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

TensorFlow is an open-source library developed by Google primarily for deep learning applications. It also supports traditional machine learning. TensorFlow was originally developed for large numerical computations without keeping deep learning in mind.

Deep learning is a [subset of machine learning](https://www.simplilearn.com/machine-learning-vs-deep-learning-major-differences-you-need-to-know-article), and it works on the structure and functions similarly to the human brain. It learns from data that is unstructured and uses complex algorithms to train a neural net.

We primarily use [neural networks](https://www.simplilearn.com/tutorials/deep-learning-tutorial/neural-network) in deep learning, which is based on AI. Here, we train networks to recognize text, numbers, images, voice, and so on. Unlike traditional [machine learning](https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-machine-learning), the data here is far more complicated, unstructured, and varied, such as images, audio, or text files.

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

Numpy is a linear algebra library for python, and one of the most important and popular libraries in Data Science. TensorFlow is a reimplementation of the Numpy API and can be accessed as tf. experimental.

Here I am writing about the most interesting differences and similarities between Numpy and TensorFlow. I will update this post as I am learning more about the topics.

Numpy is a linear algebra library for python, and one of the most important and popular libraries in Data Science. TensorFlow is a reimplementation of the Numpy API and can be accessed as [tf.experimental.numpy](https://www.tensorflow.org/api_docs/python/tf/experimental/numpy" \t "_blank).

Numpy performs a wide variety of numerical computations in Python, and it is the fundamental package for scientific computing. It is generally used for multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. On the other hand, TensorFlow is an open-source software library for numerical computation using data flow graphs and aims to be a machine learning and deep learning library. However, both Numpy and TensorFlow behave very similar in many aspects ranging from building in the concept of tensor to the fact that they are both array manipulation libraries. From this standpoint, it might be difficult to recognize the exact differences. One of the important ways that TensorFlow augments Numpy is GPU acceleration to support very large concurrency. Although there is CuPy which is a GPU array backend that implements a subset of the NumPy interface, it is not part of the library itself. The other important advantage is that the TensorFlow API performs automatic gradient computation on the arbitrary potential field on tensors. All in all, TensorFlow and NumPy also work reasonably well together. We can obtain the NumPy version from TensorFlow objects. Also, the TensorFlow tensor can be directly converted to a NumPy array in eager mode. TensorFlow reimplements a large portion of NumPy API. Most NumPy API have TensorFlow equivalence. But there are three differences. First, TensorFlow renamed a few functions: np.concatenate → tf.concat. Second, TensorFlow uses function call convention over methods. Example:

NumPy: x.reshape(3,1)

TensorFlow: tf.reshape(x, (3,1))

Last but not least, TensorFlow is sensitive highly about datatypes used. They cannot be implicitly mixed.

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

tf.range(limit, delta=1, dtype=None, name='range')  
tf.range(start, limit, delta=1, dtype=None, name='range')

Creates a sequence of numbers that begins at start and extends by increments of delta up to but not including limit.

The dtype of the resulting tensor is inferred from the inputs unless it is provided explicitly.

Like the Python builtin range, start defaults to 0, so that range(n) = range(0, n).

For example:

start = 3  
limit = 18  
delta = 3  
tf.range(start, limit, delta)  
<tf.Tensor: shape=(5,), dtype=int32,  
numpy=array([ 3, 6, 9, 12, 15], dtype=int32)>

start = 3  
limit = 1  
delta = -0.5  
tf.range(start, limit, delta)  
<tf.Tensor: shape=(4,), dtype=float32,  
numpy=array([3. , 2.5, 2. , 1.5], dtype=float32)>

limit = 5  
tf.range(limit)  
<tf.Tensor: shape=(5,), dtype=int32,  
numpy=array([0, 1, 2, 3, 4], dtype=int32)>

