**How would you describe TensorFlow in a short sentence? What are its main features? Can you name other popular Deep Learning libraries?**

Ans. TensorFlow is an open-source library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning. Its core is similar to NumPy, but it also features GPU support, support for distributed computing, computation graph analysis and optimization capabilities (with a portable graph format that allows you to train a TensorFlow model in one environment and run it in another), an optimization API based on reverse-mode autodiff, and several powerful APIs such as tf.keras, tf.data, tf.image, tf.signal, and more. Other popular Deep Learning libraries include PyTorch, MXNet, Microsoft Cognitive Toolkit, Theano, Caffe2, and Chainer.

**Is TensorFlow a drop-in replacement for NumPy? What are the main differences between the two?**

Ans. Although TensorFlow offers most of the functionalities provided by NumPy, it is not a drop-in replacement, for a few reasons. 1. the names of the functions are not always the same (for example, tf.reduce\_sum() versus np.sum() ). 2.some functions do not behave in exactly the same way (for example, tf.transpose() creates a transposed copy of a tensor, while NumPy's T attribute creates a transposed view, without actually copying any data). Lastly, NumPy arrays aremutable (variable), while TensorFlow tensors are not (but you can use a tf.Variable if you need a mutable object).numpy T, transpose() function, swapaxes() function

**Do you get the same result with tf.range(10) and tf.constant(np.arange(10))?**

Ans. Both tf.range(10) and tf.constant(np.arange(10)) return a one-dimensional tensor containing the integers 0 to 9. However, the former uses 32-bit integers while the latter uses 64-bit integers. Indeed, TensorFlow defaults to 32 bits, while NumPy defaults to 64 bits.

**Can you name six other data structures available in TensorFlow, beyond regular tensors?**

Ans. **Beyond regular tensors,**TensorFlow offers several other data structures, including [**sparse tensors,**](https://wiki.jikexueyuan.com/project/tensorflow-zh/api_docs/python/sparse_ops.html)**tensor arrays, ragged tensors, queues, string tensors, and sets**. **The last two are actually represented as regular tensors, but TensorFlow provides special functions to manipulate them (in tf.strings and tf.sets ).**

Note:

**Tensorflow ragged tensors**

Ragged tensors are irregular shapes provided by tf, or tensors with variable element lengths. such as:

digits = tf.ragged.constant([                             [3, 1, 4, 1],   
                             [],   
                             [5, 9, 2],   
                             [6],   
                             []])

or

words = tf.ragged.constant([                                            ["So", "long"],   
                                     ["thanks", "for", "all", "the", "fish"]])

Supported operations such as: tf.add, tf.concat, tf.tile, tf.string.substr

Points to note: 1 Different types cannot be stored, such as  
tf.ragged.constant([["one", "two"], [3, 4]])  
2 Different nested  
depths cannot be stored, such as tf.ragged. constant([“A”, [“B”, “C”]]) The  
correct way to write it should be:  
tf.ragged.constant([[“A”], [“B”, “C”]])

**A custom loss function can be defined by writing a function or by subclassing the keras.losses.Loss class. When would you use each option?**

Ans. When you want to define a custom loss function, in general you can just implement it as a regular Python function. However, if your custom loss function must support some hyperparameters (or any other state), then you should subclass thekeras.losses.Loss class and implement the \_\_init\_\_() and call() methods. If you want the loss function's hyperparameters to be saved along with the model, then you must also implement the get\_config() method.

**Similarly, a custom metric can be defined in a function or a subclass of keras.metrics.Metric. When would you use each option?**

Ans. Much like custom loss functions, most metrics can be defined as regular Python functions. But if you want your custom metric to support some hyperparameters (or any other state), then you should subclass the keras.metrics.Metric class. Moreover, if computing the metric over a whole epoch is not equivalent to computing the mean metric over all batches in that epoch (e.g., as for the precision and recall metrics), then you should subclass the keras.metrics.Metric class and implement the \_\_init\_\_() update\_state() , and result() methods to keep track of a running metric during each epoch. You should also implement thereset\_states() method unless all it needs to do is reset all variables to 0.0. If you want the state to be saved along with the model, then you should implement the get\_config() method as well.

**When should you create a custom layer versus a custom model?**

Ans. You should distinguish the internal components of your model (i.e., layers or reusable blocks of layers) from the model itself (i.e., the object you will train).The former should subclass the keras.layers.Layer class, while the latter should subclass the keras.models.Model class.

**What are some use cases that require writing your own custom training loop?**

Ans. Writing your own custom training loop is fairly advanced, so you should only do it if you really need to. Keras provides several tools to customize training without having to write a custom training loop: callbacks, custom regularizers, custom constraints, custom losses, and so on. You should use these instead of writing a custom training loop whenever possible: writing a custom training loop is more error-prone, and it will be harder to reuse the custom code you write. However, in some cases writing a custom training loop is necessary—for example, if you want to use different optimizers for different parts of your neural network, like in the Wide & Deep paper. A custom training loop can also be useful when debugging, or when trying to understand exactly how training works.

**Can custom Keras components contain arbitrary Python code, or must they be convertible to TF Functions?**

Ans. Custom Keras Components Should BE Convertible to TF Functions, Which means They Should Stick to TF the Operations AS much AS Possible and Respect All at The rules listed in " TF Function Rules" ON Page 409. The If you. Absolutely need to the include arbitrary Python code in A Custom component, you can either wrap it in a tf.py\_function() operation (but this will reduce performance and limit your model's portability ) or set dynamic=True when creating the custom layer or model (or set run\_eagerly=True when calling the model's compile () method).

**What are the main rules to respect if you want a function to be convertible to a TF Function?**

Ans.

**When would you need to create a dynamic Keras model? How do you do that? Why not make all your models dynamic?**

Ans. Creating a dynamic Keras model can be useful for debugging, as it will not compile any custom component to a TF Function, and you can use any Python debugger to debug your code. It can also be useful if you want to include arbitrary Python code in your model (or in your training code), including calls to external libraries. To make a model dynamic, you must set dynamic=True when creating it. Alternatively, you can set run\_eagerly=True when calling the model's compile() method. Making a model dynamic prevents Keras from using any of TensorFlow's graph features, so it will slow down training and inference, andyou will not have the possibility to export the computation graph , which will limit your model's portability.

[T-Tensorflow framework learning] Tensorflow "computation graph" introductory understanding