TensorFlow 2.0

Machine Learning

- Exploratory Analysis of data
- Cleaning data
- Feature Engineering
- Modeling
- Evaluation
- Application

Overview of TensorFlow components

Exploratory Analysis of data

(pandas, numpy)

Consume data

tf.data
tf.datasets

Model building

Primitive data structures and operations (tf.Tensor, tf.Variable, tf.Operation)

Higher building blocks (tf.Module tf.GradientTape, @function)

Main types of deep networks

- Fully connected
- Convolutional
- Recurrent
- Transformers

Keras sub-modules. (Sequential, Functional and Subclassing APIs)

Estimator API

Model training, evaluation, saving

Tensorboard

Save model (HDF5, SavedModel)

Model serving

TensorFlow serving

TensorFlow building-blocks

1- tf.Tensor

```
>>> import tensorflow as te
>>> tf.__version__
```

tf.Tensor is an n-dimensional data structure heavily used when defining machine learning models, as they are used to store inputs, intermediate outputs of layers and final outputs of the model. **tf.Tensor** is an immutable object.

```
>>> tf.constant(
        value,
        dtype= None,
        shape=None,
        name= 'Const'
        )
```

```
>>> a = tf.constant( 2, shape=[4], dtype=tf.float32 )
>>> b = tf.constant( 3, shape=[4], dtype=tf.float32 )
>>> c = tf.add(a, b)
>>> type( c ).__name__
>>> c[0].assign( 1 )
```

Create tensor

```
>>> tf.constant( 0, shape=(2,2) )
>>> tf.constant( [ 1, 2, 3, 4, 5, 6 ] )
>>> tf.constant( np.zeros( (3,3) )

>>> tf.zeros( shape)       tf.zeros_like( input )
>>> tf.ones( shape )       tf.ones_like( input )
>>> tf.eye( num_rows, num_columns=None )
>>> tf.fill(  dims, value )
>>> tf.linspace(  start, stop, num )
>>> tf.range(  start, limit, delta=1 )

>>> tf.random.normal( shape, mean=0.0, stddev=1.0 )
>>> tf.random.uniform( shape, minval=0, maxval=None )
```

Indexing

Like NumPy arrays [start : end : step]

```
>>> t[ :, 0],    t[ t < 2 ],    t[ 2 : 7 ].numpy( )        tf.gather( t,  indices=tf.constant([0,2]) )
```

Attributes and Methods

```
>>> t.device                >>> t.numpy(  )
>>> t.dtype
>>> t.name
>>> t.shape
>>> t.ndim
```

Common operations

```
>>> tf.cast( x, dtype )
>>> tf.reshape( tensor, shape )
>>> tf.transpose( a, perm=None )
>>> tf.where( condition, x=None, y=None )
>>> tf.squeeze( input, axis=None )  tf.expand_dims( input, axis=None )
>>> tf.sort( values, axis=-1, direction='ASCENDING')        tf.argsort( ... )
>>> tf.stack( values, axis=0)       tf.unstack( values, axis=0)
>>> tf.one_hot( indices, depth )
```

Math

```
>>> tf.add( x, y )          x + y            >>> tf.matmul( A, B )         A @ B
>>> tf.subtract( x, y )     x - y            >>> tf.linalg.matvec( A, x )
>>> tf.multiply( x, y )     x * y            >>> tf.norm( t , ord='euclidean')
>>> tf.divide( x, y )       x / y
```

```
>>> tf.abs( x )                                  >>> tf.math.floor( x )   tf.math.ceil( x )
>>> tf.math.sigmoid( x )                         >>> tf.math.argmax( input, axis=None  )
>>> tf.math.tanh( x )                            >>> tf.math.argmin( input, axis=None  )
>>> tf.math.cos( x )   tf.math.acos( x )         >>> tf.math.top_k( input, k=1,sorted=True )
>>> tf.math.exp( x )   tf.math.log( x )
>>> tf.math.pow( x, y )     x**2
```

```
>>> tf.reduce_sum( input_tensor, axis=None )     >>> tf.math.cumsum( x, axis=0 )
>>> tf.reduce_prod( input_tensor, axis=None )    >>> tf.math.cumprod( x, axis=0 )
>>> tf.reduce_min( input_tensor, axis=None )
>>> tf.reduce_max( input_tensor, axis=None )
```

