in the comparison result:

```
In [27]:out = tf.cast(out, dtype=tf.float32) # convert to int type
correct = tf.reduce_sum(out) # get the number of True elements
Out[27]:<tf.Tensor: id=293, shape=(), dtype=float32, numpy=12.0>
```

It can be seen that the number of correct predictions in our randomly generated prediction data is 12, so its accuracy is:

$$accuracy = \frac{12}{100} = 12\%$$

This is the normal level of random prediction models.

Except for the tf.equal function, other commonly used comparison functions are shown in Table 5-1.

Table 5-1. *Common comparison functions*

Function	Comparison logic
tf.math.greater	$a > b$
tf.math.less	$a < b$
tf.math.greater_equal	$a \geq b$
tf.math.less_equal	$a \leq b$
tf.math.not_equal	$a \neq b$
tf.math.is_nan	$a = nan$

5.4 Fill and Copy

5.4.1 Fill

The height and width of images and the length of the sequence signals may not be the same. In order to facilitate parallel computing of the network, it is necessary to expand data of different lengths to the same. We previously introduced that the length of data can be increased by copying. However, repeatedly copying data will destroy the original data structure and is not suitable for some situations. A common practice is to fill in a sufficient number of specific values at the beginning or end of the data. These specific values (e.g., 0) generally represent invalid meanings. This operation is called padding.

Consider a two-sentence tensor that each word is represented by a digital code, such as 1 for I, 2 for like, and so on. The first sentence is "I like the weather today." We assume that the sentence number is encoded as $[1, 2, 3, 4, 5, 6]$. The second sentence is "So do I." with encoding as $[7, 8, 1, 6]$. In order to store the two sentences in one tensor, we need to keep the length of these two sentences consistent, that is, we need to expand the length of the second sentence to 6. A common padding scheme is to pad a number of zeros at the end of the second sentence, that is, $[7, 8, 1, 6, 0, 0]$. Now the two sentences can be stacked and combined into a tensor of shape $[2, 6]$.

The padding operation can be implemented by the tf.pad(x, paddings) function. The parameter paddings is a list of multiple nested schemes with the format of [*Left Padding*, *Right Padding*]. For example, *paddings* = $[[0, 0], [2, 1], [1, 2]]$ indicates that the first dimension is not filled, and the left (the beginning) of the second dimension is filled with two units, and fill one unit on the right (end) of the second dimension, fill one unit on the left of the third dimension, and fill two units on the right of the third dimension. Considering the example of the preceding two sentences,

two units need to be filled to the right of the first dimension of the second sentence, and the paddings scheme is [[0, 2]]:

```
In [28]:a = tf.constant([1,2,3,4,5,6]) # 1st sentence
b = tf.constant([7,8,1,6]) # 2nd sentence
b = tf.pad(b, [[0,2]]) # Pad two 0's in the end of 2nd sentence
b
Out[28]:<tf.Tensor: id=3, shape=(6,), dtype=int32,
numpy=array([7, 8, 1, 6, 0, 0])>
```

After filling, the shape of the two tensors is consistent, and we can stack them together. The code is as follows:

```
In [29]:tf.stack([a,b],axis=0) # Stack a and b
Out[29]:<tf.Tensor: id=5, shape=(2, 6), dtype=int32, numpy=
array([[1, 2, 3, 4, 5, 6],
       [7, 8, 1, 6, 0, 0]])>
```

In natural language processing, sentences with different lengths need to be loaded. Some sentences are shorter, such as only ten words, and some sentences are longer, such as more than 100 words. In order to be able to save in the same tensor, a threshold that can cover most of the sentence length is generally selected, such as 80 words. For sentences with less than 80 words, we fill with 0s at the end of those sentences. For sentences with more than 80 words, we truncate the sentence to 80 words by removing some words at the end. We will use the IMDB dataset as an example to demonstrate how to transform sentences of unequal length into a structure of equal length. The code is as follows:

```
In [30]:total_words = 10000 # Set word number
max_review_len = 80 # Maximum length for each sentence
embedding_len = 100 # Word vector length
# Load IMDB dataset
```

```
(x_train, y_train), (x_test, y_test) = keras.datasets.imdb.
load_data(num_words=total_words)
# Pad or truncate sentences to the same length with end padding
and truncation
x_train = keras.preprocessing.sequence.pad_sequences(x_train,
maxlen=max_review_len,truncating='post',padding='post')
x_test = keras.preprocessing.sequence.pad_sequences(x_test,
maxlen=max_review_len,truncating='post',padding='post')
print(x_train.shape, x_test.shape)
Out[30]: (25000, 80) (25000, 80)
```

In the preceding code, we set the maximum length of the sentence max_review_len to 80 words. Through the keras.preprocessing.sequence.pad_sequences function, we can quickly complete the padding and truncation implementation. Take one of the sentences as an example, and the transformed vector is like this:

```
[   1  778  128   74   12  630  163   15    4 1766 7982 1051    2   32
   85  156   45   40  148  139  121  664  665   10   10 1361  173    4
  749    2   16 3804    8    4  226   65   12   43  127   24    2   10
   10    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0]
```

We can see that the final part of the sentence is filled with 0s so that the length of the sentence is exactly 80. In fact, we can also choose to fill the beginning part of the sentence when the length of the sentence is not enough. After processing, all sentence length becomes 80, so that the training set can be uniformly stored in the tensor of shape $[25000, 80]$ and the test set can be stored in the tensor of shape $[25000, 80]$.

Let's introduce an example of filling in multiple dimensions at the same time. Consider padding the height and width dimensions of images. If we have pictures with dimension 28×28 and the input layer shape of

neural network is 32 × 32, we need to fill the images to get the shape of 32 × 32. We can choose to fill 2 units each in the upper, lower, left, and right of the image matrix as shown in Figure 5-2.

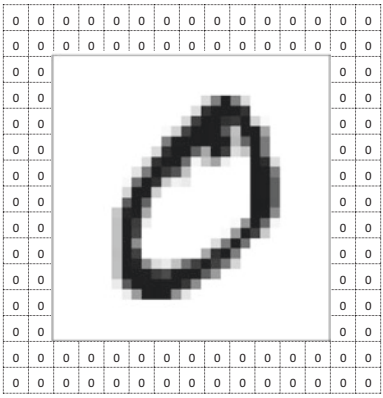


Figure 5-2. *Image padding example*

The preceding padding scheme can be implemented as follows:

```
In [31]:
x = tf.random.normal([4,28,28,1])
# Pad two units at each edge of the image
tf.pad(x,[[0,0],[2,2],[2,2],[0,0]])
Out[31]:
<tf.Tensor: id=16, shape=(4, 32, 32, 1), dtype=float32, numpy=
array([[[[ 0.        ],
         [ 0.        ],
         [ 0.        ],...
```

After the padding operation, the size of the picture becomes 32 × 32, which meets the input requirements of the neural network.

5.4.2 Copy

In the dimensional transformation section, we introduced the tf.tile function of copying the dimension of length 1. Actually, the tf.tile function can be used to repeatedly copy multiple copies of data in any dimension. For example, for image data with shape [4,32,32,3], if the copy scheme is multiples=[2, 3, 3, 1], that means the channel dimension is not copied, three copies in the height and width dimensions, and two copies in the image number dimension. The implementation is as follows:

```
In [32]:x = tf.random.normal([4,32,32,3])
tf.tile(x,[2,3,3,1])
Out[32]:<tf.Tensor: id=25, shape=(8, 96, 96, 3),
dtype=float32, numpy=
array([[[[ 1.20957184e+00,  2.82766962e+00,  1.65782201e+00],
         [ 3.85402292e-01,  2.00732923e+00, -2.79068202e-01],
         [-2.52583921e-01,  7.82584965e-01,  7.56870627e-01],...
```

5.5 Data Limiting

Consider how to implement the nonlinear activation function ReLU. In fact, it can be implemented by simple data limiting operations with the range of elements being limited to $x \in [0, +\infty)$.

In TensorFlow, the lower limit of the data can be set through tf.maximum (x, a), that is, the upper limit of the data can be set through tf.minimum (x, a).

```
In [33]:x = tf.range(9)
tf.maximum(x,2) # Set lower limit of x to 2
Out[33]:<tf.Tensor: id=48, shape=(9,), dtype=int32,
numpy=array([2, 2, 2, 3, 4, 5, 6, 7, 8])>
In [34]:tf.minimum(x,7) # Set x upper limit to 7
```

```
Out[34]:<tf.Tensor: id=41, shape=(9,), dtype=int32,
numpy=array([0, 1, 2, 3, 4, 5, 6, 7, 7])>
```

Based on tf.maximum function, we can implement ReLU as follows:

```
def relu(x): # ReLU function
    return tf.maximum(x,0.) # Set lower limit of x to be 0
```

By combining tf.maximum(x, a) and tf.minimum(x, b), you can limit the upper and lower boundaries of the data at the same time, that is, $x \in [a, b]$.

```
In [35]:x = tf.range(9)
tf.minimum(tf.maximum(x,2),7) # Set x range to be [2, 7]
Out[35]:<tf.Tensor: id=57, shape=(9,), dtype=int32,
numpy=array([2, 2, 2, 3, 4, 5, 6, 7, 7])>
```

More conveniently, we can use the tf.clip_by_value function to achieve upper and lower clipping:

```
In [36]:x = tf.range(9)
tf.clip_by_value(x,2,7) # Set x range to be [2, 7]
Out[36]:<tf.Tensor: id=66, shape=(9,), dtype=int32,
numpy=array([2, 2, 2, 3, 4, 5, 6, 7, 7])>
```

5.6 Advanced Operations

Most of the preceding functions are common and easy to understand. Next, we will introduce some commonly used but slightly more complicated functions.

