Introduction to TensorFlow and Keras

What are TensorFlow?

TensorFlow is general purpose **Python programming language** based open-source end-to-end platform Developed by Google Brain Team for creating **Machine Learning applications**.

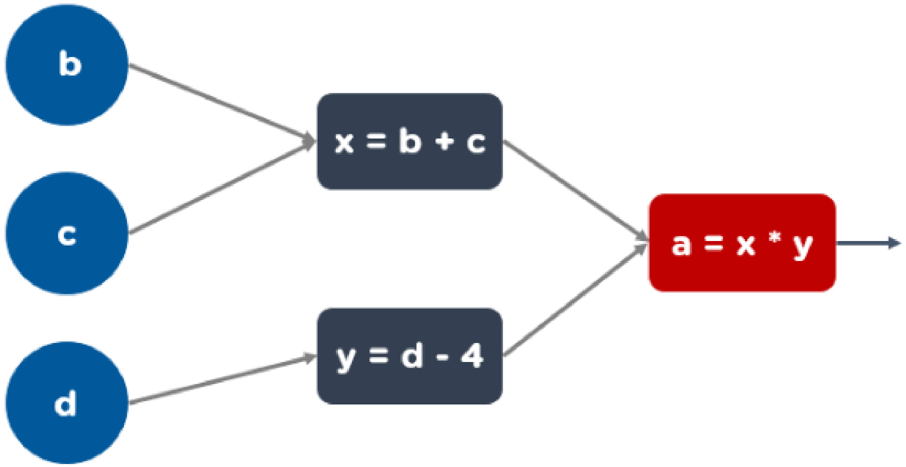
It is one of the most popular programming platform for high dimensional computation and implementing complex deep learning models.

TensorFlow

$$\begin{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} & \begin{bmatrix} 3 & 4 \end{bmatrix} \\ \begin{bmatrix} 5 & 6 \end{bmatrix} & \begin{bmatrix} 7 & 8 \end{bmatrix} \end{bmatrix}$$

Tensor

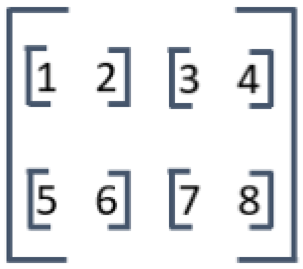
+



Data Flow Graph

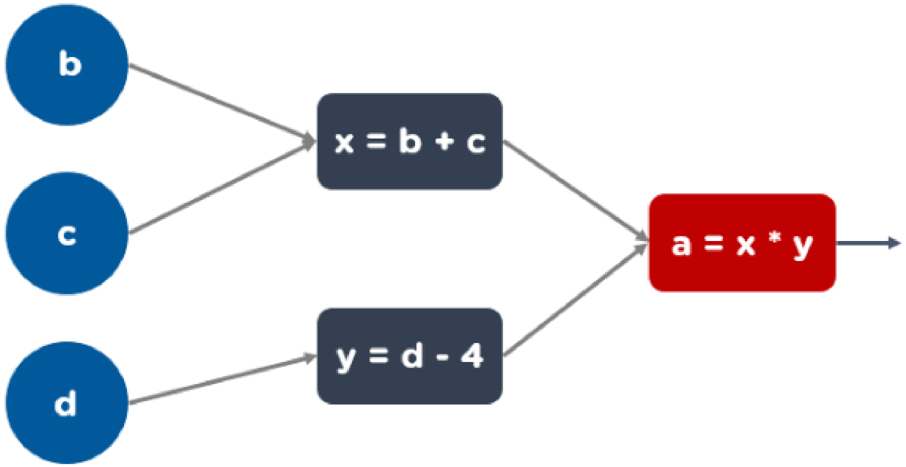
TensorFlow

Tensor + Data Flow Graph



Tensor

+



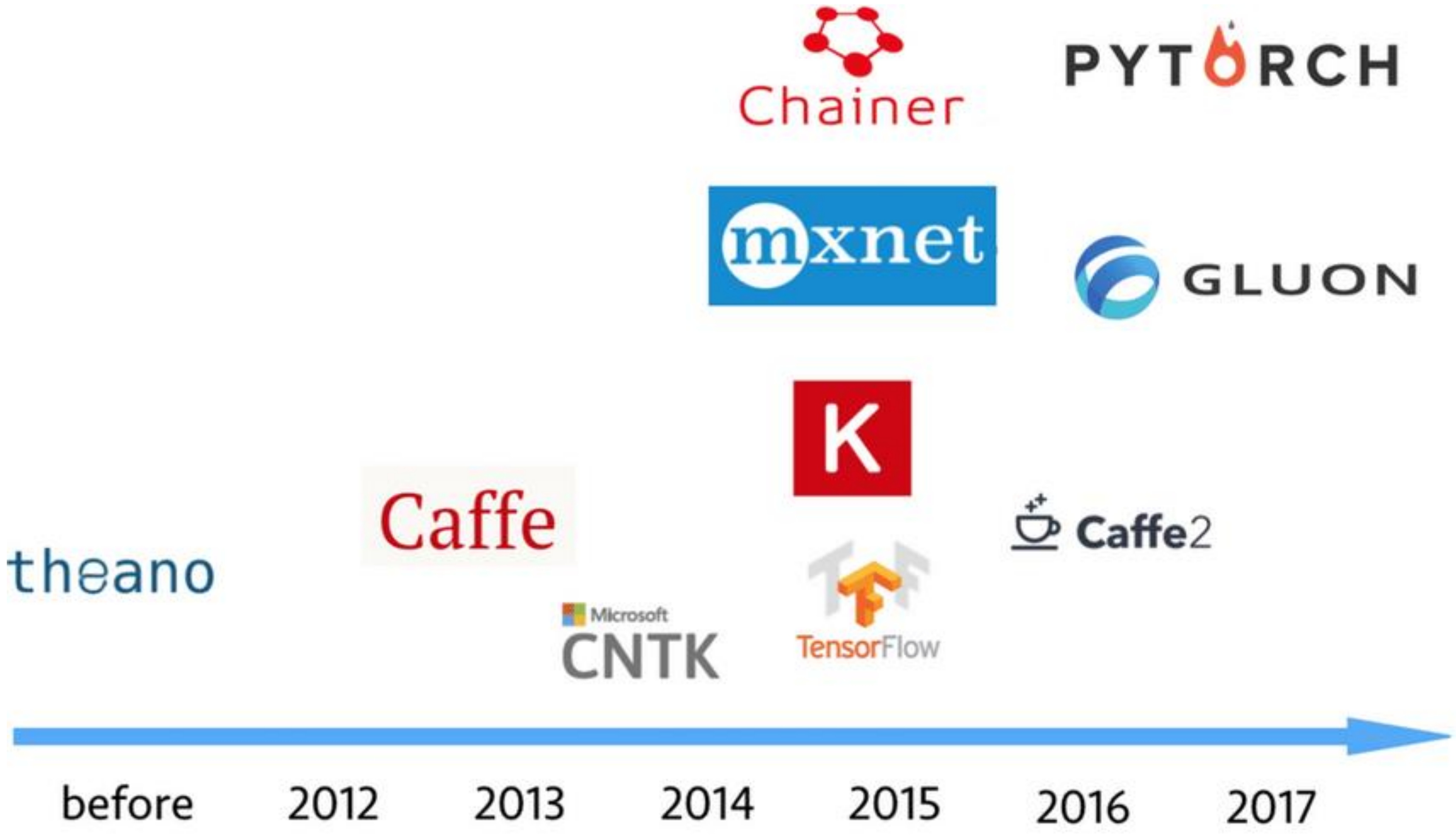
TensorFlow 2.0

- Eager execution, allowing to build the models and run instantly.
- Keras, high-level API for different Deep Learning Models, is incorporated with TensorFlow 2.0.
- Using Keras API, building complex deep learning models becomes a trivial task
- The size of the program becomes much smaller in 2.0 as compared to 1.0

TensorFlow

Keras

Different Deep Learning Frameworks



[
However, because of open source availability, easy to use, and high portability other programming platforms, we will be using TensorFlow and Keras.

At the end of this module, you will learn

- **How to write algebraic programs using TensorFlow?**
- **How to visualize the computational flow using Data Flow Graph**
- **How to implement neural networks?**
- **How to estimate parameters?**
- **How to simplify implementation of neural models using Keras?**

]

Lesson 15

What are Tensor?

What are Tensors?

What are Tensors?

Wikipedia:

A **tensor** is an algebraic object that describes a multilinear relationship between sets of algebraic objects related to a vector space.

In short, tensors are generalization of scalars and vectors

What are Tensors?

Wikipedia:

A **tensor** is an algebraic object that describes a multilinear relationship between sets of algebraic objects related to a vector space.

In short, tensors are generalization of scalars data and vectors.

0d Tensor 1d Tensor 2d Tensor 3d Tensor 4d Tensor

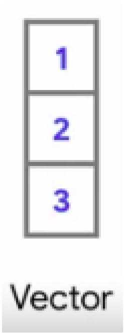
Types of Tensors?

Rank = 0

Shape = (0)

Types of Tensors?

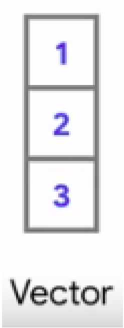
Rank = 0 Shape = (0)



Rank = 1 Shape = (3)

Types of Tensors?

Rank = 0 Shape = (0)



Rank = 1 Shape = (3) 3 number of Rank 0 tensors

Types of Tensors?

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \\ 7 & 8 \end{bmatrix}$$

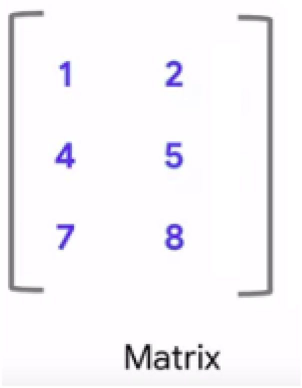
Matrix

Rank = 2

Shape = (3x2)

3 number of Rank 1 tensors of shape 2

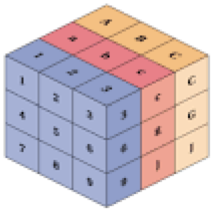
Types of Tensors?



Rank = 2

Shape = (3x2)

3 number of Rank 1 tensors of shape 2

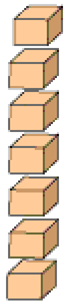


Rank = 3

Shape = (3x3x3)

3 number of Rank 2 tensors of shape 3x3

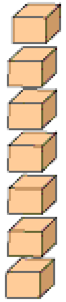
Types of Tensors?



4 d -Tensor

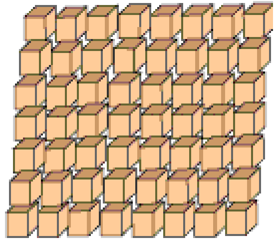
Rank = 4 Shape = (7x3x3x3) 7 number of Rank 3 tensors of shape 3x3x3

Types of Tensors?