2- tf.Variable

tf.Variable is ideal for defining model parameters and it can change the value of elements as required after it is initialzsed. **tf.Variable** is a mutable object.

Create a variable

```
>>> tf.Variable(
        initial_value= [ tensor, numpy, initializer ],
        trainable= True,
        dtype= None,
        name= None
    )
```

tf.keras.initializers

Constant(value)
Ones( ), Zeros( ), Identity( )
GlorotNormal( ), GlorotUniform( )
HeNormal( ), HeUniform( )
RandomNormal( ), RandomUniform( )

dtype

tf.float16, 32, 64
tf.uint8, 16, 32, 64
tf.int8,16, 32, 64
tf.bool

```
>>> v1 = tf.Variable( tf.constant(2.0, shape=[4]), dtype=tf.float32 )
>>> v2 = vtf.Variable( np.ones(shape=[4,3]), dtype='float32' )
>>> v3 = tf.Variable( tf.keras.initializers.RandomNormal( )( shape=[3,4,5] ) )
```

Attributes and Methods

```
>>> v.device          >>> v.numpy(  )
>>> v.dtype           >>> v.assign(  )
>>> v.name
>>> v.shape           >>> v = tf.Variable( np.zeros(shape=[4,3]  )
>>> v.trainable       >>> v = v[0,2].assign(1)
                      >>> v[2:, :2].assign( [[3,3],[3,3]] )
```

TensorFlow Concepts

```python
import tensorflow as tf
import matplotlib.pyplot as plt

# ------Generate data ------ #
TRUE_W = 3.0
TRUE_B = 2.0
NUM_EXAMPLES = 1000

x = tf.random.normal( shape=[NUM_EXAMPLES] )
noise = tf.random.normal( shape=[NUM_EXAMPLES] )
y = x * TRUE_W + TRUE_B + noise


# ------Define model ------ #
class MyModel( tf.Module ):
  def __init__(self, **kwargs):
    super().__init__( **kwargs )
    self.w = tf.Variable(5.0)
    self.b = tf.Variable(0.0)

  def __call__( self, x ):
      return self.w * x + self.b


# ------Define loss ------ #
@tf.function
def loss(target_y, predicted_y):
   return tf.reduce_mean(t f.square(target_y - predicted_y) )


# ------Train a model ------ #
model = MyModel()
learning_rate= 0.1
epochs = 25

for epoch in range(epochs):
   with tf.GradientTape( ) as tape:
      yhat = model( x )
      current_loss = loss(y, yhat)

   dw, db = tape.gradient( current_loss, [model.w, model.b] )
   model.w.assign_sub( learning_rate * dw )
   model.b.assign_sub( learning_rate * db )

   print( "Epoch %2d: loss=%2.5f" % (epoch, current_loss) )

plt.scatter( x, y, c="b" )
plt.plot(x, model(x), c="r")
plt.show( )
```

Tensors, tf.Tensor, are multi-dimensional arrays with a uniform type.A tf.Variable represents a tensor whose value can be changed by running ops on it.

Automatic differentiation is useful for implementing machine learning algorithms such as backpropagation for training neural networks.
TensorFlow provides the tf.GradientTape API for automatic differentiation. TensorFlow "records" relevant operations executed inside the context of a tf.GradientTape onto a "tape". TensorFlow then uses that tape to compute the gradients of a "recorded" computation using reverse mode differentiation.