5.6.1 tf.gather

The tf.gather function can collect data according to the index number. Consider the example of grade books. Assume that there are four classes, 35 students in each class, eight subjects in total, and the tensor shape of the grade books is [4,35,8].

```
x = tf.random.uniform([4,35,8],maxval=100,dtype=tf.int32)
```

Now we need to collect the grade books of the first and second classes. We can give the index number of the class we want to collect (e.g., [0, 1]) and specify the dimension of the class (e.g., axis = 0). And then collect the data through the tf.gather function.

```
In [38]:tf.gather(x,[0,1],axis=0) # Collect data for 1st and
2nd classes
Out[38]:<tf.Tensor: id=83, shape=(2, 35, 8),
dtype=int32, numpy=
array([[[43, 10, 93, 85, 75, 87, 28, 19],
        [52, 17, 44, 88, 82, 54, 16, 65],
        [98, 26,  1, 47, 59,  3, 59, 70],...
```

In fact, the preceding requirements can be more conveniently achieved through slicing. However, for irregular indexing methods, such as the need to spot check the grade data of students 1, 4, 9, 12, 13, and 27, the slicing method is not suitable. The tf.gather function is designed for this situation and is more convenient to use. The implementation is as follows:

```
In [39]: # Collect the grade of students 1,4,9,12,13 and 27
tf.gather(x,[0,3,8,11,12,26],axis=1)
Out[39]:<tf.Tensor: id=87, shape=(4, 6, 8), dtype=int32, numpy=
array([[[43, 10, 93, 85, 75, 87, 28, 19],
        [74, 11, 25, 64, 84, 89, 79, 85],...
```

If we need to collect the grades of the third and fifth subjects of all students, we can specify the subject dimension axis = 2 to achieve the following:

```
# Collect the grades of the 3rd and 5th subjects of all
students
In [40]:tf.gather(x,[2,4],axis=2)
Out[40]:<tf.Tensor: id=91, shape=(4, 35, 2),
dtype=int32, numpy=
array([[[93, 75],
        [44, 82],
        [ 1, 59],...
```

It can be seen that tf.gather is very suitable for situations where the index numbers are not regular. The index numbers can be arranged out of order, and the data collected will also be in the corresponding order. For example:

```
In [41]:a=tf.range(8)
a=tf.reshape(a,[4,2])
Out[41]:<tf.Tensor: id=115, shape=(4, 2), dtype=int32, numpy=
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])>
In [42]:tf.gather(a,[3,1,0,2],axis=0) # Collect element 4,2,1,3
Out[42]:<tf.Tensor: id=119, shape=(4, 2), dtype=int32, numpy=
array([[6, 7],
       [2, 3],
       [0, 1],
       [4, 5]])>
```

We will make the problem a little more complicated. If we want to check the subject scores of students [3, 4, 6, 27] in class [2, 3], we can do this by combining multiple tf.gather operations. First extract data for class [2, 3]:

```
In [43]:
students=tf.gather(x,[1,2],axis=0) # Collect data for
class 2 and 3
Out[43]:<tf.Tensor: id=227, shape=(2, 35, 8),
dtype=int32, numpy=
array([[[ 0, 62, 99,  7, 66, 56, 95, 98],...
```

Then we extract the corresponding data for selected students:

```
In [44]:
tf.gather(students,[2,3,5,26],axis=1) # Collect data for
students 3,4,6,27
Out[44]:<tf.Tensor: id=231, shape=(2, 4, 8),
dtype=int32, numpy=
array([[[69, 67, 93,  2, 31,  5, 66, 65], ...
```

Now we get the selected tensor with shape [2, 4, 8].

This time we want to spot check all subjects of the second classmate of the second class, all subjects of the third classmate of the third class, and all subjects of the fourth classmate of the fourth class. So how does it work? Data can be manually extracted one by one in a clumsy way. First extract the data of the first sampling point: $x[1, 1]$.

```
In [45]: x[1,1]
Out[45]:<tf.Tensor: id=236, shape=(8,), dtype=int32,
numpy=array([45, 34, 99, 17,  3,  1, 43, 86])>
```

Then extract the data of the second sampling point $x[2, 2]$ and the data of the third sampling point $x[3, 3]$, and finally combine the sampling results together.

```
In [46]: tf.stack([x[1,1],x[2,2],x[3,3]],axis=0)
Out[46]:<tf.Tensor: id=250, shape=(3, 8), dtype=int32, numpy=
array([[45, 34, 99, 17,  3,  1, 43, 86],
       [11, 25, 84, 95, 97, 95, 69, 69],
       [ 0, 89, 52, 29, 76,  7,  2, 98]])>
```

Using the preceding method, we can correctly obtain the result of shape $[3, 8]$, where 3 represents the number of sampling points and 4 represents the data of each sampling point. The biggest problem is that the sampling is performed manually and serially, and the calculation efficiency is extremely low. Is there a better way to achieve this?

5.6.2 tf.gather_nd

With the tf.gather_nd function, we can sample multiple points by specifying the multidimensional coordinates of each sampling point. Going back to the preceding challenge, we want to spot check all the subjects of the second classmate of the second class, all the subjects of the third classmate of the third class, and all the subjects of the fourth classmate of the fourth class. Then the index coordinates of the three sampling points can be recorded as $[1, 1]$, $[2, 2]$, *and* $[3, 3]$, and we can combine this sampling scheme into a list $[[1, 1], [2, 2], [3, 3]]$.

```
In [47]:
tf.gather_nd(x,[[1,1],[2,2],[3,3]])
Out[47]:<tf.Tensor: id=256, shape=(3, 8), dtype=int32, numpy=
array([[45, 34, 99, 17,  3,  1, 43, 86],
       [11, 25, 84, 95, 97, 95, 69, 69],
       [ 0, 89, 52, 29, 76,  7,  2, 98]])>
```

The result is consistent with the serial sampling method, and the implementation is more concise and efficient.

Generally, when using tf.gather_nd to sample multiple samples, for example, if we want to sample class *i*, student *j*, and subject *k*, we can use the expression [..., [*i*,*j*,*k*], ...]. The inner list contains the corresponding index coordinates of each sampling point, for example:

```
In [48]:
tf.gather_nd(x,[[1,1,2],[2,2,3],[3,3,4]])
Out[48]:<tf.Tensor: id=259, shape=(3,), dtype=int32,
numpy=array([99, 95, 76])>
```

In the preceding code, we extracted the grades of subject 1 of class 1 student 2, subject 2 of class 2 student 3, and class 3 of student 3 subject 4. There are a total of three grade data, and the results are summarized into a tensor with shape of [3].

5.6.3 tf.boolean_mask

In addition to sampling by a given index number, sampling can also be performed by a given mask. Continue to take the gradebook tensor with shape [4,35,8] as an example; this time we use the mask method for data extraction.

Consider sampling in the class dimension and set the corresponding mask as:

$$mask = [True, False, False, True]$$

That is, the first and fourth classes are sampled. Using the function tf.boolean_mask(x, mask, axis), the sampling can be performed on the corresponding axis according to the mask scheme, which is realized as:

```
In [49]:
tf.boolean_mask(x,mask=[True, False,False,True],axis=0)
```

```
Out[49]:<tf.Tensor: id=288, shape=(2, 35, 8),
dtype=int32, numpy=
array([[[43, 10, 93, 85, 75, 87, 28, 19],...
```

Note that the length of the mask must be the same as the length of the corresponding dimension. If we are sampling in the class dimension, we must specify the mask with length 4 to specify whether the four classes are sampling.

If mask sampling is performed on eight subjects, we need to set the mask sampling scheme to

$$mask = [True, False, False, True, True, False, False, True]$$

That is, sample the first, fourth, fifth, and eighth subjects:

```
In [50]:
tf.boolean_mask(x,mask=[True,False,False,True,True,False,False,
True],axis=2)
Out[50]:<tf.Tensor: id=318, shape=(4, 35, 4),
dtype=int32, numpy=
array([[[43, 85, 75, 19],...
```

It is not difficult to find that the usage of tf.boolean_mask here is actually very similar to tf.gather, except that one is sampled by the mask method, and the other is directly given the index number.