4 d -Tensor

Rank = 4 Shape = (7x3x3x3) 7 number of Rank 3 tensors of shape 3x3x3



5 d -Tensor

Rank = 5 Shape = (7x8x3x3x3) 7 number of Rank 4 tensors of shape 8x3x3x3

Tensors in TensorFlow

```
import tensorflow as tf

t = tf.constant( 4 )
print( t )
```

```
tf.Tensor(    4,       shape=(),     dtype=int32   )
```

Tensors in TensorFlow

```
import tensorflow as tf

t = tf.constant( 4 )
print( t )
```

```
tf.Tensor(    4,       shape=(),     dtype=int32   )
```

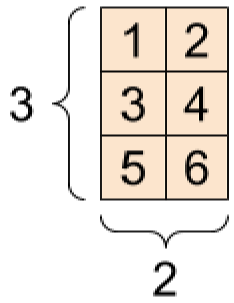
```
import tensorflow as tf

t = tf.constant( [2.0, 3.0, 4.0] )
print(t)
```

```
tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)
```

Tensors in TensorFlow

```
import tensorflow as tf

t = tf.constant( [    [1, 2],
                      [3, 4],
                      [5, 6]
                 ])
print(t)
```

```
tf.Tensor( [[1 2] [3 4] [5 6]], shape=(3, 2), dtype=float16
```



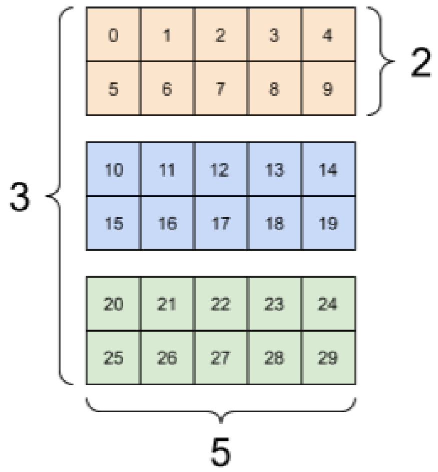
No. of rows = 3

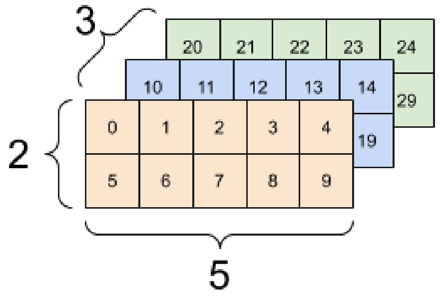
No. of Columns = 2

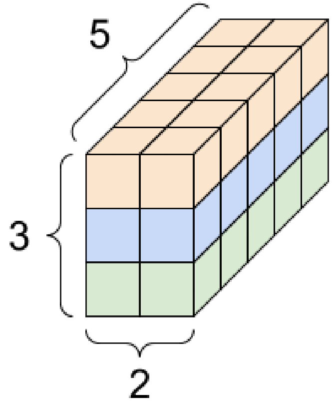
Tensors in TensorFlow

```
import tensorflow as tf

t = tf.constant([
                [ [0, 1, 2, 3, 4],
                  [5, 6, 7, 8, 9] ],
                [ [10, 11, 12, 13, 14],
                  [15, 16, 17, 18, 19] ],
               [ [20, 21, 22, 23, 24],
                 [25, 26, 27, 28, 29] ]
              ])

print(t)
```

tf.Tensor([[[0 1 2 3 4] [5 6 7 8 9]] [[10 11 12 13 14] [15 16 17 18 19]] [[20 21 22 23 24] [25 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)







Components of a Tensor?

```
import tensorflow as tf

t = tf.constant( [2.0, 3.0, 4.0] )
print(t)
```

```
tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)
```

Values

Shape (Rank)

Data type

Summary

- What are Tensors?
- Different types of Tensors
- Rank of a Tensor
- Shape of a Tensor
- Components of a Tensor

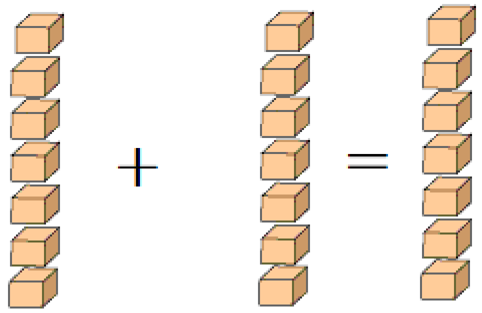
Lesson 17:

Arithmetic Operations on Tensors

Operations between two Tensors

Given two tensors x and y,

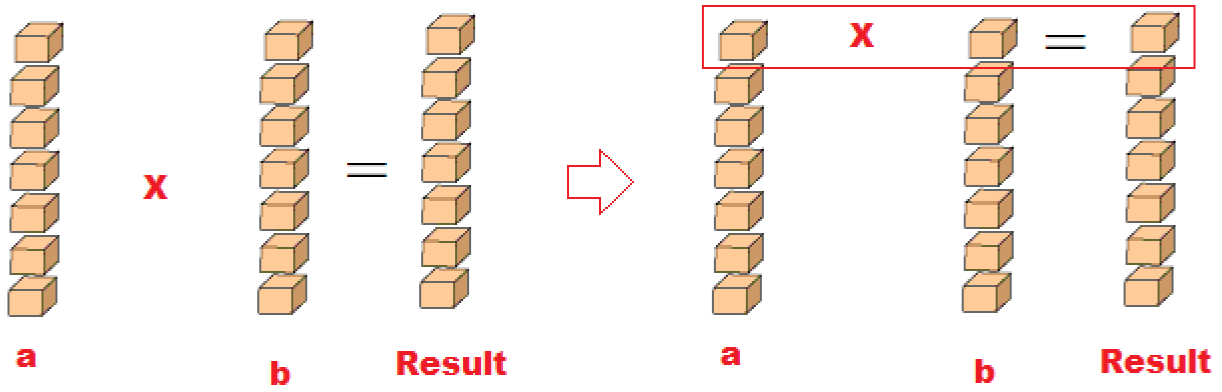
- Arithmetic operations such as **plus, minus, multiplication, division** can be performed between x and y, and produce another tensor.
    - x + y, x-y, x*y, x/y
- To perform a binary operation between two tensors, the shape of the two should be compatible.
- Element wise operations between the two tensors are performed.



Operations between two Tensors

Given two tensors x and y,

- Arithmetic operations such as **plus, minus, multiplication, division** can be performed between x and y, and produce another tensor.
  - x + y, x-y, x*y, x/y
- To perform a binary operation between two tensors, the shape of the two should be compatible.
- Element wise operations between the two tensors are performed.



Broadcast (Stretch)

X (1d array):	3
Y (1d array	1
Result (1d array) :	3

Broadcast (Stretch)

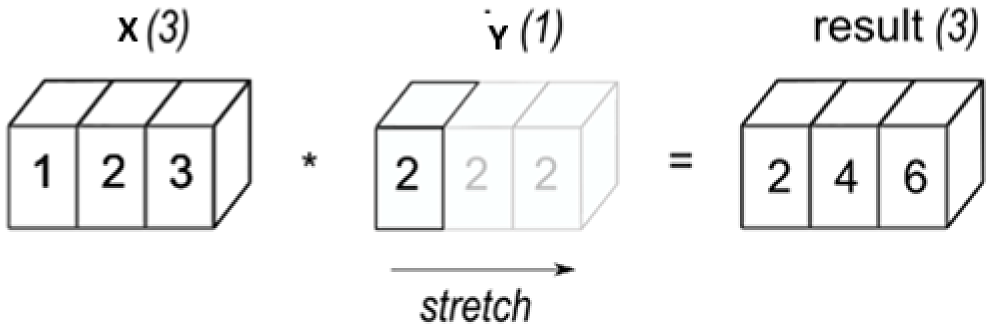
X (1d array):	3	[1, 2, 3]
Y (1d array	1	2
Result (1d array) :	3	

Broadcast (Stretch)

X (1d array):	3	[1, 2, 3]
Y (1d array	1	2
Result (1d array) :	3	

TensorFlow performs broadcast of the lower shape tensor.

It means, the low dimensional tensor is replicated till we find the matching shape

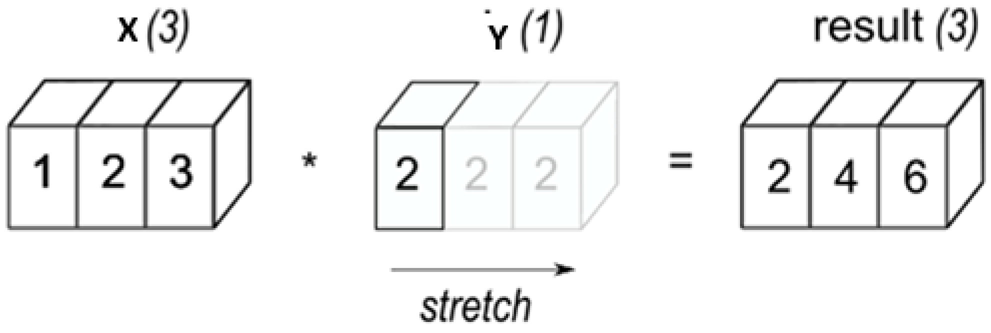


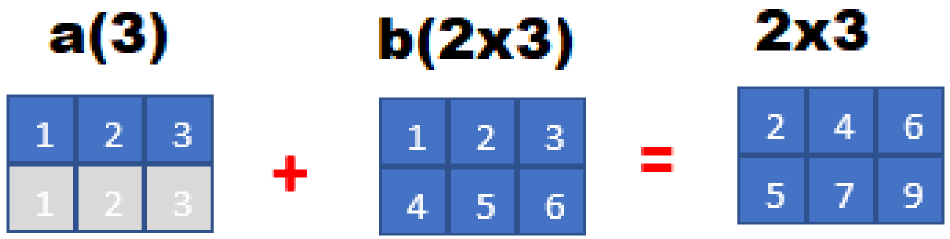
Broadcast (Stretch)

X (1d array):	3	[1, 2, 3]
Y (1d array	1	2
Result (1d array) :	3	

TensorFlow performs broadcast of the lower shape tensor.

It means, the low dimensional tensor is replicated till we find the matching shape



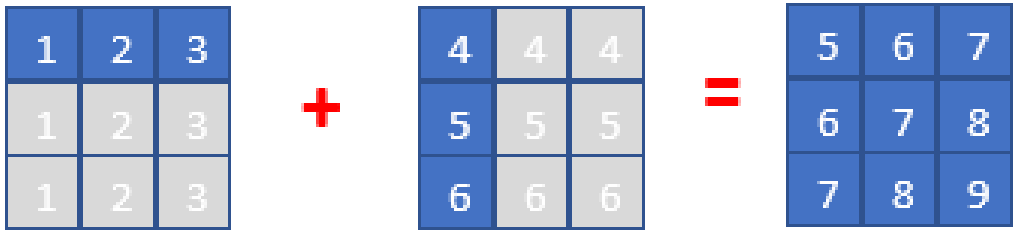


Two Sides Broadcast (Stretch)

X (1d array): 3

Y (2d array): 3 x 1

Result (2d array): 3 x 3



Lesson 18: Compatibility between two tensors

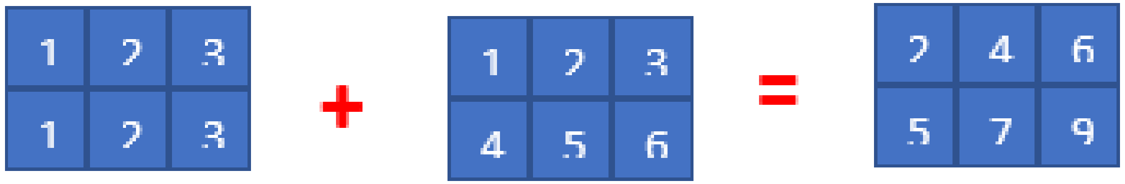
Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

- Their dimensions and shapes are same
  - Dimension(x) = Dimension(y)
  - Shape(x) = Shape(y)



X (2d array): 2 x 4

Y (2d array): 2 x 4

Result (2d array) : 2 x 4

Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

- Their dimensions and shapes are same
  - Dimension(x) = Dimension(y)
  - Shape(x) = Shape(y)

X (2d array):	2	x	4
Y (2d array):	2	x	4
Result (2d array) :	2	x	4

- Their dimensions or shapes are different, but the following condition is satisfied.
  - For all dimension position, one of component dimension has shape 1.