|  |  |
| --- | --- |
| Args | |
| start | A 0-D Tensor (scalar). Acts as first entry in the range if limit is not None; otherwise, acts as range limit and first entry defaults to 0. |
| limit | A 0-D Tensor (scalar). Upper limit of sequence, exclusive. If None, defaults to the value of start while the first entry of the range defaults to 0. |
| delta | A 0-D Tensor (scalar). Number that increments start. Defaults to 1. |
| dtype | The type of the elements of the resulting tensor. |
| name | A name for the operation. Defaults to "range". |
| Returns | |
| An 1-D Tensor of type dtype.  4.Can you name six other data structures available in TensorFlow, beyond regular tensors?  import tensorflow as tf import numpy as np  Tensors are multi-dimensional arrays with a uniform type (called a dtype). You can see all supported dtypes at [tf.dtypes.DType](https://www.tensorflow.org/api_docs/python/tf/dtypes/DType).  If you're familiar with [NumPy](https://numpy.org/devdocs/user/quickstart.html), tensors are (kind of) like np.arrays.  All tensors are immutable like Python numbers and strings: you can never update the contents of a tensor, only create a new one.  Basics  First, create some basic tensors.  Here is a "scalar" or "rank-0" tensor . A scalar contains a single value, and no "axes".  # This will be an int32 tensor by default; see "dtypes" below. rank\_0\_tensor = tf.constant(4) print(rank\_0\_tensor)  tf.Tensor(4, shape=(), dtype=int32)  A "vector" or "rank-1" tensor is like a list of values. A vector has one axis:  # Let's make this a float tensor. rank\_1\_tensor = tf.constant([2.0, 3.0, 4.0]) print(rank\_1\_tensor)  tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)  A "matrix" or "rank-2" tensor has two axes:  # If you want to be specific, you can set the dtype (see below) at creation time rank\_2\_tensor = tf.constant([[1, 2],                              [3, 4],                              [5, 6]], dtype=tf.float16) print(rank\_2\_tensor)  tf.Tensor(  [[1. 2.]  [3. 4.]  [5. 6.]], shape=(3, 2), dtype=float16)  Tensors may have more axes; here is a tensor with three axes:  # There can be an arbitrary number of # axes (sometimes called "dimensions") rank\_3\_tensor = 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(rank\_3\_tensor)  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)  There are many ways you might visualize a tensor with more than two axes.   |  |  |  | | --- | --- | --- | | A 3-axis tensor, shape: [3, 2, 5] | | | |  |  |  |   You can convert a tensor to a NumPy array either using np.array or the tensor.numpy method:  np.array(rank\_2\_tensor)  array([[1., 2.],  [3., 4.],  [5., 6.]], dtype=float16)  rank\_2\_tensor.numpy()  array([[1., 2.],  [3., 4.],  [5., 6.]], dtype=float16)  Tensors often contain floats and ints, but have many other types, including:  complex numbers  strings  The base [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor) class requires tensors to be "rectangular"---that is, along each axis, every element is the same size. However, there are specialized types of tensors that can handle different shapes:  Ragged tensors (see [RaggedTensor](https://www.tensorflow.org/guide/tensor" \l "ragged_tensors) below)  Sparse tensors (see [SparseTensor](https://www.tensorflow.org/guide/tensor" \l "sparse_tensors) below)  You can do basic math on tensors, including addition, element-wise multiplication, and matrix multiplication.  a = tf.constant([[1, 2],                  [3, 4]]) b = tf.constant([[1, 1],                  [1, 1]]) # Could have also said `tf.ones([2,2])`  print(tf.add(a, b), "\n") print(tf.multiply(a, b), "\n") print(tf.matmul(a, b), "\n")  tf.Tensor(  [[2 3]  [4 5]], shape=(2, 2), dtype=int32)  tf.Tensor(  [[1 2]  [3 4]], shape=(2, 2), dtype=int32)  tf.Tensor(  [[3 3]  [7 7]], shape=(2, 2), dtype=int32)  print(a + b, "\n") # element-wise addition print(a \* b, "\n") # element-wise multiplication print(a @ b, "\n") # matrix multiplication  tf.Tensor(  [[2 3]  [4 5]], shape=(2, 2), dtype=int32)  tf.Tensor(  [[1 2]  [3 4]], shape=(2, 2), dtype=int32)  tf.Tensor(  [[3 3]  [7 7]], shape=(2, 2), dtype=int32)  Tensors are used in all kinds of operations (or "Ops").  