Now let's consider a multidimensional mask sampling method similar to tf.gather_nd. In order to facilitate the demonstration, we reduced the number of classes to two and the number of students to three. That is, a class has only three students and the tensor shape is $[2, 3, 8]$. If we want to

sample students 1 to 2 of the first class and students 2 to 3 of the second class, we can achieve it using tf.gather_nd:

```
In [51]:x = tf.random.uniform([2,3,8],maxval=100,dtype
=tf.int32)
tf.gather_nd(x,[[0,0],[0,1],[1,1],[1,2]])
Out[51]:<tf.Tensor: id=325, shape=(4, 8), dtype=int32, numpy=
array([[52, 81, 78, 21, 50,  6, 68, 19],
       [53, 70, 62, 12,  7, 68, 36, 84],
       [62, 30, 52, 60, 10, 93, 33,  6],
       [97, 92, 59, 87, 86, 49, 47, 11]])>
```

A total of four students' results were sampled with a shape of [4, 8].

If we use a mask, how do we express it? Table 5-2 expresses the sampling of the corresponding position:

Table 5-2. *Sampling using mask method*

	Student 0	Student 1	Student 2
Class 0	True	True	False
Class 1	False	True	True

Therefore, through this table, the sampling scheme using the mask method can be well expressed. The code is implemented as follows:

```
In [52]:
tf.boolean_mask(x,[[True,True,False],[False,True,True]])
Out[52]:<tf.Tensor: id=354, shape=(4, 8), dtype=int32, numpy=
array([[52, 81, 78, 21, 50,  6, 68, 19],
       [53, 70, 62, 12,  7, 68, 36, 84],
       [62, 30, 52, 60, 10, 93, 33,  6],
       [97, 92, 59, 87, 86, 49, 47, 11]])>
```

The result is exactly the same as tf.gather_nd method. It can be seen that tf.boolean_mask method can be used for both one- and multidimensional samplings.

The preceding three operations are more commonly used, especially tf.gather and tf.gather_nd. Three additional advanced operations are added in the following.

5.6.4 tf.where

Through the tf.where(cond, a, b) function, we can read data from the parameter a or b according to the true and false conditions of the cond condition. The condition determination rule is as follows:

$$o_i = \{a_i \; cond_i 为 True \quad b_i \; cond_i 为 False$$

Among them i is the element index of the tensor. The size of the returned tensor is consistent with a and b. When the corresponding position of $cond_i$ is True, the data is copied from a_i to o_i. Otherwise, the data is copied from b_i to o_i. Consider extracting data from two tensors A and B of all 1's and 0's, where the position of True in中$cond_i$ extracts element 1 from the corresponding position of A, otherwise extracts 0 from the corresponding position of B. The code is as follows:

```
In [53]:
a = tf.ones([3,3])  # Tensor A
b = tf.zeros([3,3]) # Tensor B
# Create condition matrix
cond = tf.constant([[True,False,False],[False,True,False],[True,
True,False]])
tf.where(cond,a,b)
Out[53]:<tf.Tensor: id=384, shape=(3, 3), dtype=float32, numpy=
array([[1., 0., 0.],
```

```
       [0., 1., 0.],
       [1., 1., 0.]], dtype=float32)>
```

It can be seen that the positions of 1 in the returned tensor are all from tensor *A*, and the positions of 0 in the returned tensor are from tensor *B*.

When the parameter a=b=None, that is, a and b parameters are not specified; tf.where returns the index coordinates of all True elements in the cond tensor. Consider the following cond tensor:

```
In [54]: cond
Out[54]:<tf.Tensor: id=383, shape=(3, 3), dtype=bool, numpy=
array([[ True, False, False],
       [False,  True, False],
       [ True,  True, False]])>
```

True appears four times in total, and the index at the position of each True element is [0, 0], [1, 1], [2, 0], and [2, 1] respectively. The index coordinates of these elements can be obtained directly through the form of tf.where(cond) as follows:

```
In [55]:tf.where(cond)
Out[55]:<tf.Tensor: id=387, shape=(4, 2), dtype=int64, numpy=
array([[0, 0],
       [1, 1],
       [2, 0],
       [2, 1]], dtype=int64)>
```

So what's the use of this? Consider a scenario where we need to extract all the positive data and indexes in a tensor. First construct tensor a and obtain the position masks of all positive numbers through comparison operations:

```
In [56]:x = tf.random.normal([3,3]) # Create tensor a
Out[56]:<tf.Tensor: id=403, shape=(3, 3), dtype=float32, numpy=
```

```
array([[-2.2946844 ,  0.6708417 , -0.5222212 ],
       [-0.6919401 , -1.9418817 ,  0.3559235 ],
       [-0.8005251 ,  1.0603906 , -0.68819374]],
       dtype=float32)>
```

By comparison operation, we get the mask of all positive numbers:

```
In [57]:mask=x>0 # equivalent to tf.math.greater()
mask
Out[57]:<tf.Tensor: id=405, shape=(3, 3), dtype=bool, numpy=
array([[False,  True, False],
       [False, False,  True],
       [False,  True, False]])>
```

Extract the index coordinates of the True element in the mask tensor via tf.where:

```
In [58]:indices=tf.where(mask) # Extract all element
greater than 0
Out[58]:<tf.Tensor: id=407, shape=(3, 2), dtype=int64, numpy=
array([[0, 1],
       [1, 2],
       [2, 1]], dtype=int64)>
```

After getting the index, we can restore all positive elements through tf.gather_nd:

```
In [59]:tf.gather_nd(x,indices) # Extract all positive elements
Out[59]:<tf.Tensor: id=410, shape=(3,), dtype=float32,
numpy=array([0.6708417, 0.3559235, 1.0603906], dtype=float32)>
```

In fact, after we get the mask, we can also get all the positive elements directly through tf.boolean_mask:

```
In [60]:tf.boolean_mask(x,mask) # Extract all positive elements
Out[60]:<tf.Tensor: id=439, shape=(3,), dtype=float32,
numpy=array([0.6708417, 0.3559235, 1.0603906], dtype=float32)>
```

Through the preceding series of comparisons, we can intuitively feel that this function has great practical applications and also get a deep understanding of their nature to be able to achieve our purpose in a more flexible, simple, and efficient way.

5.6.5 tf.scatter_nd

The tf.scatter_nd(indices, updates, shape) function can efficiently refresh part of the tensor data, but this function can only perform refresh operations on all 0 tensors, so it may be necessary to combine other operations to implement the data refresh function for non-zero tensors.

Figure 5-3 shows the refresh calculation principle of the one-dimensional all-zero tensor. The shape of the whiteboard is represented by the shape parameter, the index number of the data to be refreshed is represented by indices, and updates parameter contains the new data. The tf.scatter_nd(indices, updates, shape) function writes the new data to the all-zero tensor according to the index position given by indices and returns the updated result tensor.

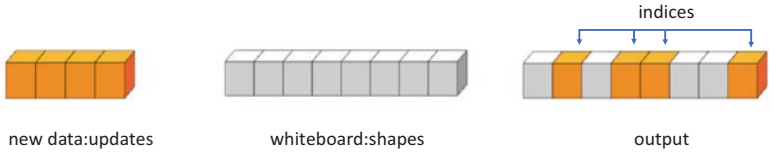


Figure 5-3. *scatter_nd function for refreshing data*

We implement a refresh example of the tensor in Figure 5-3 as follows:

```
In [61]: # Create indices for refreshing data
indices = tf.constant([[4], [3], [1], [7]])
# Create data for filling the indices
updates = tf.constant([4.4, 3.3, 1.1, 7.7])
# Refresh data for all 0 vector of length 8
tf.scatter_nd(indices, updates, [8])
Out[61]:<tf.Tensor: id=467, shape=(8,), dtype=float32,
numpy=array([0. , 1.1, 0. , 3.3, 4.4, 0. , 0. , 7.7],
dtype=float32)>
```

It can be seen that on the all-zero tensor of length 8, the data of the corresponding positions are filled in with values from updates.

Consider an example of a three-dimensional tensor. As shown in Figure 5-4, the shape of the all-zero tensor is a feature map with four channels in total, and each channel has a size 4×4. New data updates have a shape $[2, 4, 4]$, which needs to be written in indices $[1, 3]$.