X (2d array):	2	x	1
Y (2d array):	2	x	4
Result (2d array) :	2	x	4

Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

- Their dimensions and shapes are same
  - Dimension(x) = Dimension(y)
  - Shape(x) = Shape(y)

X (2d array):	2	x	4
Y (2d array):	2	x	4
Result (2d array) :	2	x	4

- Their dimensions or shapes are different, but the following condition is satisfied.
  - For all dimension position, one of component dimension has shape 1.

X (2d array):	2	x	1
Y (2d array):	2	x	4
Result (2d array) :	2	x	4

X (3d array):	2	x	1	x	3
Y (3d array):	2	x	4	x	1
Result (3d array) :	2	x	4	x	3

Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

- Their dimensions and shapes are same
  - Dimension(x) = Dimension(y)
  - Shape(x) = Shape(y)

X (2d array):	2	x	4
Y (2d array):	2	x	4
Result (2d array) :	2	x	4

- Their dimensions or shapes are different, but the following condition is satisfied.
  - For all dimension position, one of component dimension has shape 1.

X (2d array):	2	x	1
Y (2d array):	2	x	4
Result (2d array) :	2	x	4

X (3d array):	2	x	1	x	3
Y (3d array):	2	x	4	x	1
Result (3d array) :	2	x	4	x	3

X (3d array):	2	x	1	x	3
Y (3d array):	1	x	4	x	1
Result (3d array) :	2	x	4	x	3

Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

- Their dimensions and shapes are same
  - Dimension(x) = Dimension(y)
  - Shape(x) = Shape(y)

X (2d array):	2	x	3
Y (2d array):	2	x	4
Result (2d array) :	2	x	4

- Their dimensions and/or shapes are different, but the following condition is satisfied.
  - For all dimension position, one of component dimension has shape 1.

a:	(2d array):	256	x	3
b:	(1d array):			3
Result:	(2d array):	256	x	3

Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

- Their dimensions and shapes are same
  - Dimension(x) = Dimension(y)
  - Shape(x) = Shape(y)

X (2d array): 2 x 3

Y (2d array): 2 x 4

Result (2d array) : 2 x 4

- Their dimensions and/or shapes are different, but the following condition is satisfied.
  - For all dimension position, one of component dimension has shape 1.

a: (2d array): 256 x 3

b: (1d array): 3

Result: (2d array): 256 x 3

We assume that, we have default 1.

(3) ~ (1 x 3)

(2 x 3) ~ (1 x 2 x 3)

Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

- Their dimensions and shapes are same
  - Dimension(x) = Dimension(y)
  - Shape(x) = Shape(y)

X (2d array): 2 x 3
Y (2d array): 2 x 4
Result (2d array) : 2 x 4

- Their dimensions and/or shapes are different, but the following condition is satisfied.
  - For all dimension position, one of component dimension has shape 1.

```
     a: (2d array): 256 x 3
     b: (1d array):       3
Result: (2d array): 256 x 3
```

~

```
     a: (2d array): 256 x 3
     b: (1d array):   1 x 3
Result: (2d array): 256 x 3
```

Compatibility of two Tensors for Arithmetic Operation

Two tensors x and y are said to be compatible if

- Their dimensions and shapes are same
  - Dimension(x) = Dimension(y)
  - Shape(x) = Shape(y)

```
X (2d array):      2 x 3
Y (2d array):      2 x 4
Result (2d array): 2 x 4
```

- Their dimensions and/or shapes are different, but the following condition is satisfied.
  - For all dimension position, one of component dimension has shape 1.

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

The resultant dimension is

- Higher dimension
- Higher shape of the component dimensions

They are not compatible

```
A      (1d array):  3
B      (1d array):  4 # trailing dimensions do not match

A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3 # second from last dimensions mismatched
```

Summary

- Operations are performed element wise.
- If the shapes between the two tensors are different, but compatible, the tensor with smaller shape is stretched.

Lesson 18: More on Tensor

Variable Tensor

A variable tensor is created using ***tf.Variable()*** function.

Syntax: *tf.Variable(initial_value=None, trainable=None, validate_shape=True, caching_device=None, name=None, variable_def=None, dtype=None, import_scope=None, constraint=None, synchronization=tf.VariableSynchronization.AUTO, aggregation=tf.compat.v1.VariableAggregation.NONE, shape=None)*

- ***initial_value****: by default None. The initial value for the Variable is a Tensor, or a Python object convertible to a Tensor.*
- ***trainable****: by default None. If True, GradientTapes will keep an eye on this variable's usage.*
- ***validate_shape****: by default True. Allows the variable to be initialised with an unknown shape value if False. The shape of initial value must be known if True, which is the default.*
- ***name****:by default None. The variable's optional name. Defaults to 'Variable' and is automatically uniquified.*
- ***variable_def:*** *by default None.*
- ***dtype:*** *by default None. If set, initial_value will be converted to the given type. If None, either the datatype will be kept (if initial_value is a Tensor), or convert_to_tensor will decide.*
- ***shape:*** *by default None. if None the shape of initial_value will be used. if any shape is specified, the variable will be assigned with that particular shape.*

Few examples of Variable Tensor

```
import tensorflow as tf

x = tf.Variable( [1, 2, 3, 4] )

x = tf.Variable( [1.2, 4.4, 5, 6] )

x = tf.Variable( ['a’, 'b’, 'c’, 'd’] )

x = tf.Variable( [True, False] )

x = tf.Variable( [3 + 4j] )
```

```python
import tensorflow as tf

x = tf.Variable([1,2,3,4])
print(x)
```

```
<tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4])>
```

```python
x = tf.Variable([[1,2,3,4],[5,6,7,8]])
print(x)
```

```
<tf.Variable 'Variable:0' shape=(2, 4) dtype=int32, numpy= array([[1, 2, 3, 4],
                                                                  [5, 6, 7, 8]])>
```

Find the attributes of Tensor

```
x = tf.Variable([1,2,3,4])
print(x.name)

print(x.shape)

print(x.dtype)

print(x.numpy())
```

```
Variable:0
(4,)
<dtype: 'int32'>
[1 2 3 4]
```

```
x = tf.Variable([[1,2,3,4],[5,6,7,8]])
print(x.name)

print(x.shape)

print(x.dtype)

print(x.numpy())
```

```
Variable:0
(2, 4)
<dtype: 'int32'>
[[1 2 3 4]
 [5 6 7 8]]
```

Find the attributes of Tensor

```python
x = tf.Variable([1,2,3,4])
print(x.name)

print(x.shape)

print(x.dtype)

print(x.numpy())
```

```
Variable:0
(4,)
<dtype: 'int32'>
[1 2 3 4]
```

```python
x = tf.constant([1,2,3,4])
#print(x.name)      #possible, when eager execution is disabled

print(x.shape)

print(x.dtype)

print(x.numpy())
```

```
(4,)
<dtype: 'int32'>
[1 2 3 4]
```

Constant tensor can be converted to Variable tensor

```
x_con = tf.constant([1,2,3,4])

x_var = tf.Variable(t_con)
print(x_var)
```

```
<tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4])>
```

Variable tensor can be converted to Constant tensor

```
x_var = tf.Variable([1,2,3,4])
x_con = tf.constant(x_var)
print(x_con)
```

```
tf.Tensor([1 2 3 4], shape=(4,), dtype=int32)
```

Include data type as parameter

```
x = tf.constant([1,2,3,4], dtype=tf.float32)
print(x)
```

```
tf.Tensor([1. 2. 3. 4.], shape=(4,), dtype=float32)
```

Lesson 20

Reshape the structure of tensor

```
x = tf.constant([1,2,3,4,5,6])
print(x)
```

```
tf.Tensor([1 2 3 4 5 6], shape=(6,), dtype=int32)
```

[1, 2, 3, 4, 5, 6]

Reshape the structure of tensor

```
x = tf.constant([1,2,3,4,5,6])
print(x)
```

```
tf.Tensor([1 2 3 4 5 6], shape=(6,), dtype=int32)
```

[1, 2, 3, 4, 5, 6]

```
x = tf.constant([1,2,3,4,5,6], shape=(2,3))
print(x)
```

```
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
```

[1, 2, 3]
[4, 5, 6]

Reshape the structure of tensor

```
x = tf.constant([1,2,3,4,5,6])
print(x)
```

```
tf.Tensor([1 2 3 4 5 6], shape=(6,), dtype=int32)
```

[1, 2, 3, 4, 5, 6]

```
x = tf.constant([1,2,3,4,5,6], shape=(2,3))
print(x)
```

```
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
```

[1, 2, 3]
[4, 5, 6]

[1, 2]
[3, 4]
[5, 6]

Reshape the structure of tensor

```
x = tf.constant([1,2,3,4,5,6])
print(x)
```

```
tf.Tensor([1 2 3 4 5 6], shape=(6,), dtype=int32)
```

[1, 2, 3, 4, 5, 6]

```
x = tf.constant([1,2,3,4,5,6], shape=(2,3))
print(x)
```

```
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
```

[1, 2, 3]
[4, 5, 6]

```
tf.reshape(x, (3,2))
```

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[1, 2],
       [3, 4],
       [5, 6]])>
```

[1, 2]
[3, 4]
[5, 6]

Flatten the tensor

[1, 2, 3, 4]
[5, 6, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8]

```
x = tf.constant([[1,2,3,4],[5,6,7,8]], shape=(8))
print(x)
```

```
tf.Tensor([1 2 3 4 5 6 7 8], shape=(8,), dtype=int32)
```

Flatten the tensor

[1, 2, 3, 4]
[5, 6, 7, 8]

→

[1, 2, 3, 4, 5, 6, 7, 8]

```
x = tf.constant([[1,2,3,4],[5,6,7,8]], shape=(8))
print(x)
```

```
tf.Tensor([1 2 3 4 5 6 7 8], shape=(8,), dtype=int32)
```

```
x = tf.constant([[1,2,3,4],[5,6,7,8]])
tf.reshape(x, (8))
```

```
<tf.Tensor: shape=(8,), dtype=int32, numpy=array([1, 2, 3, 4, 5, 6, 7, 8])>
```

Reshape the tensor with (-1)

```
x = tf.constant([[1,2,3,4],[5,6,7,8]])
print(x)
```

```
tf.Tensor(
[[1 2 3 4]
 [5 6 7 8]], shape=(2, 4), dtype=int32)
```

[1, 2, 3, 4]
[5, 6, 7, 8]

```
tf.reshape(x, (-1))
```

```
<tf.Tensor: shape=(6,), dtype=int32, numpy=array([1, 2, 3, 4, 5, 6])>
```

[1, 2, 3, 4, 5, 6, 7, 8]

More on (-1) shape

