COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

Inteligencia en el Negocio

Tensorflow and Keras

comillas.edu

1

Tensorflow

Tensorflow

Tensorflow is an open source library for deep learning developed by Google and created in 2015.

The library have APIs for several languages such as Python.

Tensorfow

- To install tensorflow:

```
pip install tensorflow
```

- Currently, the last version is tensorflow 2.7.

Tensor

- All operations in tensorflow are based on tensors. A tensor is a n-dimensional matrix.
- The concept is very similar to numpy's ndarray objects but optimized to work in deep learning.

```
[1]: import tensorflow as tf

[5]: a = tf.convert_to_tensor([1, 2, 3])
     a

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

[6]: a.shape

[6]: TensorShape([3])

[7]: 2 * a

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

[8]: b= tf.convert_to_tensor([5, 6, 7])
     a + b

[8]: <tf.Tensor: shape=(3,), dtype=int32, numpy=array([ 6,  8, 10], dtype=int32)>

[ ]:
```

2

keras

tf.keras

Developing complex arquitectures in tensorflow and implementing training algorithms is very expensive.

There is a high level library included in tensorflow which makes the development much more simple. This library is called Keras.

tf.keras

There are three different ways to develop a model in keras:

- **Sequential Keras:** very simple, but cannot implement some types of architectures
- **Functional Keras**: based on functional programming (Recommended).
- **Object Based Keras**: based on object orienting programming.

Layers

Models in Keras are seen as a stack of layers. Keras includes several architectures of layers which inherit from tf.keras.Layer.

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(
        initial_value=w_init(shape=(input_shape[-1], self.units),
                             dtype='float32'),
        trainable=True)
    b_init = tf.zeros_initializer()
    self.b = tf.Variable(
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b

# Instantiates the layer.
linear_layer = SimpleDense(4)

# This will also call `build(input_shape)` and create the weights.
y = linear_layer(tf.ones((2, 2)))
assert len(linear_layer.weights) == 2

# These weights are trainable, so they're listed in `trainable_weights`:
assert len(linear_layer.trainable_weights) == 2
```

Fully connected layer

A fully connected perceptron is called a **Dense layer**.

```
[20]: layer_fully_connected = tf.keras.layers.Dense(units=20, input_shape=(3,),  activation="tanh", name="fully_connected_network")
```

We can pass a tensor through a layer:

```
•[24]: input_data = tf.convert_to_tensor([[1, 2, 3], [5, 6, 7]])

       layer_fully_connected(input_data)

 [24]: <tf.Tensor: shape=(2, 20), dtype=float32, numpy=
       array([[-0.89970976,  0.85315806,  0.595504  , -0.8262282 ,  0.9027754 ,
               -0.9084048 , -0.934704  , -0.59040296,  0.82051814,  0.5234247 ,
               -0.82376695, -0.7923588 ,  0.7504015 ,  0.05584188, -0.7327128 ,
                0.9288546 , -0.62394017, -0.9804115 , -0.42038485, -0.97999233],
              [-0.98888576,  0.9988982 ,  0.96401596, -0.99916345,  0.99382216,
               -0.99344915, -0.9994847 , -0.90813035,  0.99533963,  0.90798295,
               -0.9975692 , -0.9918107 ,  0.99585956,  0.45807612, -0.99462485,
                0.9999618 , -0.93172824, -0.99999833, -0.9227754 , -0.99999475]],
             dtype=float32)>
```

We can access (and change) the weights:

```
[23]: layer_fully_connected.weights

[23]: [<tf.Variable 'fully_connected_network/kernel:0' shape=(3, 20) dtype=float32, numpy=
       array([[ 0.48618072,  0.40848303, -0.03599849, -0.23658735, -0.31678694,
                0.48030967, -0.00122738, -0.04940742, -0.1096485 ,  0.26866156,
               -0.1886411 , -0.24742737,  0.22776657,  0.20087636, -0.09462219,
                0.3630839 , -0.04067105, -0.38047093,  0.02661341, -0.16028646],
              [-0.34380892, -0.22159037,  0.37110794, -0.3856871 ,  0.19758958,
               -0.4478202 , -0.13130856,  0.14824659,  0.46439683, -0.4172165 ,
               -0.0947727 ,  0.3206591 ,  0.15694642, -0.12842241, -0.39394102,
                0.460954  ,  0.10784394, -0.43484843, -0.47450083, -0.48899806],
              [-0.42308575,  0.4341141 , -0.00668865, -0.05605698,  0.46954322,
               -0.36766392, -0.47684836, -0.3084567 ,  0.11308521,  0.38227224,
               -0.26341313, -0.49054298,  0.14407134,  0.0372895 , -0.01735184,
                0.12166202, -0.30214933, -0.35262945,  0.15807629, -0.38636222]],
             dtype=float32)>,
       <tf.Variable 'fully_connected_network/bias:0' shape=(20,) dtype=float32, numpy=
       array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0.], dtype=float32)>]
```

Embedding

Keras has a special layer for dealing with categorical variables: **the embedding**. An embedding maps each different type of value to a fix length vector.

```
•[25]: embedding = tf.keras.layers.Embedding(input_dim=10, output_dim=4, name="embedding")

[29]: input_data = tf.convert_to_tensor([1, 3, 1])
      embedding(input_data)

[29]: <tf.Tensor: shape=(3, 4), dtype=float32, numpy=
      array([[ 0.01829534,  0.00556229, -0.01420671,  0.01422423],
             [ 0.00400547,  0.04107917,  0.0206295 ,  0.04760632],
             [ 0.01829534,  0.00556229, -0.01420671,  0.01422423]],
            dtype=float32)>
```

This **avoids having to compute the one hot vectors**, which consumes a lot of resources.

Merge layer

There are several types of layers which expect two or more tensors and return a unique one. These are called **merge layers**.

```
[39]: a = tf.convert_to_tensor([[1, 2, 3]])
      b = tf.convert_to_tensor([[5, 6, 7]])

[40]: tf.keras.layers.Concatenate(name="concat")([a, b])

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

[41]: tf.keras.layers.Add(name="add")([a, b])

[41]: <tf.Tensor: shape=(1, 3), dtype=int32, numpy=array([[ 6,  8, 10]], dtype=int32)>

[42]: tf.keras.layers.Subtract(name="subtract")([a, b])

[42]: <tf.Tensor: shape=(1, 3), dtype=int32, numpy=array([[-4, -4, -4]], dtype=int32)>

[47]: tf.keras.layers.Maximum(name="max")([a, b])

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

Sequential Model

A **sequential model** can be created adding the different layers one by one.

```
[61]: model = tf.keras.models.Sequential(name="sequential_model")
      model.add(tf.keras.layers.Dense(10, input_shape=(50, ), activation="relu", name="hidden_layer_1"))
      model.add(tf.keras.layers.Dense(5, activation="relu", name="hidden_layer_2"))
      model.add(tf.keras.layers.Dense(1, activation="sigmoid", name="final_layer"))
```

```
[62]: model.summary()
```

```
Model: "sequential_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_layer_1 (Dense)      (None, 10)                510

 hidden_layer_2 (Dense)      (None, 5)                 55

 final_layer (Dense)         (None, 1)                 6

=================================================================
Total params: 571
Trainable params: 571
Non-trainable params: 0
_________________________________________________________________
```

Functional Model

In order to use the functional model, we need to use the Input layer. This **returns a symbolic tensor**, that can be seen as a placeholder for storing a tensor but has not been filled yet.

```
[55]: input_layer = tf.keras.layers.Input(shape=(50, ), name="input")
      input_layer

[55]: <KerasTensor: shape=(None, 50) dtype=float32 (created by layer 'input')>
```