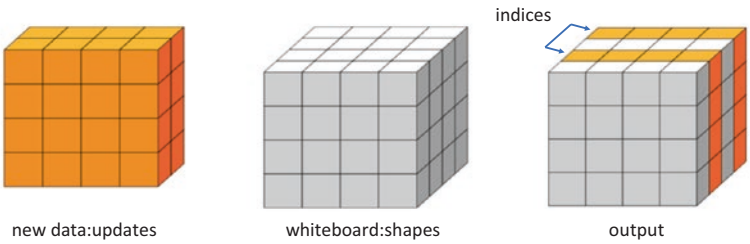


Figure 5-4. *3D tensor data refreshing*

We write the new feature map into the existing tensor as follows:

```
In [62]:
indices = tf.constant([[1],[3]])
updates = tf.constant([
    [[5,5,5,5],[6,6,6,6],[7,7,7,7],[8,8,8,8]],
    [[1,1,1,1],[2,2,2,2],[3,3,3,3],[4,4,4,4]]
```

```
])
tf.scatter_nd(indices,updates,[4,4,4])
Out[62]:<tf.Tensor: id=477, shape=(4, 4, 4),
dtype=int32, numpy=
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],
       [[5, 5, 5, 5], # New data 1
        [6, 6, 6, 6],
        [7, 7, 7, 7],
        [8, 8, 8, 8]],
       [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],
       [[1, 1, 1, 1], # New data 2
        [2, 2, 2, 2],
        [3, 3, 3, 3],
        [4, 4, 4, 4]]])>
```

It can be seen that the data is refreshed onto the second and fourth channel feature maps.

5.6.6 tf.meshgrid

The tf.meshgrid function can easily generate the coordinates of the sampling points of the two-dimensional grid, which is convenient for applications such as visualization. Consider the Sinc function with two independent variables x and y as:

$$z = \frac{sin\ sin\left(x^2 + y^2\right)}{x^2 + y^2}$$

If we need to draw a 3D surface of the Sinc function in the interval $x \in [-8, 8]$, $y \in [-8, 8]$, as shown in Figure 5-5, we first need to generate the grid point coordinate set of the x and y axes, so that the output value of the function at each position can be calculated by the expression of the Sinc function z. We can generate 10,000 coordinate sampling points by:

```
points = []
for x in range(-8,8,100): # Loop to generate 100 sampling point
for x-axis
for y in range(-8,8,100): # Loop to generate 100 sampling point
for y-axis
        z = sinc(x,y)
        points.append([x,y,z])
```

Obviously, this serial sampling method is extremely inefficient. Is there a simple and efficient way to generate grid coordinates? The answer is the tf.meshgrid function.

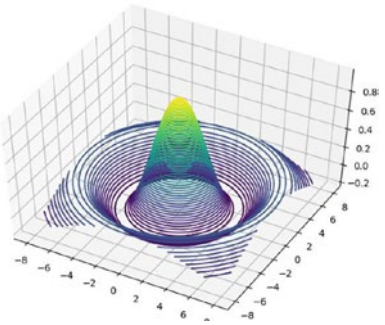


Figure 5-5. *Sinc function*

By sampling 100 data points on the x-axis and y-axis, respectively, the tf.meshgrid(x, y) can be used to generate tensor data of these 10,000 data points and save them in a tensor of shape [100,100,2]. For the convenience of calculation, tf.meshgrid will return two tensors after cutting in the

axis = two-dimensional, where tensor *A* contains the x-coordinates of all points and tensor *B* contains the y-coordinates of all points.

```
In [63]:
x = tf.linspace(-8.,8,100) # x-axis
y = tf.linspace(-8.,8,100) # y-axis
x,y = tf.meshgrid(x,y)
x.shape,y.shape
Out[63]: (TensorShape([100, 100]), TensorShape([100, 100]))
```

Using the generated grid point coordinate tensors, the Sinc function is implemented in TensorFlow as follows:

```
z = tf.sqrt(x**2+y**2)
z = tf.sin(z)/z  # sinc function
```

The matplotlib library can be used to draw the 3D surface of the function as shown in Figure 5-5.

```
import matplotlib
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
# Plot Sinc function
ax.contour3D(x.numpy(), y.numpy(), z.numpy(), 50)
plt.show()
```

5.7 Load Classic Datasets

So far, we have learned the common tensor operations and are ready to implement most of the deep networks. Finally, we will complete this chapter with a classification network model implemented in a tensor

PERCEPTRON MODEL:

The perceptron model is a fundamental building block of more complex neural network architectures and was one of the earliest neural network models developed. While it's limited to solving linearly separable problems, it laid the foundation for more sophisticated neural network architectures capable of solving complex tasks.

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.

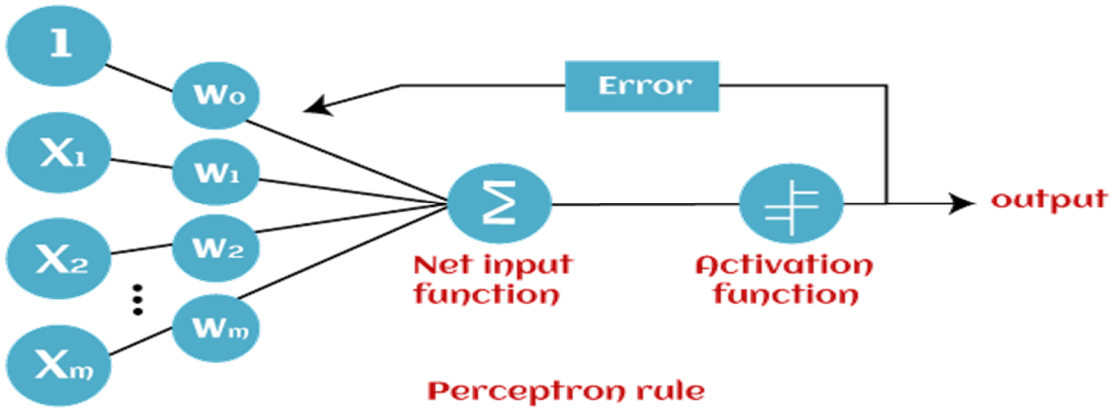
This model uses a hyperplane line that classifies two inputs and classifies them on the basis of the 2 classes that a machine learns, thus implying that the perceptron model is a linear classification model.
There are 4 constituents of a perceptron model. They are as follows-

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

Output: parameters w and b

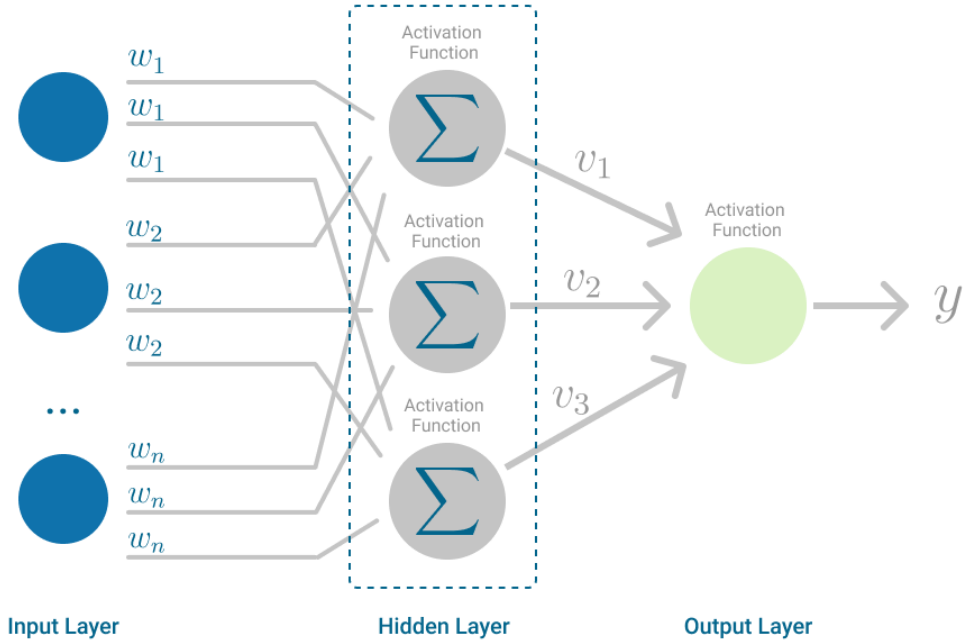
NOTE: Here η is learning rate

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

A fully connected layer, also known as a dense layer, is one of the basic building blocks of a neural network. It is characterized by each neuron in the layer being connected to every neuron in the previous layer, thus forming a fully connected network structure.

A fully connected layer consists of :

- **Input**: Each neuron in the fully connected layer receives input from every neuron in the previous layer. If the previous layer is the input layer, the input would be the features of the input data.
- **Weights**: Every connection between neurons in the previous layer and neurons in the fully connected layer is associated with a weight. These weights determine the strength of the connection and are adjusted during the training process to learn the patterns in the data.
- **Bias**: Each neuron in the fully connected layer also has a bias term associated with it. The bias term provides the neuron with additional flexibility to learn the desired output.
- **Weighted Sum**: For each neuron in the fully connected layer, the inputs from the previous layer are multiplied by their corresponding weights, and these weighted values are summed together. Additionally, the bias term is added to the sum.
- **Activation Function**: The weighted sum computed for each neuron is then passed through an activation function, which introduces non-linearity into the network. Common activation functions used in fully connected layers include sigmoid, tanh, ReLU (Rectified Linear Unit), and softmax, depending on the task at hand.
- **Output**: The output of each neuron in the fully connected layer is the result of applying the activation function to the weighted sum.