```
x = tf.constant([[1,2,3,4],[5,6,7,8]])
print(x)
```

```
tf.Tensor(
[[1 2 3 4]
 [5 6 7 8]], shape=(2, 4), dtype=int32)
```

[1, 2, 3, 4]
[5, 6, 7, 8]

```
tf.reshape(x, (4,-1))
```

```
<tf.Tensor: shape=(4, 2), dtype=int32, numpy=
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])>
```

[1, 2]
[3, 4]
[5, 6]
[7, 8]

More on (-1) shape

```
x = tf.constant([1,2,3,4,5,6,7,8])
tf.reshape(x, (2,-1,2))
```

```
<tf.Tensor: shape=(2, 2, 2), dtype=int32, numpy=
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])>
```

```
[ [ 1, 2]
  [ 3, 4] ]
[ [ 5, 6]
  [ 7, 8] ]
```

More on (-1) shape

```
x = tf.constant([1,2,3,4,5,6,7,8])
tf.reshape(x, (2,-1,2))
```

```
<tf.Tensor: shape=(2, 2, 2), dtype=int32, numpy=
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])>
```

```
x = tf.constant([[1,2,3,4],[5,6,7,8]])
tf.reshape(x, (-1,2,2))
```

```
<tf.Tensor: shape=(2, 2, 2), dtype=int32, numpy=
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])>
```

```
[ [ 1, 2]
  [ 3, 4] ]
[ [ 5, 6]
  [ 7, 8] ]
```

More on (-1) shape

```
x = tf.constant([1,2,3,4,5,6,7,8])
tf.reshape(x, (2,-1,2))
```

```
<tf.Tensor: shape=(2, 2, 2), dtype=int32, numpy=
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])>
```

```
[ [ 1, 2]
  [ 3, 4] ]
[ [ 5, 6]
  [ 7, 8] ]
```

```
x = tf.constant([[1,2,3,4],[5,6,7,8]])
tf.reshape(x, (-1,2,2))
```

```
<tf.Tensor: shape=(2, 2, 2), dtype=int32, numpy=
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])>
```

```
x = tf.constant([[1,2,3,4],[5,6,7,8]])
tf.reshape(x, (2,2,-1))
```

```
<tf.Tensor: shape=(2, 2, 2), dtype=int32, numpy=
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])>
```

Summary

We have learnt

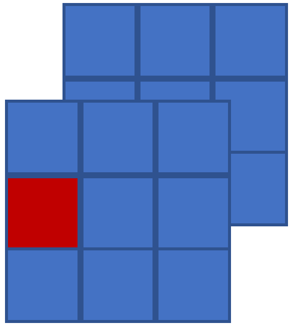
- How to create a variable tensor?
- How to convert a constant tensor to a variable tensor, and vice versa?
- How to reshape a tensor?

Lesson 22: Accessing the element of a Tensor

Extract a slice from a tensor

tf.slice(<input>,<begin>,<size>)

- input: Tensor
- begin: starting location for each dimension of input
- size: number of elements for each dimension of input, using -1 includes all remaining elements



Shape: [2, 3, 3]

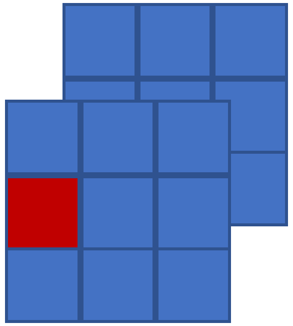
Begin: [0, 1, 0]

Size: [1, 1, 1]

Extract a slice from a tensor

tf.slice(<input>,<begin>,<size>)

- input: Tensor
- begin: starting location for each dimension of input
- size: number of elements for each dimension of input, using -1 includes all remaining elements



Shape: [2, 3, 3]

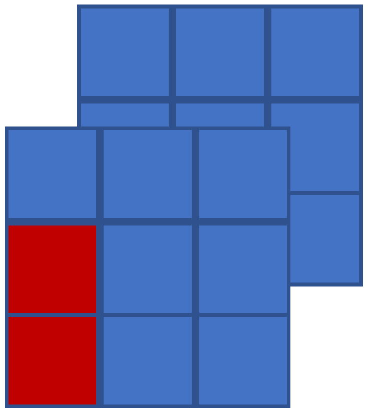
Begin: [0, 1, 0]

Size: [1, 1, 1]

```
import tensorflow as tf
x = tf.constant([[[1., 2., 3.], [4., 5.,6 ], [7. , 8.,9 ]],
                 [[10., 11.,12], [13., 14., 15], [16., 17., 18]]])
print(x)


res = tf.slice(x, [0, 1, 0], [1, 1, 1])
print("\n")
print(res)
```

```
tf.Tensor(
[[[ 1.  2.  3.]
  [ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[10. 11. 12.]
  [13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 3, 3), dtype=float32)


tf.Tensor([[[4.]]], shape=(1, 1, 1), dtype=float32)
```

Extract a slice from a tensor

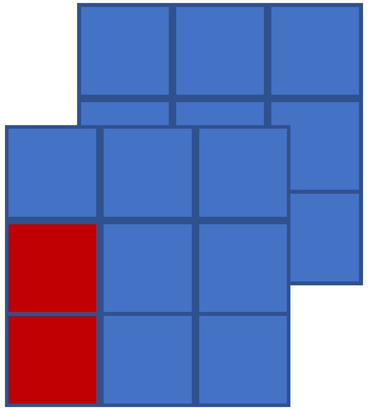


Shape: [2, 3, 3]

Begin: [0, 1, 0]

Size:

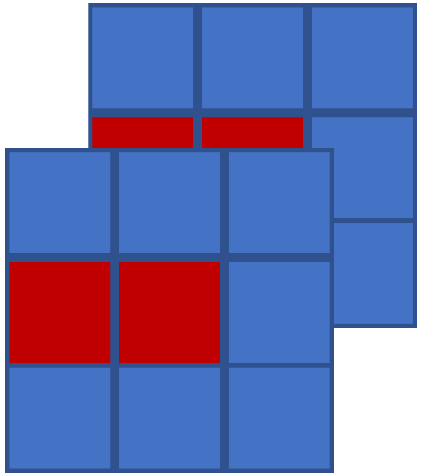
Extract a slice from a tensor



```
import tensorflow as tf
x = tf.constant([[[1., 2., 3.], [4., 5.,6 ], [7. , 8.,9 ]],
                 [[10., 11.,12], [13., 14., 15], [16., 17., 18]]])
print(x)


res = tf.slice(x, [0, 1, 0], [1, 2, 1])
print("\n")
print(res)
```

```
tf.Tensor(
[[[ 1.  2.  3.]
  [ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[10. 11. 12.]
  [13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 3, 3), dtype=float32)


tf.Tensor(
[[[4.]
  [7.]]], shape=(1, 2, 1), dtype=float32)
```

Extract a slice from a tensor

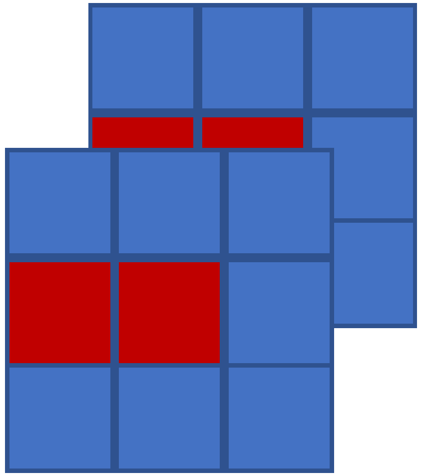


Shape: [2, 3, 3]

Begin: [0, 1, 0]

Size:

Extract a slice from a tensor



Shape: [2, 3, 3]

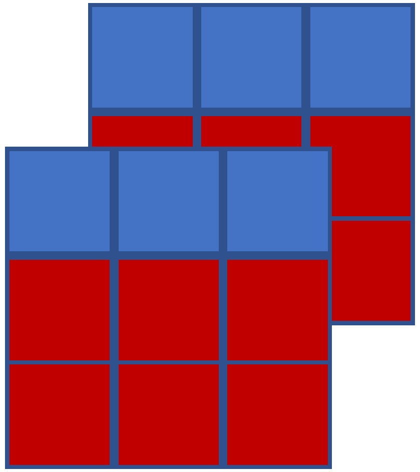
Begin: [0, 1, 0]

Size:

```
import tensorflow as tf
x = tf.constant([[[1., 2., 3.], [4., 5.,6 ], [7. , 8.,9 ]],
                 [[10., 11.,12], [13., 14., 15], [16., 17., 18]]])
print(x)


res = tf.slice(x, [0, 1, 0], [2, 1, 2])
print("\n")
print(res)
```

```
tf.Tensor(
[[[ 1.  2.  3.]
  [ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[10. 11. 12.]
  [13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 3, 3), dtype=float32)


tf.Tensor(
[[[ 4.  5.]]

 [[13. 14.]]], shape=(2, 1, 2), dtype=float32)
```

Extract a slice from a tensor

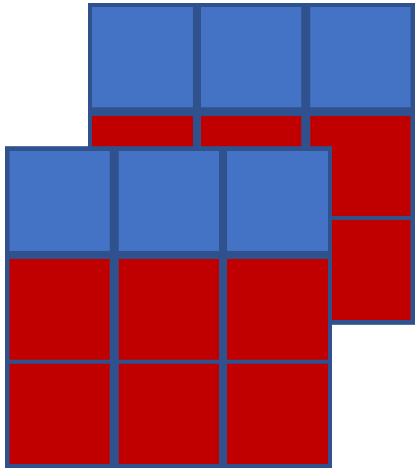


Shape: [2, 3, 3]

Begin: [0, 1, 0]

Size:

Extract a slice from a tensor



Shape: [2, 3, 3]

Begin: [0, 1, 0]

Size:

```
import tensorflow as tf
x = tf.constant([[[1., 2., 3.], [4., 5.,6 ], [7. , 8.,9 ]],
                 [[10., 11.,12], [13., 14., 15], [16., 17., 18]]])
print(x)


res = tf.slice(x, [0, 1, 0], [-1, -1, -1])
print("\n")
print(res)
```

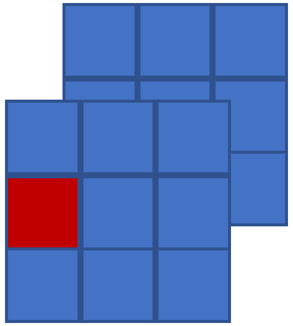
```
tf.Tensor(
[[[ 1.  2.  3.]
  [ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[10. 11. 12.]
  [13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 3, 3), dtype=float32)


tf.Tensor(
[[[ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 2, 3), dtype=float32)
```

Lesson 5: Accessing the element of a Tensor

Extract a slice from a tensor

tf.slice(<input>,<begin>,<size>)

- input: Tensor
- begin: starting location for each dimension of input
- size: number of elements for each dimension of input, using -1 includes all remaining elements



Shape: [2, 3, 3]

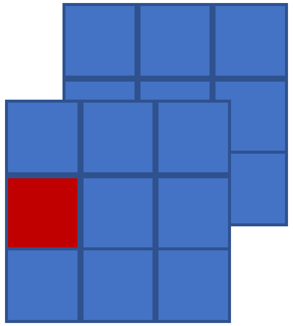
Begin: [0, 1, 0]

Size: [1, 1, 1]

Extract a slice from a tensor

tf.slice(<input>,<begin>,<size>)

- input: Tensor
- begin: starting location for each dimension of input
- size: number of elements for each dimension of input, using -1 includes all remaining elements



Shape: [2, 3, 3]

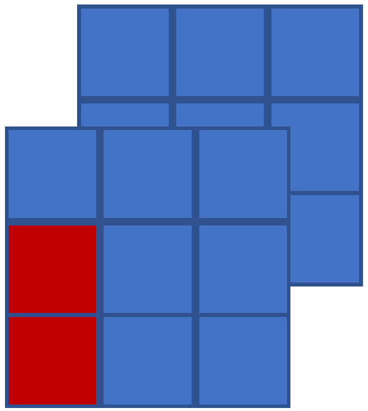
Begin: [0, 1, 0]

Size: [1, 1, 1]