Graphs are data structures that contain a set of tf.Operation objects, which represent units of computation; and tf.Tensor objects, which represent the units of data that flow between operations. They are defined in a tf.Graph context; graphs are extremely useful and let your TensorFlow run fast, run in parallel, and run efficiently on multiple devices.

You create and run a graph in TensorFlow by using tf.function, either as a direct call or as a decorator. tf.function takes a regular function as input and returns a **Function**. A Function is a Python callable that builds TensorFlow graphs from the Python function. You use a Function in the same way as its Python equivalent.

While TensorFlow operations are easily captured by a tf.Graph, Python-specific logic needs to undergo an extra step in order to become part of the graph. tf.function uses a library called **AutoGraph** (tf.autograph) to convert Python code into graph-generating code.

Most **models** are made of layers. Layers are functions with a known mathematical structure that can be reused and have trainable variables. In TensorFlow, most high-level implementations of **layers** and **models**, such as Keras, are built on the same foundational class: tf.Module. By **subclassing** tf.Module, any tf.Variable or tf.Module instances assigned to this object's properties are **automatically** and **recursively** collected.

tf.keras

1- tf.keras

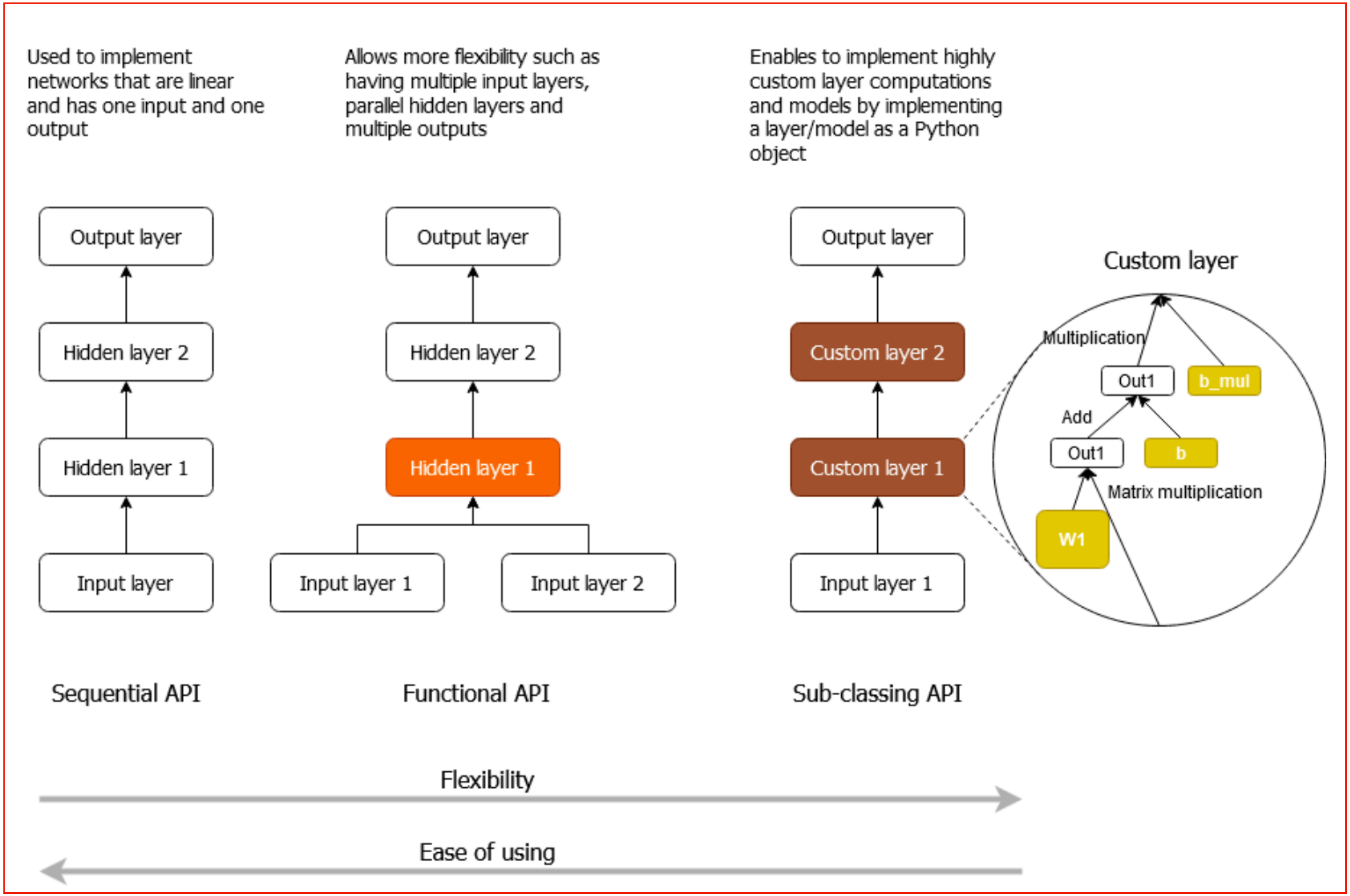


Image taken from "TensorFlow 2.0 in Action "

Keras: standard training

```python
import tensorflow as tf

# ------Define model ------ #
model = tf.keras.models.Sequential( )

model.add( tf.keras.layers.Flatten( input_shape=x_train[0].shape) )