Then we create the flow as if it were a normal tensor.

```
[59]: x = tf.keras.layers.Dense(10, name="hidden_layer_1")(input_layer)
      x = tf.keras.layers.Dense(10, name="hidden_layer_2")(x)
      x = tf.keras.layers.Dense(10, name="final_layer")(x)
      model = tf.keras.models.Model(input_layer, x, name="functional_model")

[60]: model.summary()

      Model: "functional_model"
      _________________________________________________________________
       Layer (type)                Output Shape              Param #
      =================================================================
       input (InputLayer)          [(None, 50)]              0

       hidden_layer_1 (Dense)      (None, 10)                510

       hidden_layer_2 (Dense)      (None, 10)                110

       final_layer (Dense)         (None, 10)                110

      =================================================================
      Total params: 730
      Trainable params: 730
      Non-trainable params: 0
      _________________________________________________________________
```

Functional Model

Using functional models, we can create models with several inputs.

```
[4]: input_numerical = tf.keras.layers.Input(shape=(1, ), name="numerical_input")
     input_categorical = tf.keras.layers.Input(shape=(1, ), name="categorical_input")

     x_numeric = tf.keras.layers.Dense(10, activation="tanh", name="encoding_numerical")(input_numerical)
     x_categorical = tf.keras.layers.Embedding(input_dim=5, output_dim=3, name="embedding_categorical")(input_categorical)

     x_categorical = tf.keras.layers.Reshape(target_shape=(3, ), name="flat_vector")(x_categorical)
     x = tf.keras.layers.Concatenate()([x_numeric, x_categorical])

     x = tf.keras.layers.Dense(10, activation="tanh")(x)
     x = tf.keras.layers.Dense(1)(x)
     model = tf.keras.models.Model([input_numerical, input_categorical], x, name="model_with_two_inputs")
```

```
[5]: model.summary(line_length=150)
```

```
Model: "model_with_two_inputs"
______________________________________________________________________________________________________________________________________________________
 Layer (type)                                Output Shape                    Param #          Connected to
======================================================================================================================================================
 categorical_input (InputLayer)             [(None, 1)]                      0                []

 numerical_input (InputLayer)               [(None, 1)]                      0                []

 embedding_categorical (Embedding)          (None, 1, 3)                     15               ['categorical_input[0][0]']

 encoding_numerical (Dense)                 (None, 10)                       20               ['numerical_input[0][0]']

 flat_vector (Reshape)                      (None, 3)                        0                ['embedding_categorical[0][0]']

 concatenate_1 (Concatenate)                (None, 13)                       0                ['encoding_numerical[0][0]',
                                                                                               'flat_vector[0][0]']

 dense (Dense)                              (None, 10)                       140              ['concatenate_1[0][0]']

 dense_1 (Dense)                            (None, 1)                        11               ['dense[0][0]']

======================================================================================================================================================
Total params: 186
Trainable params: 186
Non-trainable params: 0
______________________________________________________________________________________________________________________________________________________
```

Model compile

Before training, a model must be compiled using model.compile method.

When compiling, we must pass:

- **The algorithm** we want to use to fit the model (for example, ‘adam’).
- **The loss function** (“mse”, “binary_crossentropy” …)
- **Metrics we want to compute during training process** (such as accuracy in classification models, mean absolute error in regression …)

```
model.compile("adam", "binary_crossentropy", metrics=["accuracy", tf.keras.metrics.AUC(name="area_under_curve")])
```

Model fitting

Once compiled, the model can be fitted using model.fit .

```
model.fit(X_train, y_train)
```

If the model has several inputs, we must pass them as a tuple:

```
model.fit((X_train_numerical, X_train_categorical), y_train)
```

Or inside a dictionary:

```
model.fit({"numerical_input": X_train_numerical, "categorical_input": X_train_numerical}, y_train)
```

Data can be a tf tensor or a numpy’s ndarray.

Model fitting

Other useful parameters:

- **epochs**: number of epochs to train
- **batch_size**: size of the batch used during training

In order to use validation data, we have two ways:

- **validation_split**: a random selection of the training data is chosen and used to validate.

```
model.fit(X_train, y_train, validation_split=0.2)
```

- **validation_data**: the validation data is directly passed

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

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

comillas.edu