```python
import tensorflow as tf
x = tf.constant([[[1., 2., 3.], [4., 5.,6 ], [7. , 8.,9 ]],
                 [[10., 11.,12], [13., 14., 15], [16., 17., 18]]])
print(x)


res = tf.slice(x, [0, 1, 0], [1, 1, 1])
print("\n")
print(res)
```

```
tf.Tensor(
[[[ 1.  2.  3.]
  [ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[10. 11. 12.]
  [13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 3, 3), dtype=float32)


tf.Tensor([[[4.]]], shape=(1, 1, 1), dtype=float32)
```

Extract a slice from a tensor

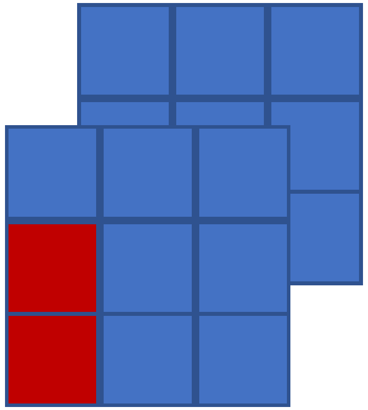


Shape: [2, 3, 3]

Begin: [0, 1, 0]

Size:

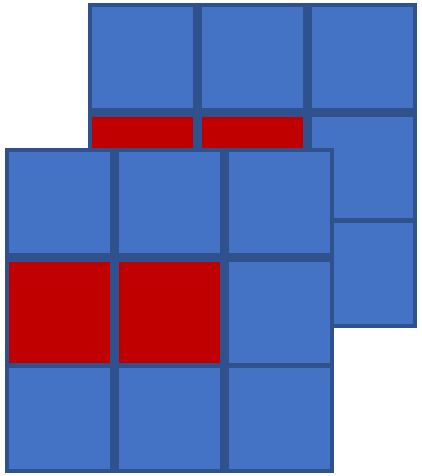
Extract a slice from a tensor



```
import tensorflow as tf
x = tf.constant([[[1., 2., 3.], [4., 5.,6 ], [7. , 8.,9 ]],
                  [[10., 11.,12], [13., 14., 15], [16., 17., 18]]])
print(x)


res = tf.slice(x, [0, 1, 0], [1, 2, 1])
print("\n")
print(res)
```

```
tf.Tensor(
[[[ 1.  2.  3.]
  [ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[10. 11. 12.]
  [13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 3, 3), dtype=float32)


tf.Tensor(
[[[4.]
  [7.]]], shape=(1, 2, 1), dtype=float32)
```

Extract a slice from a tensor

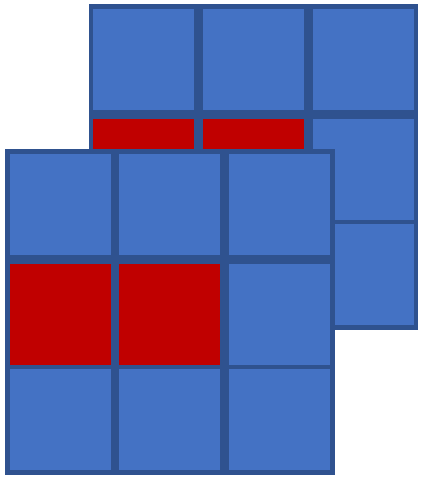


Shape: [2, 3, 3]

Begin: [0, 1, 0]

Size:

Extract a slice from a tensor



Shape: [2, 3, 3]

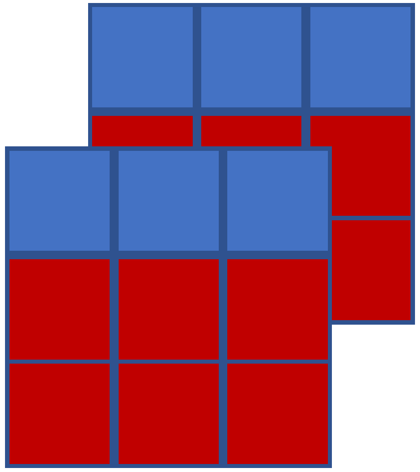
Begin: [0, 1, 0]

Size:

```
import tensorflow as tf
x = tf.constant([[[1., 2., 3.], [4., 5.,6 ], [7. , 8.,9 ]],
                 [[10., 11.,12], [13., 14., 15], [16., 17., 18]]])
print(x)


res = tf.slice(x, [0, 1, 0], [2, 1, 2])
print("\n")
print(res)
```

```
tf.Tensor(
[[[ 1.  2.  3.]
  [ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[10. 11. 12.]
  [13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 3, 3), dtype=float32)


tf.Tensor(
[[[ 4.  5.]]

 [[13. 14.]]], shape=(2, 1, 2), dtype=float32)
```

Extract a slice from a tensor

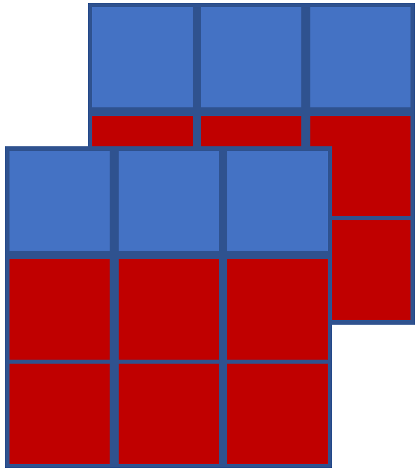


Shape: [2, 3, 3]

Begin: [0, 1, 0]

Size:

Extract a slice from a tensor



```
import tensorflow as tf
x = tf.constant([[[1., 2., 3.], [4., 5.,6 ], [7. , 8.,9 ]],
                  [[10., 11.,12], [13., 14., 15], [16., 17., 18]]])
print(x)


res = tf.slice(x, [0, 1, 0], [-1, -1, -1])
print("\n")
print(res)
```

```
tf.Tensor(
[[[ 1.  2.  3.]
  [ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[10. 11. 12.]
  [13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 3, 3), dtype=float32)


tf.Tensor(
[[[ 4.  5.  6.]
  [ 7.  8.  9.]]

 [[13. 14. 15.]
  [16. 17. 18.]]], shape=(2, 2, 3), dtype=float32)
```

Shape: [2, 3, 3]

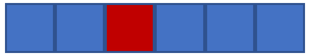
Begin: [0, 1, 0]

Size:

Extract non-contiguous slices from the first dimension

tf.gather(<params>,<indices>,<axis>)

- params: A tensor you want to extract values from.
- indices: A tensor specifying the indices pointing into params
- Axis: axis to apply the operation



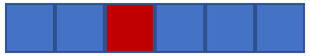
```
x = tf.constant([3, 5, 1, 6, 8, 7])
tf.gather(x, [2])
```

```
<tf.Tensor: shape=(1,), dtype=int32, numpy=array([1])>
```

Extract non-contiguous slices from the first dimension

tf.gather(<params>,<indices>,<axis>)

- params: A tensor you want to extract values from.
- indices: A tensor specifying the indices pointing into params
- Axis: axis to apply the operation



```
x = tf.constant([3, 5, 1, 6, 8, 7])
tf.gather(x, [2])
```

```
<tf.Tensor: shape=(1,), dtype=int32, numpy=array([1])>
```

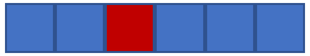
```
x = tf.constant([3, 5, 1, 6, 8, 7])
tf.gather(x, [0,3])
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([3, 6])>
```

Extract non-contiguous slices from the first dimension

tf.gather(<params>,<indices>,<axis>)

- params: A tensor you want to extract values from.
- indices: A tensor specifying the indices pointing into params
- Axis: axis to apply the operation



```
x = tf.constant([3, 5, 1, 6, 8, 7])
tf.gather(x, [2])
```

```
<tf.Tensor: shape=(1,), dtype=int32, numpy=array([1])>
```

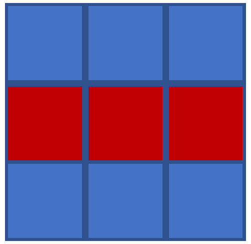
```
x = tf.constant([3, 5, 1, 6, 8, 7])
tf.gather(x, [0,3])
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([3, 6])>
```

```
tf.gather(x, [[2, 0], [2, 5]])
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=  array([[1, 3],
                                                      [1, 7]])>
```

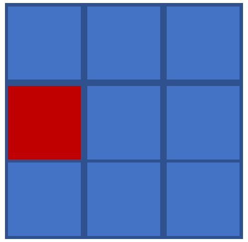
Extract non-contiguous slices from the first dimension



```
x = tf.constant([     [10.0, 11.0, 12.0],
                      [20.0, 21.0, 22.0],
                      [30.0, 31.0, 32.0]])
y = tf.gather(x, indices=[1])
print(y)
```

```
tf.Tensor([[20. 21. 22.]], shape=(1, 3), dtype=float32)
```

Extract non-contiguous slices from the first dimension



```
x = tf.constant([     [10.0, 11.0, 12.0],
                      [20.0, 21.0, 22.0],
                      [30.0, 31.0, 32.0]])
y = tf.gather(x, indices=[1])
z = tf.gather(y, indices=[0], axis=1)
print(z)
```

```
tf.Tensor([[20.]], shape=(1, 1), dtype=float32)
```

Maximum element of a Tensor

Maximum element

```
x = tf.constant([[9,2,10,4],[5,6,7,8]])
print(tf.reduce_max(x))
```

```
tf.Tensor(10, shape=(), dtype=int32)
```

[9, 2, 10, 4]
[5, 6, 7 , 8]

Index of the Maximum element

```
x = tf.constant([[9,2,10,4],[5,6,7,8]])
print(tf.math.argmax(x))
```

```
tf.Tensor([0 1 0 1], shape=(4,), dtype=int64)
```

[**9**, 2, **10**, 4]
[5, **6**, 7 , **8**]

```
x = tf.constant([[2, 20, 30, 3, 6], [3, 11, 16, 1, 8],
                 [14, 45, 23, 5, 27]])
print(tf.math.argmax(x))
```

```
tf.Tensor([2 2 0 2 2], shape=(5,), dtype=int64)
```

[2, 20, **30**, 3, 6]
[3, 11, 16, 1, 8]
[**14**, **45**, 23, **5**, **27**]

Minimum element of a Tensor

Minimum element

```
x = tf.constant([[9,2,10,4],[5,6,7,8]])
print(tf.reduce_min(x))
```

```
tf.Tensor(2, shape=(), dtype=int32)
```

[9, **2**, 10, 4]
[5, 6, 7 , 8]

Index of the Minimum element