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:

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

The output of the second node is:

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

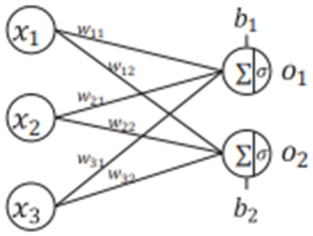
Putting them together, the output vector is $o = [o_1, o_2]$. The entire network layer can be expressed by the matrix relationship:

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

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.



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

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

Where:

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

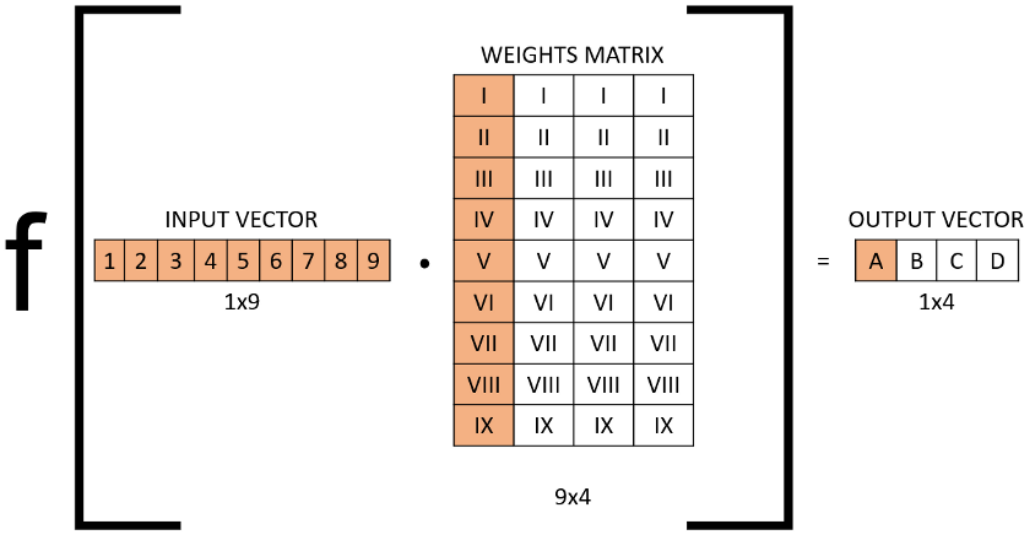
Why are fully connected layers required?

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:

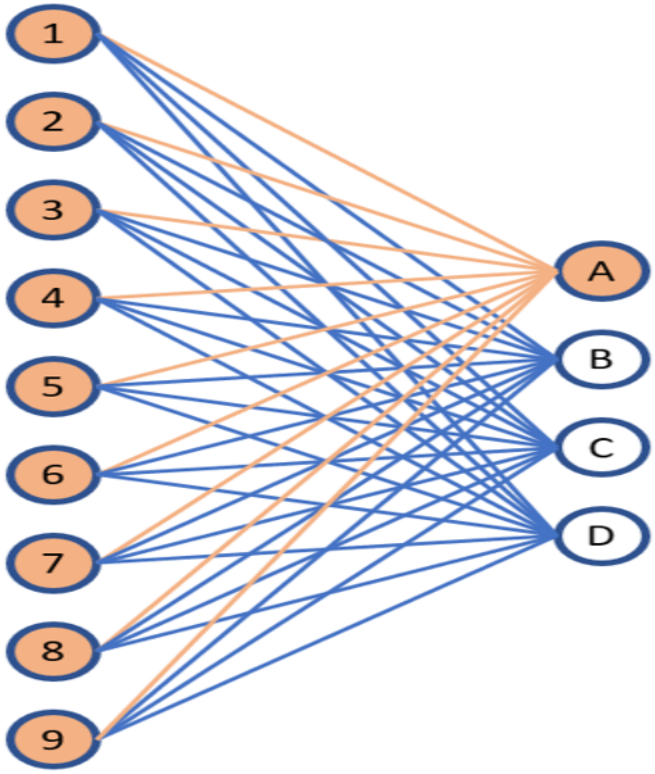


The ***activation function f*** wraps the dot product between the input of the layer and the weights matrix of that layer.

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 $\mathbb{R}$ m to $\mathbb{R}$ 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

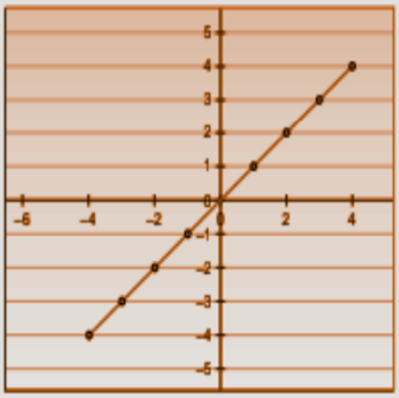
ACTIVATION FUNCTION - TYPES:

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:

$$f(x)=x$$

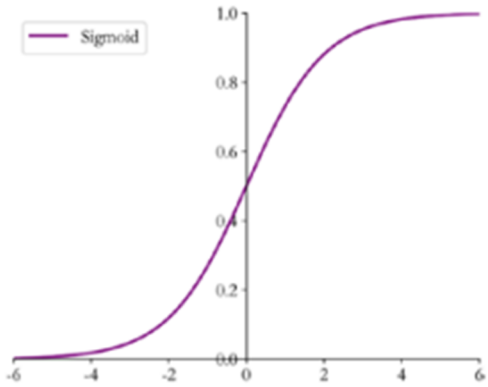
Sigmoid/Logistic Function:

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

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

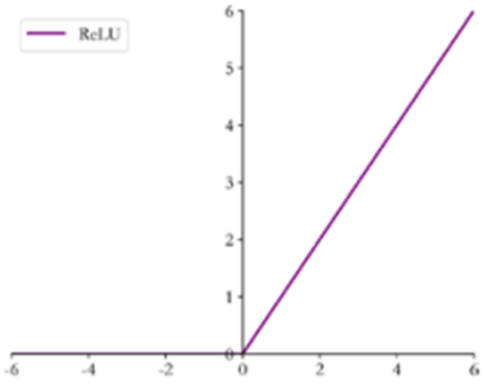
It squishes all the values to a probability between 0 and 1, which reduces extreme values or outliers. It is used to classify data into two classes

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.**

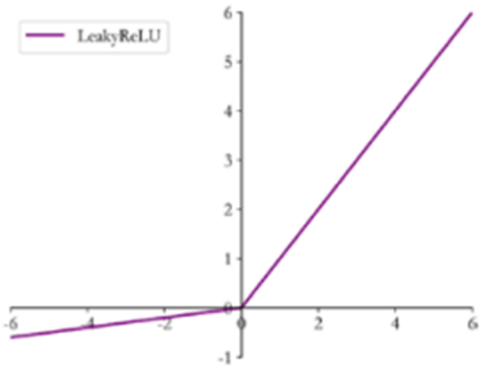
ReLU function:



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

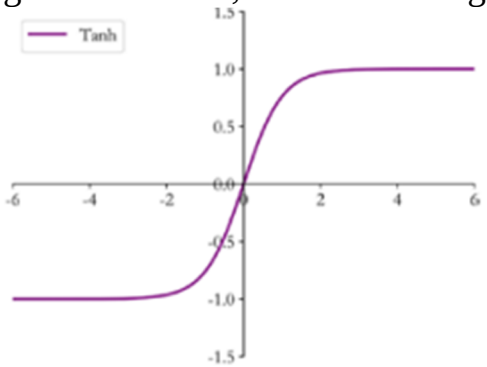
Tanh function:

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

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

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

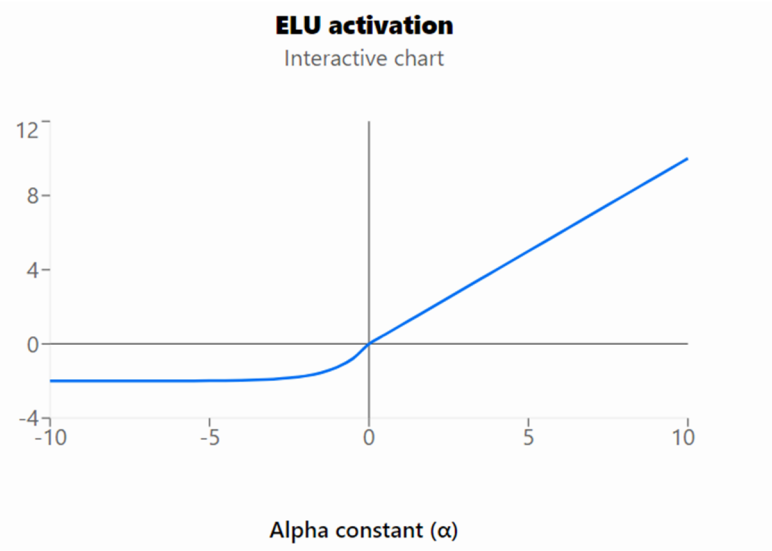
It can be seen that the Tanh activation function can be realized after zooming and translated by the Sigmoid function, as shown in Figure:



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.