model.add(  tf.keras.layers.Dense( 128, activation='tanh') )
model.add(  tf.keras.layers.Dense( 128, activation='tanh') )
model.add( tf.keras.layers.Dropout( 0.1) )
model.add( tf.keras.layers.Dense(10, activation='softmax') )


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

# ------Fit  model ------ #
result = model.fit(
    x_train,
    y_train,
    validation_data=(x_test, y_test),
    epochs=10)
```

Model: Attributes and Methods

```
>>> model.inputs
>>> model.outputs
>>> model.trainable_variables
>>> model.layers

>>> model.get_layer(name)
```

Layer: Attributes and Methods

```
>>> layer.weights
>>> layer.input
>>> layer.input_shape
>>> layer.output
>>> layer.output_shape
>>> layer.trainable
```

```python
feature_extractor = tf.keras.Model(
    inputs = initial_model.inputs,
    outputs = [ layer.output for layer in initial_model.layers ]
)

features = feature_extractor( x )
```

Linear Layers

Linear Layers

```
# Dense implements the operation: output = activation( dot( input, kernel ) + bias )
>>> tf.keras.layers.Dense( units, activation=None, [ input_shape,  name ] )
```

Dropout Layers

```
# The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time
>>> tf.keras.layers.Dropout( rate, [ input_shape,  name ] )
```

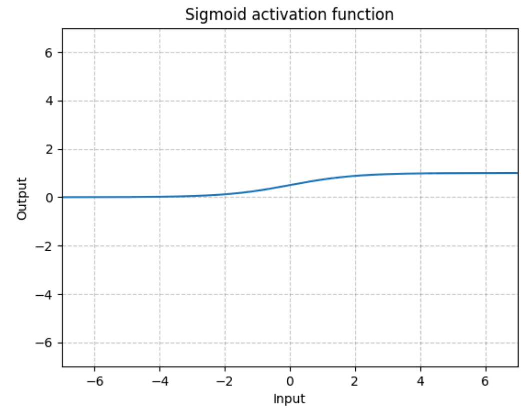
Utility Functions

```
# Input( ) is used to instantiate a Keras tensor.
>>> tf.keras.Input( shape )

# Flattens the input.
>>> tf.keras.layers.Flatten(  [ input_shape,  name ] )

# Layer that reshapes inputs into the given shape
>>> tf.keras.layers.Reshape( target_shape,  [ input_shape, name ] )

# Layer that concatenates a list of inputs.
>>> tf.keras.layers.Concatenate( axis=-1  [input_shape,  name ] )
```