```
x = tf.constant([[9,2,10,4],[5,6,7,8]])
print(tf.math.argmin(x))
```

```
tf.Tensor([1 0 1 0], shape=(4,), dtype=int64)
```

[9, 2, 10, 4]
[5, 6, 7 , 8]

```
x = tf.constant([[2, 20, 30, 3, 6], [3, 11, 16, 1, 8],
                 [14, 45, 23, 5, 27]])
print(x)
print(tf.math.argmin(x))
```

```
tf.Tensor(
[[ 2 20 30  3  6]
 [ 3 11 16  1  8]
 [14 45 23  5 27]], shape=(3, 5), dtype=int32)
tf.Tensor([0 1 1 1 0], shape=(5,), dtype=int64)
```

[2, 20, 30, 3, 6]
[3, 11, 16, 1, 8]
[14, 45, 23, 5, 27]

Minimum/Maximum of Two Tensors

Minimum

```
x = tf.constant([0., 0., 0., 0.])
y = tf.constant([-5., -2., 0., 3.])
tf.math.minimum(x, y)
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([-5., -2.,  0.,  0.], dtype=float32)>
```

Maximum

```
x = tf.constant([0., 0., 0., 0.])
y = tf.constant([-5., -2., 0., 3.])
tf.math.maximum(x, y)
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([0., 0., 0., 3.], dtype=float32)>
```

Concatenation of Two Tensors

Along 0-Axis

```
x = [[1, 2, 3], [4, 5, 6]]
y = [[7, 8, 9], [10, 11, 12]]
tf.concat([x, y], 0)
```

```
<tf.Tensor: shape=(4, 3), dtype=int32, numpy=
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])>
```

Along 1-Axis

```
x = [[1, 2, 3], [4, 5, 6]]
y = [[7, 8, 9], [10, 11, 12]]
tf.concat([x, y], 1)
```

```
<tf.Tensor: shape=(2, 6), dtype=int32, numpy=
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])>
```

Modifying the value of a Tensor

Not a simple operation.

Possible for Variable type. But, for such operation, we would prefer to use numpy library.

Matrix Multiplication

```
import tensorflow as tf

A1 = tf.constant([[1, 2, 3, 4]])
B1 = tf.constant([[3], [4], [5], [5]])
C1 = tf.multiply(A1, B1)
tf.print(C1)
```

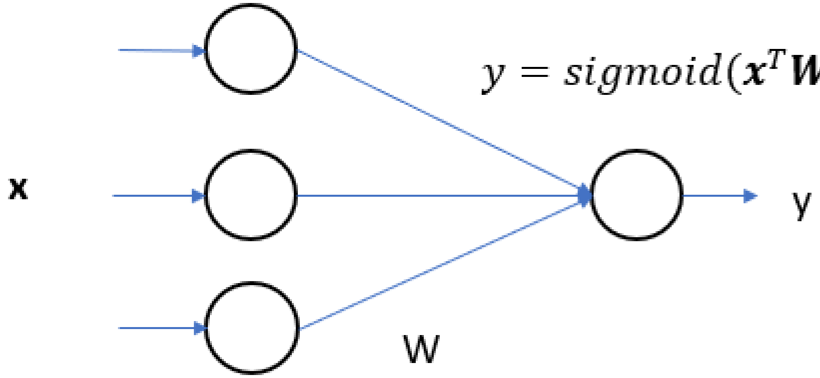
```
[[3 6 9 12]
 [4 8 12 16]
 [5 10 15 20]
 [5 10 15 20]]
```

```
import tensorflow as tf

A1 = tf.constant([[1, 2, 3, 4]])
B1 = tf.constant([[3], [4], [5], [5]])
C1 = tf.matmul(A1, B1)
tf.print(C1)
```

```
[[46]]
```

Lesson 28-29

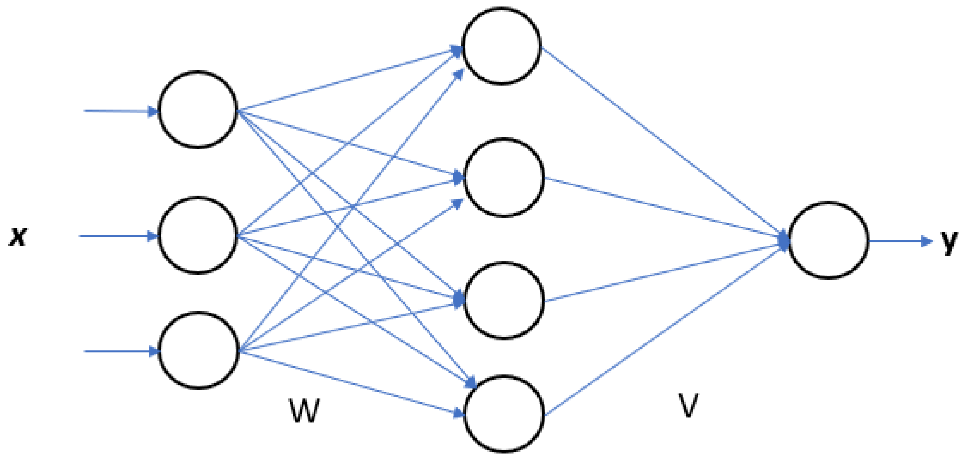
Tensorflow and Keras



$$y = sigmoid(\boldsymbol{x}^T\boldsymbol{W})$$

```python
import numpy as np

D = np.array([[4,500,6],
              [4,550,5.5],
              [2,200,3.5],
              [2,250,4]])
label = np.array([[1,1,0,0]]).T

np.random.seed(1)
w = np.random.random((3,1))

for iteration in range(10):
    iLayer = D
    p = np.dot(iLayer,w)       # Perceptron
    oLayer = 1/(1+np.exp(-p))  # Sigmoid(x)

    MSE = 2*np.square(np.subtract(oLayer,label)).mean() # Mean Square Error
    print(MSE)

    der = oLayer * (1-oLayer) # dirivatives of sigmoid
    grad = np.dot(iLayer.T, der *MSE)

    w += 0.01*grad
    print(w)

print(oLayer)
```

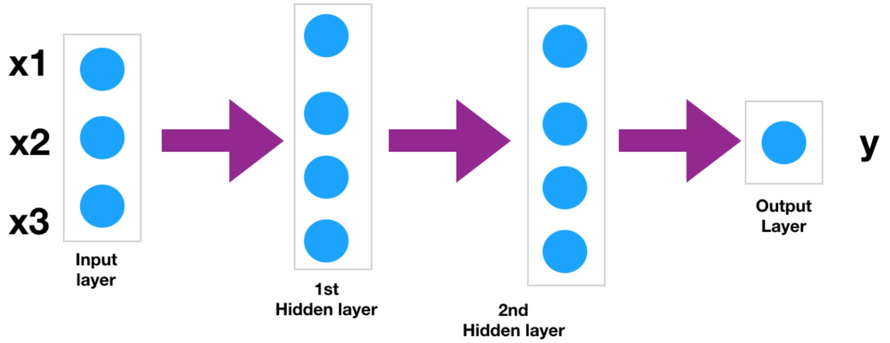
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np

D = np.array([[4,500,6],
              [4,550,5.5],
              [2,200,3.5],
              [2,250,4]])
label = np.array([[1,1,0,0]]).T


model = Sequential()
model.add(Dense(1, input_shape=(3,),activation='sigmoid'))
model.summary()

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(training, output, epochs=10, batch_size=250, verbose=1, validation_split=0.2)
```




```python
import numpy as np

D = np.array([[4,500,6],
              [4,550,5.5],
              [2,200,3.5],
              [2,250,4]])
label = np.array([[1,1,0,0]]).T

np.random.seed(1)
w = np.random.random((3,4))
v = np.random.random((4,1))

for iteration in range(10):
    iLayer = D
    hP = np.dot(iLayer,w)       # Perceptron
    hLayer = 1/(1+np.exp(-hP))  # Sigmoid(x)

    oP = np.dot(hLayer,v)       # Perceptron
    oLayer = 1/(1+np.exp(-oP))  # Sigmoid(x)

    MSE = 2*np.square(np.subtract(oLayer,label)).mean() # Mean Square Error
    print(MSE)

    oDer = oP * (1-oP) # dirivatives of sigmoid
    vGrad = np.dot(oLayer.T, oDer *MSE)
    v += 0.00000001*vGrad
    print(v)

    hDer = hP * (1-hP) # dirivatives of sigmoid
    wGrad = np.dot(iLayer.T, hDer *v*oDer*MSE)
    w += 0.00000001*wGrad
    print(w)

print(oLayer)
```

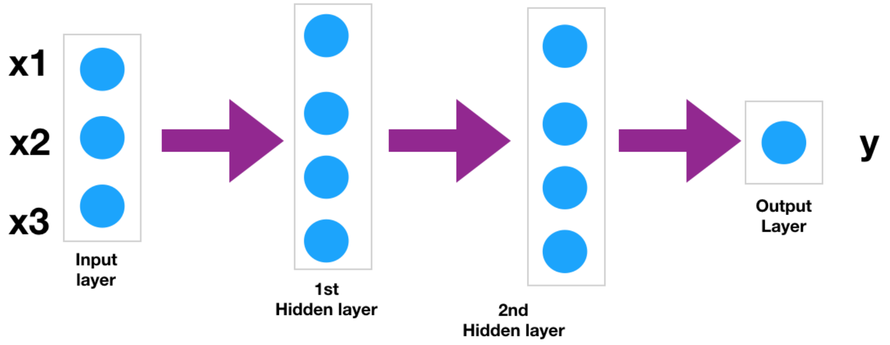
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np

D = np.array([[4,500,6],
              [4,550,5.5],
              [2,200,3.5],
              [2,250,4]])
label = np.array([[1,1,0,0]]).T