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.

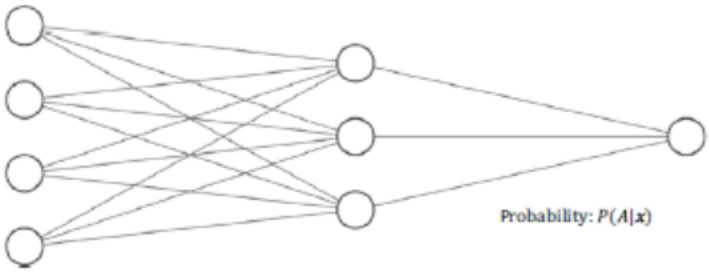
It is necessary to decide whether to use the activation function and what type of activation function to use according to the specific tasks. We will classify the discussions based on the range of output values.

[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:

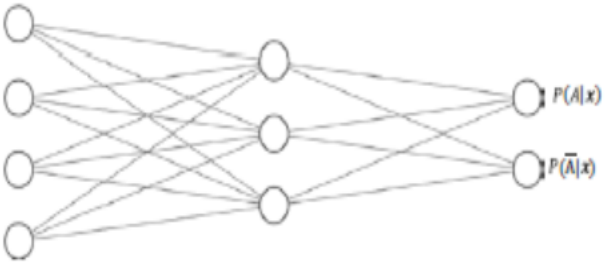
$$P(x) = o$$

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



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.



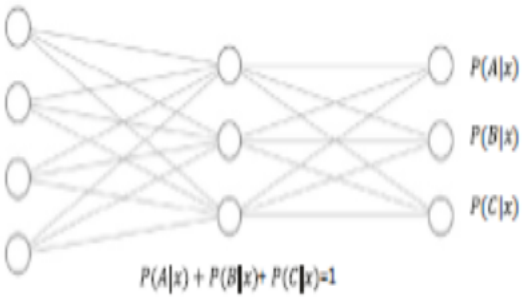
The output value of the first node represents the probability of the occurrence of event A $P(x)$, and the output value of the second node represents the probability of the occurrence of the opposite event $P(x)$. The function can only compress a single value to the interval (0, 1) and does not consider 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.

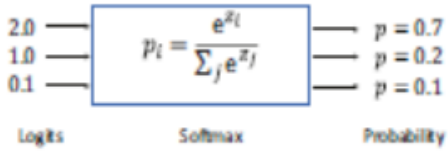
The network structure in the figure is used to handle three classification tasks. The output value distribution of the three nodes represents the probability that the current sample belongs to category A, B, and C: $P(x)$, $P(B|x)$, and $P(C|x)$. Because the sample in the multi-classification problem 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:

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

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:

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

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:

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

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.

HANDS-ON OF AUTOMOBILE FUEL CONSUMPTION DATASET

```
import tensorflow as tf
import keras
import pandas as pd
# Download the dataset online
dataset_path = keras.utils.get_file("auto-
mpg.data","http://archive.ics.uci.edu/ml/machine-learning-databases/
auto-mpg/auto-mpg.data")
# Use Pandas library to read the dataset
column_names =
['MPG','Cylinders','Displacement','Horsepower','Weight',
'Acceleration', 'Model Year', 'Origin']
raw_dataset = pd.read_csv(dataset_path, names=column_names,na_values
="?", comment='\t',sep=" ", skipinitialspace=True)
dataset = raw_dataset.copy()
# Show some data
dataset.head()
```

```
Downloading data from http://archive.ics.uci.edu/ml/machine-learning-
databases/auto-mpg/auto-mpg.data
      0/Unknown 0s 0s/step
```


```
{"summary":"{\n  \"name\": \"dataset\",\n  \"rows\": 398,\n
\"fields\": [\n    {\n      \"column\": \"MPG\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
7.815984312565782,\n        \"min\": 9.0,\n        \"max\": 46.6,\n
\"num_unique_values\": 129,\n        \"samples\": [\n          17.7,\n
30.5,\n          30.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Cylinders\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 1,\n        \"min\": 3,\n
\"max\": 8,\n        \"num_unique_values\": 5,\n        \"samples\":
[\n          4,\n          5,\n          6\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"Displacement\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
104.26983817119581,\n        \"min\": 68.0,\n        \"max\": 455.0,\n
\"num_unique_values\": 82,\n        \"samples\": [\n          122.0,\n
307.0,\n          360.0\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"Horsepower\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 38.49115993282855,\n
\"min\": 46.0,\n        \"max\": 230.0,\n
\"num_unique_values\": 93,\n        \"samples\": [\n          92.0,\n
100.0,\n          52.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Weight\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 846.8417741973271,\n        \"min\": 1613.0,\n
\"max\": 5140.0,\n        \"num_unique_values\": 351,\n
\"samples\": [\n          3730.0,\n          1995.0,\n
2215.0\n        ],\n        \"semantic_type\": \"\",\n
```

\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"Acceleration\",\n   \"properties\": {\n   \"dtype\":
\"number\",\n   \"std\": 2.7576889298126757,\n   \"min\":
8.0,\n   \"max\": 24.8,\n   \"num_unique_values\": 95,\n
\"samples\": [\n   14.7,\n   18.0,\n   14.3\n
],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n
}\n   },\n   {\n   \"column\": \"Model Year\",\n
\"properties\": {\n   \"dtype\": \"number\",\n   \"std\":
3,\n   \"min\": 70,\n   \"max\": 82,\n
\"num_unique_values\": 13,\n   \"samples\": [\n   81,\n
79,\n   70\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"Origin\",\n   \"properties\": {\n   \"dtype\": \"number\",\n
\"std\": 0,\n   \"min\": 1,\n   \"max\": 3,\n
\"num_unique_values\": 3,\n   \"samples\": [\n   1,\n
3,\n   2\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   }\n]\
n}","type":"dataframe","variable_name":"dataset"}

```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   MPG           398 non-null    float64
 1   Cylinders     398 non-null    int64  
 2   Displacement  398 non-null    float64
 3   Horsepower    392 non-null    float64
 4   Weight        398 non-null    float64
 5   Acceleration  398 non-null    float64
 6   Model Year    398 non-null    int64  
 7   Origin        398 non-null    int64  
dtypes: float64(5), int64(3)
memory usage: 25.0 KB
```

```
dataset.isna().sum() # Calculate the number of missing values
```

```
MPG             0
Cylinders       0
Displacement    0
Horsepower      6
Weight          0
Acceleration    0
Model Year      0
Origin          0
dtype: int64
```

```
dataset = dataset.dropna() # Drop missing value records

dataset.isna().sum() # Calculate the number of missing values again
```

```
MPG             0
Cylinders       0
Displacement    0
Horsepower      0
Weight          0
Acceleration    0
Model Year      0
Origin          0
dtype: int64
```

```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 392 entries, 0 to 397
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   MPG           392 non-null    float64
 1   Cylinders     392 non-null    int64  
 2   Displacement  392 non-null    float64
 3   Horsepower    392 non-null    float64
 4   Weight        392 non-null    float64
 5   Acceleration  392 non-null    float64
 6   Model Year    392 non-null    int64  
 7   Origin        392 non-null    int64  
dtypes: float64(5), int64(3)
memory usage: 27.6 KB
```

```
origin = dataset.pop('Origin')
dataset['USA'] = (origin == 1)*1.0
dataset['Europe'] = (origin == 2)*1.0
dataset['Japan'] = (origin == 3)*1.0
dataset.tail()
```


```
{"summary":"{\n  \"name\": \"dataset\",\n  \"rows\": 5,\n  \"fields\":
[\n    {\n      \"column\": \"MPG\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 6.8044103344816005,\n
\"min\": 27.0,\n        \"max\": 44.0,\n        \"num_unique_values\":
5,\n        \"samples\": [\n          44.0,\n          31.0,\n
32.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Cylinders\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0,\n        \"min\": 4,\n
\"max\": 4,\n        \"num_unique_values\": 1,\n        \"samples\":
[\n          4\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Displacement\",\n      \"properties\": {\n        \"dtype\":
```