Activations

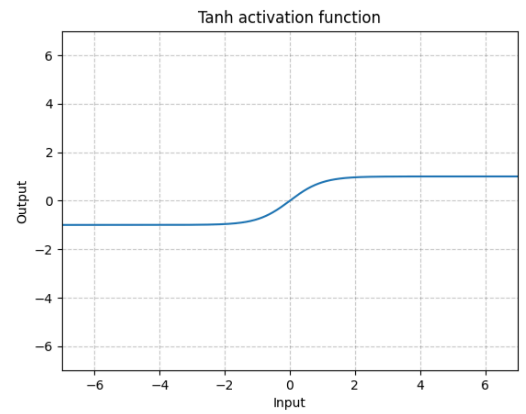
```
# [0,1]
>>> tf.keras.activations.sigmoid( )
```

$$\mathrm{Sigmoid}(x) = \sigma(x) = \frac{1}{1+\exp(-x)}$$



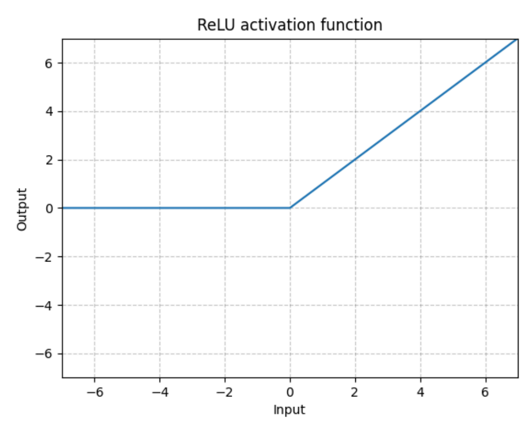

```
# [-1, 1]
>>> tf.keras.activations.tanh( )
```

$$\mathrm{Tanh}(x) = \tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$




```
# [0, +inf]
>>> tf.keras.activations.relu( )
```

$$\mathrm{ReLU}(x) = (x)^{+} = \max(0, x)$$




```
# [0, 1]
>>> tf.keras.activations.softmax( )
```

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

Losses

Regression

```
# mean of squares of errors between labels and predictions.
# model.compile( optimizer='sgd', loss=tf.keras.losses.MeanSquaredError( ) )
>>> tf.keras.losses.MeanSquaredError( )

# mean of absolute difference between labels and predictions
# model.compile( optimizer='sgd', loss=tf.keras.losses.MeanAbsoluteError( ) )
>>> tf.keras.losses.MeanAbsoluteError( )
```

Classification

```
# Use this cross-entropy loss for binary (0 or 1) classification applications
# model.compile( loss=tf.keras.losses.BinaryCrossentropy(from_logits=True) )
>>> tf.keras.losses.BinaryCrossentropy( from_logits=False )

# Use this crossentropy loss function when there are two or more label classes.
# We expect labels to be provided in a one_hot representation.
# model.compile( loss=tf.keras.losses.CategoricalCrossentropy( ) )
>>> tf.keras.losses.CategoricalCrossentropy( from_logits=False )

# Use this crossentropy loss function when there are two or more label classes.
#We expect labels to be provided as integers.
# model.compile( loss=tf.keras.losses.SparseCategoricalCrossentropy( ) )
>>> tf.keras.losses.SparseCategoricalCrossentropy( from_logits=False )
```

Optimization Algorithms

```
# Gradient descent (with momentum) optimizer
>>> tf.keras.optimizers.SGD( learning_rate=0.01,
                             momentum=0.0,
                             nesterov=False )

# Optimizer that implements the Adam algorithm
>>> tf.keras.optimizers.Adam( learning_rate=0.001,
                              beta_1=0.9,
                              beta_2=0.999
                              eps=1e-08,
                              amsgrad=False )
```