model = Sequential()
model.add(Dense(4, input_shape=(3,),activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(training, output, epochs=10, batch_size=250, verbose=1, validation_split=0.2)
```

Sequential Vs Functional Keras API

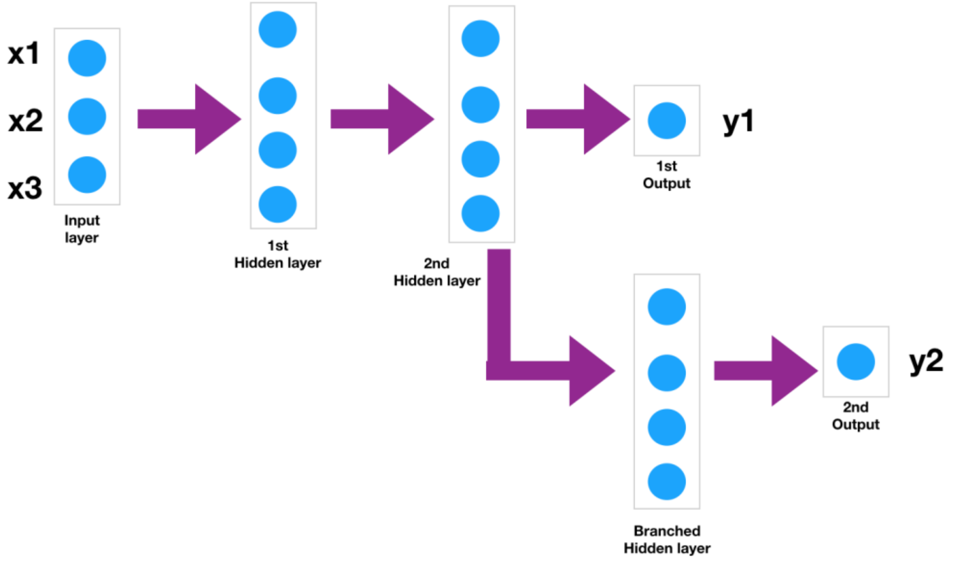


Sequential

Sequential Vs Functional Keras API

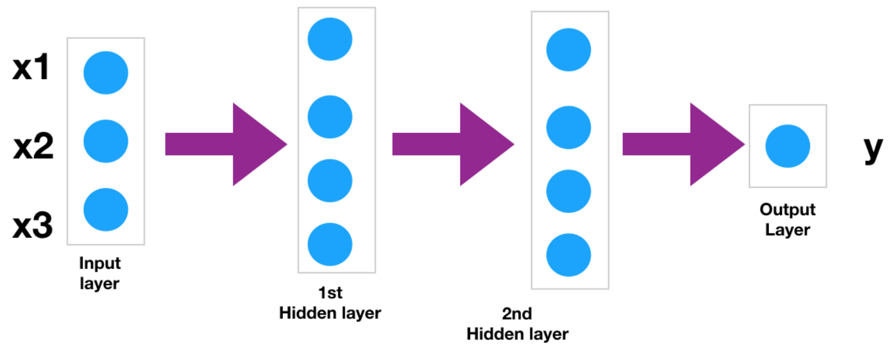


Sequential



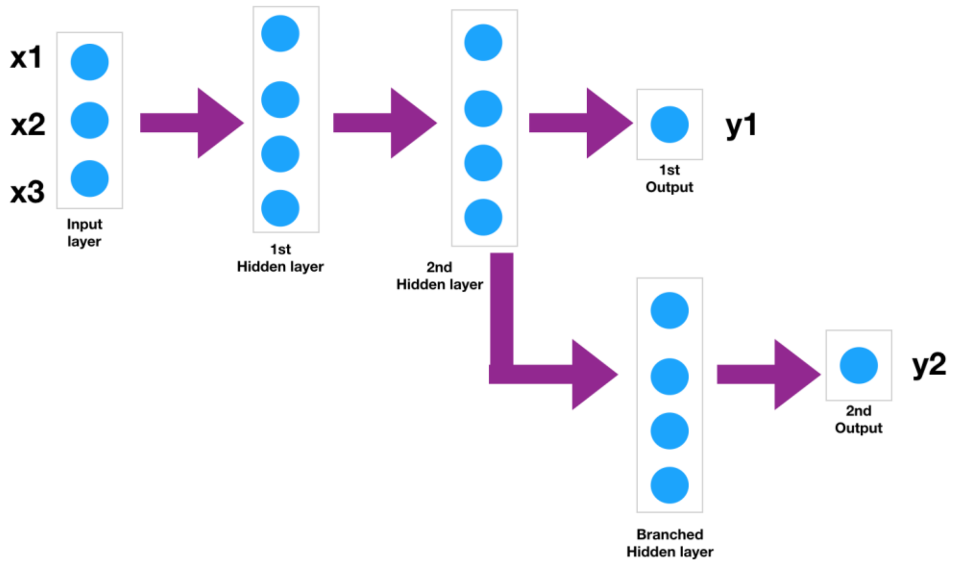
Functional

Functional Keras API



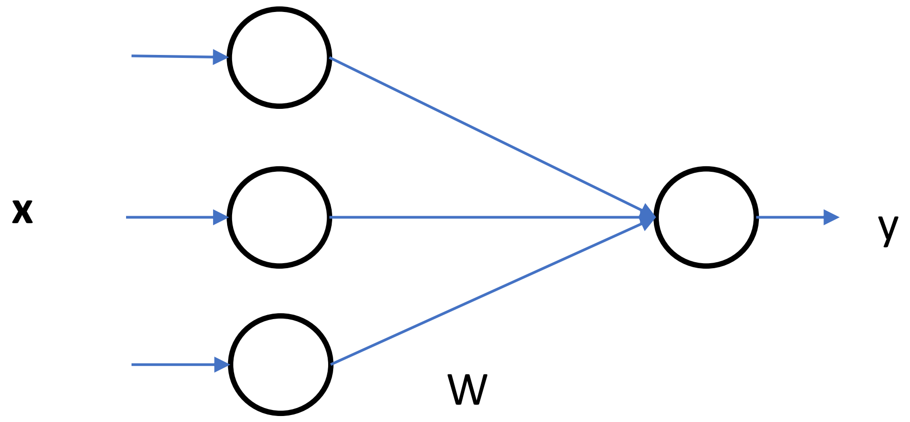

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,Dense

D = np.array([[4,500,6],
              [4,550,5.5],
              [2,200,3.5],
              [2,250,4]])
label = np.array([[1,1,0,0]]).T

## Creating the layers
input_layer = Input(shape=(3,))
layer_1 = Dense(4, activation="relu")(input_layer)
layer_2 = Dense(4, activation="relu")(layer_1)
o_layer = Dense(4, activation="relu")(layer_2)

##Defining the model by specifying the input and output layers
model = Model(inputs=input_layer, outputs=o_layer)
model.summary()

## defining the optimiser and loss function
model.compile(optimizer='adam', loss='mse')

## training the model
model.fit(D, label,epochs=2, batch_size=128,validation_data=(D,label))
```

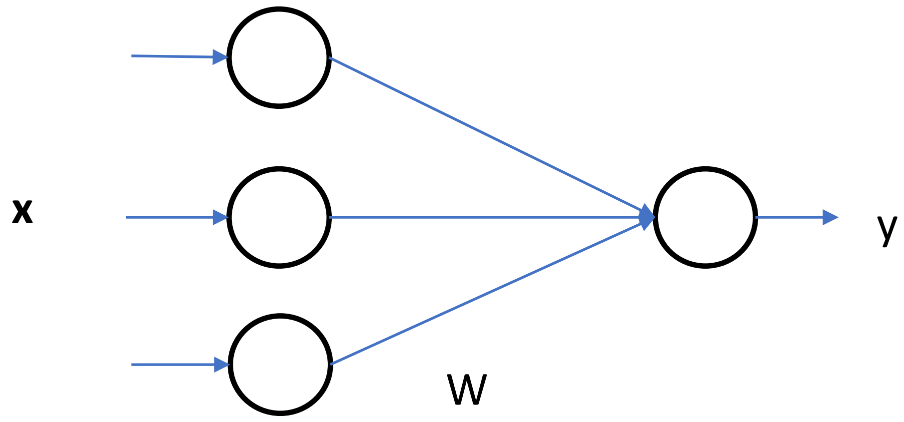
Functional Keras API




```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,Dense

D = np.array([[4,500,6],
              [4,550,5.5],
              [2,200,3.5],
              [2,250,4]])
label = np.array([[1,1,0,0]]).T

## Creating the layers
input_layer = Input(shape=(3,))
layer_1 = Dense(4, activation="relu")(input_layer)
layer_2 = Dense(4, activation="relu")(layer_1)
layer_3 = Dense(4, activation="relu")(layer_2)
o1_layer= Dense(1, activation="linear")(layer_2)
o2_layer= Dense(1, activation="linear")(layer_3)

##Defining the model by specifying the input and output layers
model = Model(inputs=input_layer, outputs=[o1_layer,o2_layer])
model.summary()

## defining the optimiser and loss function
model.compile(optimizer='adam', loss='mse')

## training the model
model.fit(D, label,epochs=2, batch_size=128,validation_data=(D,label))
```

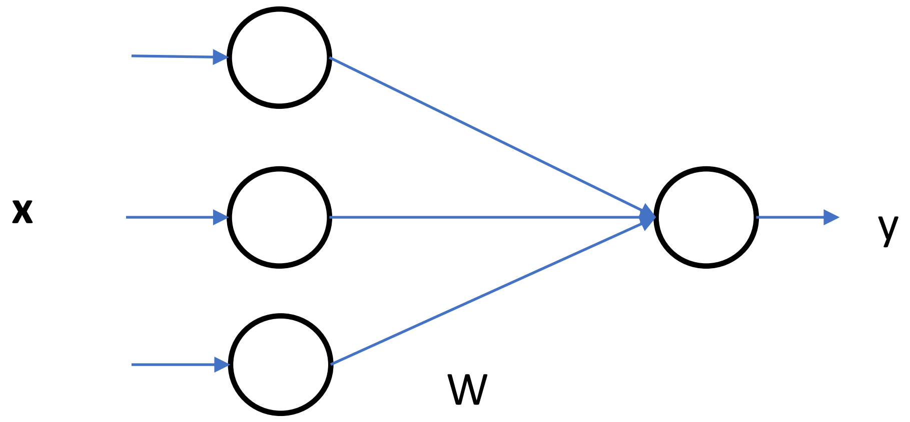
Lesson 25-27 : Implementing Neural Network in Python


x
W
y



$$perceptron = \boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{W}$$

$$y = sigmoid(perceptron)$$

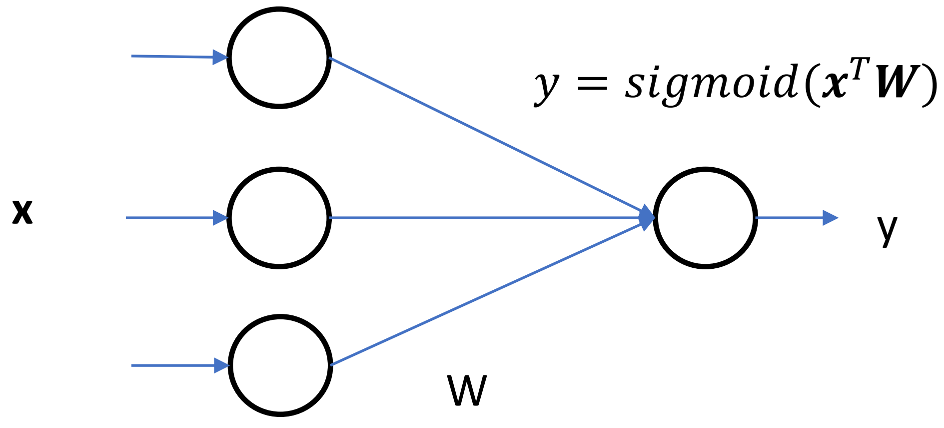


$$y = sigmoid(\boldsymbol{x}^T \boldsymbol{W})$$

$$perceptron = \boldsymbol{x}^T \boldsymbol{W}$$

$$y = sigmoid(perceptron)$$

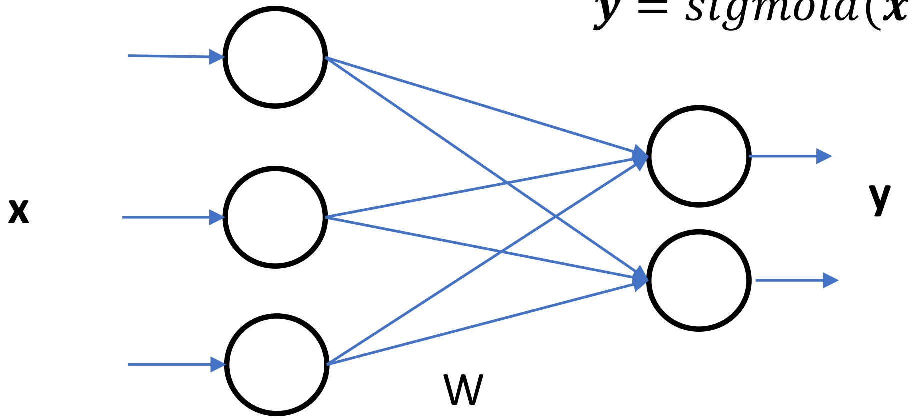
Feed forward pass




```
import numpy as np

D = np.array([[1,2,3],      # dataset
              [1,0,2],
              [0,1,4],
              [2,1,4]])

# Initialize weight matrix
np.random.seed(1)
w = np.random.random((3,1))
print("Weight Matrix : ")
print(w)

#Forward Pass
for iteration in range(1):
    iLayer = D
    oPer = np.dot(iLayer,w)                     # Perceptron
    oLayer = 1/(1+np.exp(-oPer))      # Sigmoid

print("Input :")
print(D)
print("Predicted Output: ")
print(oLayer)
```