\"number\",\n   \"std\": 16.813684902483452,\n   \"min\":
97.0,\n   \"max\": 140.0,\n   \"num_unique_values\": 5,\n
\"samples\": [\n   97.0\n   ],\n   \"semantic_type\":
\"\",\n   \"description\": \"\"\n   }\n   },\n   {\n
\"column\": \"Horsepower\",\n   \"properties\": {\n
\"dtype\": \"number\",\n   \"std\": 13.992855319769443,\n
\"min\": 52.0,\n   \"max\": 86.0,\n   \"num_unique_values\":
5,\n   \"samples\": [\n   52.0\n   ],\n
\"semantic_type\": \"\",\n   \"description\": \"\"\n   }\
n   },\n   {\n   \"column\": \"Weight\",\n   \"properties\":
{\n   \"dtype\": \"number\",\n   \"std\":
285.62650437240586,\n   \"min\": 2130.0,\n   \"max\":
2790.0,\n   \"num_unique_values\": 5,\n   \"samples\": [\n
2130.0\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"Acceleration\",\n   \"properties\": {\n   \"dtype\":
\"number\",\n   \"std\": 4.81123684721507,\n   \"min\":
11.6,\n   \"max\": 24.6,\n   \"num_unique_values\": 5,\n
\"samples\": [\n   24.6\n   ],\n   \"semantic_type\":
\"\",\n   \"description\": \"\"\n   }\n   },\n   {\n
\"column\": \"Model Year\",\n   \"properties\": {\n
\"dtype\": \"number\",\n   \"std\": 0,\n   \"min\": 82,\n
\"max\": 82,\n   \"num_unique_values\": 1,\n   \"samples\":
[\n   82\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"USA\",\n   \"properties\": {\n   \"dtype\": \"number\",\n
\"std\": 0.44721359549995804,\n   \"min\": 0.0,\n   \"max\":
1.0,\n   \"num_unique_values\": 2,\n   \"samples\": [\n
0.0\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"Europe\",\n   \"properties\": {\n   \"dtype\": \"number\",\n
\"std\": 0.44721359549995804,\n   \"min\": 0.0,\n   \"max\":
1.0,\n   \"num_unique_values\": 2,\n   \"samples\": [\n
1.0\n   ],\n   \"semantic_type\": \"\",\n
\"description\": \"\"\n   }\n   },\n   {\n   \"column\":
\"Japan\",\n   \"properties\": {\n   \"dtype\": \"number\",\n
\"std\": 0.0,\n   \"min\": 0.0,\n   \"max\": 0.0,\n
\"num_unique_values\": 1,\n   \"samples\": [\n   0.0\n
],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n
}\n   }\n]\n}","type":"dataframe"}

```
#Split the data into training (80%) and testing (20%) datasets:
train_dataset = dataset.sample(frac=0.8,random_state=0)
test_dataset = dataset.drop(train_dataset.index)

train_dataset.columns
```

```
Index(['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',
       'Acceleration', 'Model Year', 'USA', 'Europe', 'Japan'],
      dtype='object')
```

```
train_labels = train_dataset.pop('MPG')
test_labels = test_dataset.pop('MPG')

#Calculate the mean and standard deviation of each field value of the
training set and complete
#the standardization of the data, through the norm() function; the
code is as follows:
train_stats = train_dataset.describe()
train_stats = train_stats.transpose()


# Normalize the data
def norm(x): # minus mean and divide by std
 return (x - train_stats['mean']) / train_stats['std']
normed_train_data = norm(train_dataset)
normed_test_data = norm(test_dataset)

#Print the shape of training and testing datasets:
print(normed_train_data.shape,train_labels.shape)
print(normed_test_data.shape, test_labels.shape)

(314, 9) (314,)
(78, 9) (78,)

#Create TensorFlow dataset:
train_db
=tf.data.Dataset.from_tensor_slices((normed_train_data.values,train_la
bels.values))
train_db = train_db.shuffle(100).batch(32) # Shuffle and batch
```

##Create a Network Considering the small size of the auto MPG dataset, we only create a three-layer fully connected network to complete the MPG prediction task. There are nine input features, so the number of input nodes in the first layer is 9. The number of output nodes of the first layer and the second layer is designed as 64 and 64. Since there is only one kind of prediction value, the output node of the output layer is designed as 1. Because MPG belong to the real number space, the activation function of the output layer may not be added. We implement the network as a custom network class. We only need to create each sub-network layer in the initialization function and implement the calculation logic of the custom network class in the forward calculation function. The custom network class inherits from the keras.Model class, which is also the standard writing method of the custom network class, in order to conveniently use the various convenient functions such as trainable_variables and save_weights provided by the keras.Model class. The network model class is implemented as follows:

```
class Network(keras.Model):
 # regression network
  def __init__(self):
    super(Network, self).__init__()
 # create 3 fully-connected layers
    self.fc1 = layers.Dense(64, activation='relu')
```

```
    self.fc2 = layers.Dense(64, activation='relu')
    self.fc3 = layers.Dense(1)
  def call(self, inputs, training=None, mask=None):
 # pass through the 3 layers sequentially
    x = self.fc1(inputs)
    x = self.fc2(x)
    x = self.fc3(x)
    return x

##Training and Testing
#After the creation of the main network model class, let's instantiate
the network object and
#create the optimizer as follows:

from tensorflow.keras import layers
model = Network() # Instantiate the network
# Build the model with 4 batch and 9 features
model.build(input_shape=(4, 9))
model.summary() # Print the network
# Create the optimizer with learning rate 0.001
optimizer = tf.keras.optimizers.RMSprop(0.001)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/layer.py:393:
UserWarning: `build()` was called on layer 'network', however the
layer does not have a `build()` method implemented and it looks like
it has unbuilt state. This will cause the layer to be marked as built,
despite not being actually built, which may cause failures down the
line. Make sure to implement a proper `build()` method.
  warnings.warn(

Model: "network"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━
┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━
┩
│ dense (Dense)                        │ ?                           │
0 (unbuilt) │
├──────────────────────────────────────┼─────────────────────────────┼─
──────────────┤
│ dense_1 (Dense)                      │ ?                           │
0 (unbuilt) │
├──────────────────────────────────────┼─────────────────────────────┼─
──────────────┤
│ dense_2 (Dense)                      │ ?                           │
0 (unbuilt) │
└──────────────────────────────────────┴─────────────────────────────┴─
──────────────┘
```

```
 Total params: 0 (0.00 B)

 Trainable params: 0 (0.00 B)

 Non-trainable params: 0 (0.00 B)
```

```
#Next, implement the network training part.
#Through the double-layer loop training network composed of Epoch and
Step, a total of 200
#Epochs are trained.
for epoch in range(200): # 200 Epoch
 for step, (x,y) in enumerate(train_db): # Loop through training set
once
 # Set gradient tape
  with tf.GradientTape() as tape:
    out = model(x) # Get network output
    loss = tf.reduce_mean(keras.losses.mean_squared_error(y, out))  #
Calculate MSE
 # Calculate MSE
    mae_loss = tf.reduce_mean(keras.losses.mean_absolute_error(y,
out))  # Calculate MAE
 # Calculate MAE
  if step % 10 == 0: # Print training loss every 10 steps
    print(epoch, step, float(loss))
 # Calculate and update gradients
  grads = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

```
0 0 642.5635986328125
1 0 616.5436401367188
2 0 479.43792724609375
3 0 457.57733154296875
4 0 341.8994140625
5 0 342.6177062988281
6 0 221.7519989013672
7 0 137.39523315429688
8 0 81.50424194335938
9 0 66.85401153564453
10 0 28.310165405273438
11 0 17.649381637573242
12 0 25.856441497802734
13 0 21.489347457885742
14 0 8.428808212280273
15 0 16.974346160888672
16 0 8.164365768432617
17 0 6.144606590270996
18 0 7.341747283935547
19 0 12.277522087097168
20 0 8.451617240905762
21 0 6.8862624168396
```

```
22 0 9.072221755981445
23 0 11.132036209106445
24 0 5.3742218017578125
25 0 6.371731758117676
26 0 6.280424118041992
27 0 7.156704425811768
28 0 4.935726642608643
29 0 7.143799304962158
30 0 11.380046844482422
31 0 7.1176276206970215
32 0 6.185653209686279
33 0 7.505523204803467
34 0 4.204354286193848
35 0 8.01803970336914
36 0 4.7725067138671875
37 0 8.435873985290527
38 0 7.965193748474121
39 0 7.377237796783447
40 0 5.400815486907959
41 0 5.948269844055176
42 0 4.127633571624756
43 0 6.899347305297852
44 0 3.1742026805877686
45 0 7.352101802825928
46 0 4.375650405883789
47 0 7.812374591827393
48 0 7.526069641113281
49 0 5.086910724639893
50 0 4.287204742431641
51 0 3.6443400382995605
52 0 8.66423225402832
53 0 6.571686744689941
54 0 3.784060478210449
55 0 6.598706245422363
56 0 5.910862922668457
57 0 3.633151054382324
58 0 6.177596569061279
59 0 3.857351779937744
60 0 4.632004261016846
61 0 3.2200164794921875
62 0 4.068142414093018
63 0 8.488738059997559
64 0 5.056893348693848
65 0 5.4780683517456055
66 0 2.601785659790039
67 0 4.281123638153076
68 0 5.465846061706543
69 0 4.899689674377441
70 0 8.210004806518555
```

```
71 0 3.5678744316101074
72 0 9.537012100219727
73 0 7.302920818328857
74 0 5.171238422393799
75 0 5.413606643676758
76 0 3.1875338554382324
77 0 2.888105869293213
78 0 4.217975616455078
79 0 4.645174503326416
80 0 1.8717327117919922
81 0 6.203494071960449
82 0 3.602125406265259
83 0 3.335519790649414
84 0 4.971295356750488
85 0 3.0401768684387207
86 0 5.448073387145996
87 0 7.987362861633301
88 0 3.8066892623901367
89 0 3.7379133701324463
90 0 6.700997352600098
91 0 4.744523525238037
92 0 5.135449409484863
93 0 3.2504892349243164
94 0 4.832118988037109
95 0 4.259736061096191
96 0 3.288226366043091
97 0 4.5675554275512695
98 0 6.156699180603027
99 0 2.41693115234375
100 0 4.587540626525879
101 0 9.990177154541016
102 0 4.856575012207031
103 0 4.609582901000977
104 0 6.181292533874512
105 0 5.357446670532227
106 0 6.831010818481445
107 0 5.0209879875183105
108 0 5.246515274047852
109 0 5.610892295837402
110 0 1.8415918350219727
111 0 8.548630714416504
112 0 2.7810704708099365
113 0 6.845144271850586
114 0 4.809013366699219
115 0 5.4113874435424805
116 0 2.5574679374694824
117 0 3.0382487773895264
118 0 6.095086097717285
119 0 5.164796352386475
```

```
120 0 2.484609603881836
121 0 7.0102458000183105
122 0 6.210278511047363
123 0 5.484976768493652
124 0 6.937043190002441
125 0 3.0198843479156494
126 0 4.335622310638428
127 0 3.5974860191345215
128 0 4.954357147216797
129 0 4.311243534088135
130 0 5.5847272872924805
131 0 6.318434238433838
132 0 3.3971192836761475
133 0 4.048720359802246
134 0 3.1456053256988525
135 0 8.30167007446289
136 0 5.746269226074219
137 0 3.407099723815918
138 0 5.820940971374512
139 0 6.694208145141602
140 0 6.974605560302734
141 0 2.956892967224121
142 0 8.208221435546875
143 0 3.9922051429748535
144 0 3.2593674659729004
145 0 9.48184585571289
146 0 2.7541277408599854
147 0 4.374349594116211
148 0 6.2676286697387695
149 0 3.7122602462768555
150 0 3.097111701965332
151 0 7.082757949829102
152 0 2.8054018020629883
153 0 3.815640926361084
154 0 7.888819217681885
155 0 3.1115691661834717
156 0 5.453729629516602
157 0 6.592550277709961
158 0 2.5052082538604736
159 0 5.167928695678711
160 0 7.102262496948242
161 0 3.7863128185272217
162 0 3.4383325576782227
163 0 5.224042892456055
164 0 8.030010223388672
165 0 3.8777332305908203
166 0 5.2513017654418945
167 0 6.365333080291748
168 0 2.8833518028259277
```

```
169 0 2.268159866333008
170 0 5.658957004547119
171 0 1.7851099967956543
172 0 4.505185127258301
173 0 3.1681630611419678
174 0 7.459545135498047
175 0 6.28612756729126
176 0 5.09937858581543
177 0 4.460833549499512
178 0 4.301008701324463
179 0 5.45867919921875
180 0 3.506822109222412
181 0 6.404150009155273
182 0 6.99424934387207
183 0 6.113267421722412
184 0 4.83872127532959
185 0 4.795818328857422
186 0 4.453548431396484
187 0 6.2626447677612305
188 0 7.75407600402832
189 0 4.436655044555664
190 0 9.814167022705078
191 0 5.7964630126953125
192 0 5.75758695602417
193 0 4.679468631744385
194 0 3.2683730125427246
195 0 4.601797103881836
196 0 4.052825450897217
197 0 4.155115127563477
198 0 3.7126054763793945
199 0 8.212604522705078
```

Examples of Scenario-based Questions

Example 1: Fraud Detection System

A retail store wants to develop a fraud detection system using a single-layer perceptron. The system takes two input values:

- The number of items purchased
- The time of purchase (in hours from midnight)

If the weighted sum of inputs exceeds a certain threshold, the purchase is flagged as suspicious.

Tasks:

i. Implement a single-layer perceptron using TensorFlow that takes the number of items and time as input and applies a weighted sum followed by a step activation function.

ii. Modify the perceptron to include a bias term to improve classification performance.

Example 2: Sales Forecast Grid

Generate a temperature forecast grid for the next 7 days using tf.meshgrid(). Use tf.where() to identify cities where the temperature drops below 10 degrees Celsius and replace them with a minimum threshold of 10 degrees Celsius.

Example 3: Data Updates and Filtering

Update temperature readings at indices 2 and 5 with new values 30 degrees Celsius and 35 degrees Celsius using tf.scatter_nd(). Extract cities with temperatures greater than 33 degrees Celsius using tf.boolean_mask().

Example 4: Neural Network Design

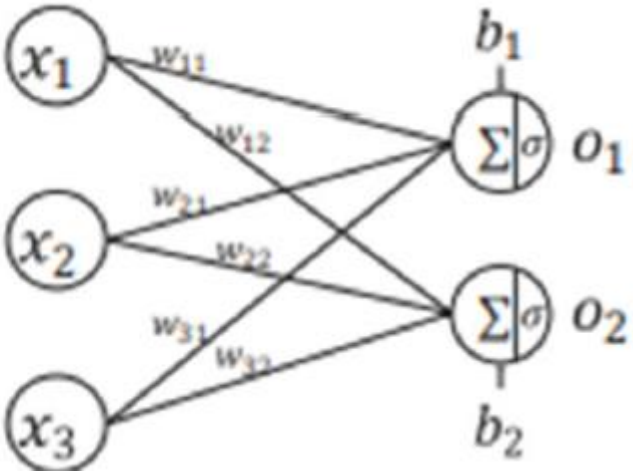
To design a fully connected neural network with 4 input neurons, 5 hidden layer neurons, and 3 output neurons using the ReLU activation function, follow these steps:

1. Define the structure of the network.
2. Randomly initialize the weights and biases for each layer.

3. Represent the weights and biases in matrix form.
4. Use the ReLU activation function to process the inputs.

Example 5:

Write an equation for outputs o1 and o1



From the given neural network diagram, we can derive the output equations for o_1 and o_2.

Each output neuron receives weighted inputs from all input neurons and applies an activation function σ. The equations for o_1 and o_2 can be written as:

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

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

where:

- w_{ij} represents the weight connecting input neuron x_i to output neuron o_j.
- b_1 and b_2 are bias terms for the respective output neurons.
- σ represents the activation function applied to the summed inputs.

This is a standard feedforward computation in a fully connected neural network layer.

Example 6: The CIFAR-10 dataset contains 60,000 color images of size 32×32 pixels across 10 classes. Before training a deep learning model, preprocessing is necessary. Explain:
Why is it essential to resize all images to a fixed size before training a deep learning model?
How does normalizing these pixel values (e.g., scaling them to the range [0,1] or [-1,1]) improve model performance?
What factors should be considered when splitting the CIFAR-10 dataset into training, validation, and test sets?
Why is it necessary to apply one-hot encoding to the class labels instead of using categorical values?
Answer:

Why is it essential to resize all images to a fixed size before training a deep learning model?

Resizing all images to a fixed size ensures consistent input dimensions, as neural networks expect inputs of uniform shape. This is important for computational efficiency, as fixed-size images allow for manageable computations and help the model process data in a uniform manner. Moreover, models that have been pre-trained on fixed-size images require the same input size for compatibility.

2. How does normalizing these pixel values (e.g., scaling them to the range [0,1] or [-1,1]) improve model performance?

Normalization improves convergence during training by ensuring that input values are within a manageable range, preventing large gradients that could destabilize learning. It helps the activation functions (e.g., sigmoid, tanh) work more effectively and consistently, thus speeding up the training process. Normalized inputs allow the network to learn faster and more efficiently by maintaining stability and avoiding issues like vanishing/exploding gradients.

. What factors should be considered when splitting the CIFAR-10 dataset into training, validation, and test sets?

When splitting the CIFAR-10 dataset, it's important to maintain the class distribution (stratification) to avoid class imbalances in the splits. The training set should be large (typically 70-80%) to ensure the model has enough data to learn. The validation set (10-15%) is used for hyperparameter tuning, while the test set (10-20%) should remain untouched for final evaluation. Additionally, data should be shuffled to ensure randomness.

4. Why is it necessary to apply one-hot encoding to the class labels instead of using categorical values?

One-hot encoding is necessary because it represents the target labels as independent vectors, allowing the model to output probabilities for each class using softmax. Categorical values could imply an ordinal relationship that doesn't exist between classes. One-hot encoding prevents this issue and ensures compatibility with common loss functions like categorical cross-entropy, improving the model's ability to learn and make accurate predictions