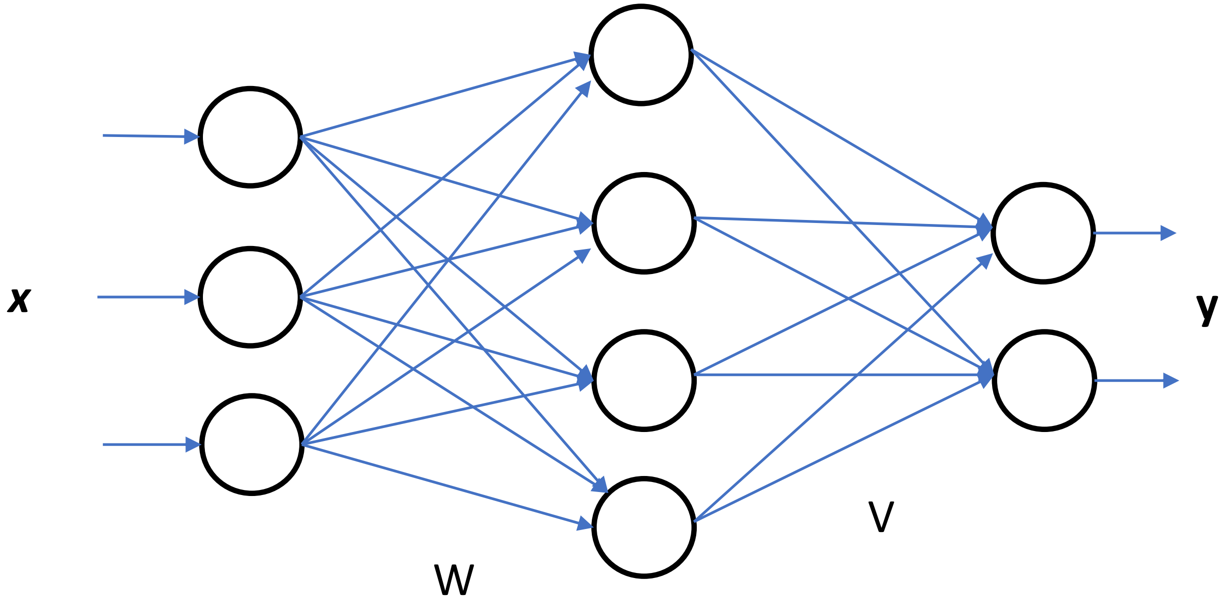
Feed forward pass

$$y = sigmoid(\boldsymbol{x}^T \boldsymbol{W})$$




```
import numpy as np

D = np.array([[1,2,3],      # dataset
              [1,0,2],
              [0,1,4],
              [2,1,4]])

# Initialize weight matrix
np.random.seed(1)
w = np.random.random((3,2))
print("Weight Matrix : ")
print(w)

#Forward Pass
for iteration in range(1):
    iLayer = D
    oPer = np.dot(iLayer,w)                  # Perceptron
    oLayer = 1/(1+np.exp(-oPer))       # Sigmoid

print("Input :")
print(D)
print("Predicted Output: ")
print(oLayer)
```

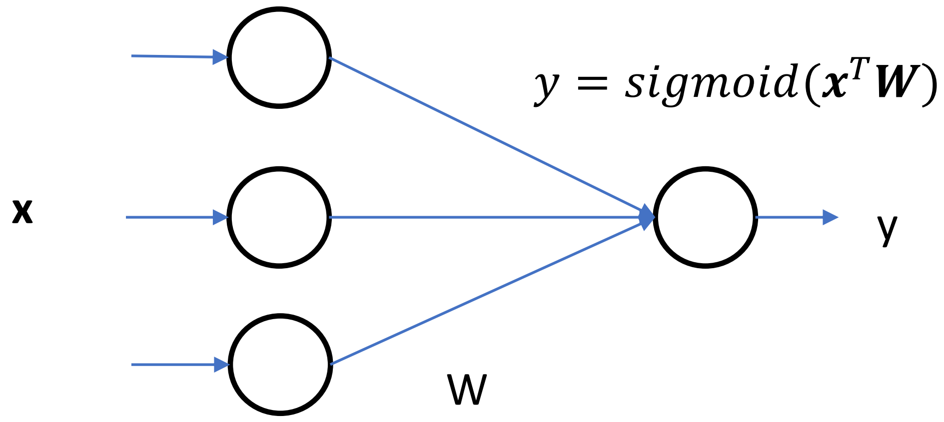
Feed forward pass



$$y = sigmoid(sigmoid(\boldsymbol{x}^T\boldsymbol{W})^{\boldsymbol{T}}V)$$

```python
import numpy as np

D = np.array([[1,2,3],
              [1,0,2],
              [0,1,4],
              [2,1,4]])

# Initialize Weight Matrices
np.random.seed(1)
W = np.random.random((3,4))
print("W weight matrix:")
print(W)

print("V weight matrix:")
V = np.random.random((4,2))
print(V)

for iteration in range(1):
    iLayer = D

    hP = np.dot(iLayer,W)        # Hidden Layer
    hLayer = 1/(1+np.exp(-hP))

    oP = np.dot(hLayer,V)        # Output Layer
    oLayer = 1/(1+np.exp(-oP))

print("Input :")
print(training)
print("Predicted Output: ")
print(oLayer)
```

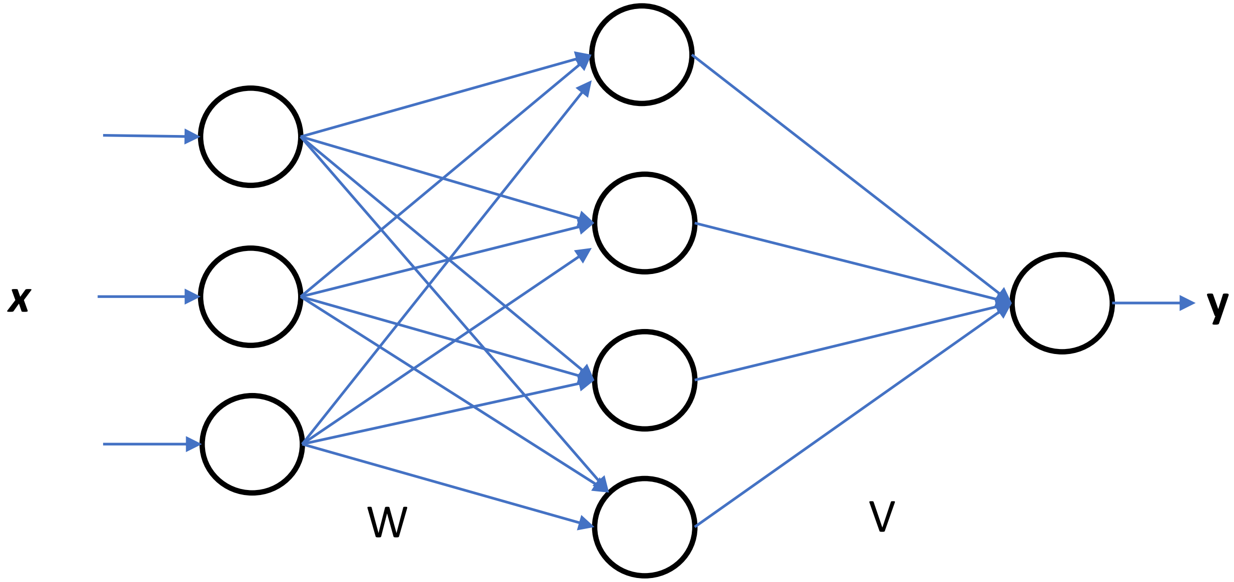
Backpropagation



$$\frac{\delta E}{\delta W_{11}} = \frac{\delta z_1}{\delta W_{11}} \times \frac{\delta y_1}{\delta z_1} \times \frac{\delta E}{\delta y_1}$$

$$\frac{\delta E}{\delta W_{11}} = x \times z(1-z) \times 2(\acute{y} - y)$$

```
import numpy as np

D = np.array([[4,500,6],
              [4,550,5.5],
              [2,200,3.5],
              [2,250,4]])
label = np.array([[1,1,0,0]]).T

np.random.seed(1)
w = np.random.random((3,1))

for iteration in range(10):
    iLayer = D
    p = np.dot(iLayer,w)       # Perceptron
    oLayer = 1/(1+np.exp(-p))  # Sigmoid(x)

    MSE = 2*np.square(np.subtract(oLayer,label)).mean() # Mean Square Error
    print(MSE)

    der = oLayer * (1-oLayer) # dirivatives of sigmoid
    grad = np.dot(iLayer.T, der *MSE)

    w += 0.01*grad
    print(w)

print(oLayer)
```



$$\frac{\delta E}{\delta V_{11}} = \frac{\delta z_1}{\delta V_{11}} \times \frac{\delta y_1}{\delta z_1} \times \frac{\delta E}{\delta y_1} = \frac{\delta E}{\delta V_{11}}$$
$$= x \times z(1-z) \times 2(\acute{y} - y)$$

$$\frac{\delta E}{\delta W_{11}} = \frac{\delta a_1}{\delta W_{11}} \times \frac{\delta h_1}{\delta a_1} \times \frac{\delta z_1}{\delta h_1} \times \frac{\delta y_1}{\delta z_1} \times \frac{\delta E}{\delta y_1}$$
$$= x \times a(1-a) \times V_{11} \times z(1-z) \times 2(\acute{y} - y)$$

```python
import numpy as np

D = np.array([[4,500,6],
              [4,550,5.5],
              [2,200,3.5],
              [2,250,4]])
label = np.array([[1,1,0,0]]).T

np.random.seed(1)
w = np.random.random((3,4))
v = np.random.random((4,1))

for iteration in range(10):
    iLayer = D
    hP = np.dot(iLayer,w)        # Perceptron
    hLayer = 1/(1+np.exp(-hP))   # Sigmoid(x)

    oP = np.dot(hLayer,v)        # Perceptron
    oLayer = 1/(1+np.exp(-oP))   # Sigmoid(x)

    MSE = 2*np.square(np.subtract(oLayer,label)).mean() # Mean Square Error
    print(MSE)

    oDer = oP * (1-oP) # dirivatives of sigmoid
    vGrad = np.dot(oLayer.T, oDer *MSE)
    v += 0.00000001*vGrad
    print(v)

    hDer = hP * (1-hP) # dirivatives of sigmoid
    wGrad = np.dot(iLayer.T, hDer *v*oDer*MSE)
    w += 0.00000001*wGrad
    print(w)

print(oLayer)
```