c = tf.constant([[4.0, 5.0], [10.0, 1.0]])  # Find the largest value print(tf.reduce\_max(c)) # Find the index of the largest value print(tf.math.argmax(c)) # Compute the softmax print(tf.nn.softmax(c))  tf.Tensor(10.0, shape=(), dtype=float32)  tf.Tensor([1 0], shape=(2,), dtype=int64)  tf.Tensor(  [[2.6894143e-01 7.3105854e-01]  [9.9987662e-01 1.2339458e-04]], shape=(2, 2), dtype=float32)  5.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?  Losses  The purpose of loss functions is to compute the quantity that a model should seek to minimize during training.  Available losses  Note that all losses are available both via a class handle and via a function handle. The class handles enable you to pass configuration arguments to the constructor (e.g. loss\_fn = CategoricalCrossentropy(from\_logits=True)), and they perform reduction by default when used in a standalone way (see details below).  [Probabilistic losses](https://keras.io/api/losses/probabilistic_losses)  [BinaryCrossentropy class](https://keras.io/api/losses/probabilistic_losses/#binarycrossentropy-class)  [CategoricalCrossentropy class](https://keras.io/api/losses/probabilistic_losses/#categoricalcrossentropy-class)  [SparseCategoricalCrossentropy class](https://keras.io/api/losses/probabilistic_losses/#sparsecategoricalcrossentropy-class)  [Poisson class](https://keras.io/api/losses/probabilistic_losses/#poisson-class)  [binary\_crossentropy function](https://keras.io/api/losses/probabilistic_losses/#binary_crossentropy-function)  [categorical\_crossentropy function](https://keras.io/api/losses/probabilistic_losses/#categorical_crossentropy-function)  [sparse\_categorical\_crossentropy function](https://keras.io/api/losses/probabilistic_losses/#sparse_categorical_crossentropy-function)  [poisson function](https://keras.io/api/losses/probabilistic_losses/#poisson-function)  [KLDivergence class](https://keras.io/api/losses/probabilistic_losses/#kldivergence-class)  [kl\_divergence function](https://keras.io/api/losses/probabilistic_losses/#kl_divergence-function)  [Regression losses](https://keras.io/api/losses/regression_losses)  [MeanSquaredError class](https://keras.io/api/losses/regression_losses/#meansquarederror-class)  [MeanAbsoluteError class](https://keras.io/api/losses/regression_losses/#meanabsoluteerror-class)  [MeanAbsolutePercentageError class](https://keras.io/api/losses/regression_losses/#meanabsolutepercentageerror-class)  [MeanSquaredLogarithmicError class](https://keras.io/api/losses/regression_losses/#meansquaredlogarithmicerror-class)  [CosineSimilarity class](https://keras.io/api/losses/regression_losses/#cosinesimilarity-class)  [mean\_squared\_error function](https://keras.io/api/losses/regression_losses/#mean_squared_error-function)  [mean\_absolute\_error function](https://keras.io/api/losses/regression_losses/#mean_absolute_error-function)  [mean\_absolute\_percentage\_error function](https://keras.io/api/losses/regression_losses/#mean_absolute_percentage_error-function)  [mean\_squared\_logarithmic\_error function](https://keras.io/api/losses/regression_losses/#mean_squared_logarithmic_error-function)  [cosine\_similarity function](https://keras.io/api/losses/regression_losses/#cosine_similarity-function)  [Huber class](https://keras.io/api/losses/regression_losses/#huber-class)  [huber function](https://keras.io/api/losses/regression_losses/#huber-function)  [LogCosh class](https://keras.io/api/losses/regression_losses/#logcosh-class)  [log\_cosh function](https://keras.io/api/losses/regression_losses/#log_cosh-function)  [Hinge losses for "maximum-margin" classification](https://keras.io/api/losses/hinge_losses)  [Hinge class](https://keras.io/api/losses/hinge_losses/#hinge-class)  [SquaredHinge class](https://keras.io/api/losses/hinge_losses/#squaredhinge-class)  [CategoricalHinge class](https://keras.io/api/losses/hinge_losses/#categoricalhinge-class)  [hinge function](https://keras.io/api/losses/hinge_losses/#hinge-function)  [squared\_hinge function](https://keras.io/api/losses/hinge_losses/#squared_hinge-function)  [categorical\_hinge function](https://keras.io/api/losses/hinge_losses/#categorical_hinge-function)  Usage of losses with compile() & fit()  A loss function is one of the two arguments required for compiling a Keras model:  from tensorflow import keras  from tensorflow.keras import layers  model = keras.Sequential()  model.add(layers.Dense(64, kernel\_initializer='uniform', input\_shape=(10,)))  model.add(layers.Activation('softmax'))  loss\_fn = keras.losses.SparseCategoricalCrossentropy()  model.compile(loss=loss\_fn, optimizer='adam')  All built-in loss functions may also be passed via their string identifier:  # pass optimizer by name: default parameters will be used  model.compile(loss='sparse\_categorical\_crossentropy', optimizer='adam')  Loss functions are typically created by instantiating a loss class (e.g. [keras.losses.SparseCategoricalCrossentropy](https://keras.io/api/losses/probabilistic_losses" \l "sparsecategoricalcrossentropy-class)). All losses are also provided as function handles (e.g. [keras.losses.sparse\_categorical\_crossentropy](https://keras.io/api/losses/probabilistic_losses" \l "sparsecategoricalcrossentropy-function)).  Using classes enables you to pass configuration arguments at instantiation time, e.g.:  loss\_fn = keras.losses.SparseCategoricalCrossentropy(from\_logits=True)  Standalone usage of losses  A loss is a callable with arguments loss\_fn(y\_true, y\_pred, sample\_weight=None):  y\_true: Ground truth values, of shape (batch\_size, d0, ... dN). For sparse loss functions, such as sparse categorical crossentropy, the shape should be (batch\_size, d0, ... dN-1)  y\_pred: The predicted values, of shape (batch\_size, d0, .. dN).  sample\_weight: Optional sample\_weight acts as reduction weighting coefficient for the per-sample losses. If a scalar is provided, then the loss is simply scaled by the given value. If sample\_weight is a tensor of size [batch\_size], then the total loss for each sample of the batch is rescaled by the corresponding element in the sample\_weight vector. If the shape of sample\_weight is (batch\_size, d0, ... dN-1) (or can be broadcasted to this shape), then each loss element of y\_pred is scaled by the corresponding value of sample\_weight. (Note ondN-1: all loss functions reduce by 1 dimension, usually axis=-1.)  By default, loss functions return one scalar loss value per input sample, e.g.  >>> tf.keras.losses.mean\_squared\_error(tf.ones((2, 2,)), tf.zeros((2, 2)))  <tf.Tensor: shape=(2,), dtype=float32, numpy=array([1., 1.], dtype=float32)>  However, loss class instances feature a reduction constructor argument, which defaults to "sum\_over\_batch\_size" (i.e. average). Allowable values are "sum\_over\_batch\_size", "sum", and "none":  "sum\_over\_batch\_size" means the loss instance will return the average of the per-sample losses in the batch.  "sum" means the loss instance will return the sum of the per-sample losses in the batch.  "none" means the loss instance will return the full array of per-sample losses.  >>> loss\_fn = tf.keras.losses.MeanSquaredError(reduction='sum\_over\_batch\_size')  >>> loss\_fn(tf.ones((2, 2,)), tf.zeros((2, 2)))  <tf.Tensor: shape=(), dtype=float32, numpy=1.0>  >>> loss\_fn = tf.keras.losses.MeanSquaredError(reduction='sum')  >>> loss\_fn(tf.ones((2, 2,)), tf.zeros((2, 2)))  <tf.Tensor: shape=(), dtype=float32, numpy=2.0>  >>> loss\_fn = tf.keras.losses.MeanSquaredError(reduction='none')  >>> loss\_fn(tf.ones((2, 2,)), tf.zeros((2, 2)))  <tf.Tensor: shape=(2,), dtype=float32, numpy=array([1., 1.], dtype=float32)>  Note that this is an important difference between loss functions like [tf.keras.losses.mean\_squared\_error](https://keras.io/api/losses/regression_losses" \l "meansquarederror-function) and default loss class instances like [tf.keras.losses.MeanSquaredError](https://keras.io/api/losses/regression_losses" \l "meansquarederror-class): the function version does not perform reduction, but by default the class instance does.  >>> loss\_fn = tf.keras.losses.mean\_squared\_error  >>> loss\_fn(tf.ones((2, 2,)), tf.zeros((2, 2)))  <tf.Tensor: shape=(2,), dtype=float32, numpy=array([1., 1.], dtype=float32)>  >>> loss\_fn = tf.keras.losses.MeanSquaredError()  >>> loss\_fn(tf.ones((2, 2,)), tf.zeros((2, 2)))  <tf.Tensor: shape=(), dtype=float32, numpy=1.0>  When using fit(), this difference is irrelevant since reduction is handled by the framework.  Here's how you would use a loss class instance as part of a simple training loop:  loss\_fn = tf.keras.losses.CategoricalCrossentropy(from\_logits=True)  optimizer = tf.keras.optimizers.Adam()  # Iterate over the batches of a dataset.  for x, y in dataset:  with tf.GradientTape() as tape:  logits = model(x)  # Compute the loss value for this batch.  loss\_value = loss\_fn(y, logits)  # Update the weights of the model to minimize the loss value.  gradients = tape.gradient(loss\_value, model.trainable\_weights)  optimizer.apply\_gradients(zip(gradients, model.trainable\_weights))  Creating custom losses  Any callable with the signature loss\_fn(y\_true, y\_pred) that returns an array of losses (one of sample in the input batch) can be passed to compile() as a loss. Note that sample weighting is automatically supported for any such loss.  Here's a simple example:  def my\_loss\_fn(y\_true, y\_pred):  squared\_difference = tf.square(y\_true - y\_pred)  return tf.reduce\_mean(squared\_difference, axis=-1) # Note the `axis=-1`  model.compile(optimizer='adam', loss=my\_loss\_fn)  The add\_loss() API  Loss functions applied to the output of a model aren't the only way to create losses.  When writing the call method of a custom layer or a subclassed model, you may want to compute scalar quantities that you want to minimize during training (e.g. regularization losses). You can use the add\_loss() layer method to keep track of such loss terms.  Here's an example of a layer that adds a sparsity regularization loss based on the L2 norm of the inputs:  from tensorflow.keras.layers import Layer  class MyActivityRegularizer(Layer):  """Layer that creates an activity sparsity regularization loss."""  def \_\_init\_\_(self, rate=1e-2):  super(MyActivityRegularizer, self).\_\_init\_\_()  self.rate = rate  def call(self, inputs):  # We use `add\_loss` to create a regularization loss  # that depends on the inputs.  self.add\_loss(self.rate \* tf.reduce\_sum(tf.square(inputs)))  return inputs  Loss values added via add\_loss can be retrieved in the .losses list property of any Layer or Model (they are recursively retrieved from every underlying layer):  from tensorflow.keras import layers  class SparseMLP(Layer):  """Stack of Linear layers with a sparsity regularization loss."""  def \_\_init\_\_(self, output\_dim):  super(SparseMLP, self).\_\_init\_\_()  self.dense\_1 = layers.Dense(32, activation=tf.nn.relu)  self.regularization = MyActivityRegularizer(1e-2)  self.dense\_2 = layers.Dense(output\_dim)  def call(self, inputs):  x = self.dense\_1(inputs)  x = self.regularization(x)  return self.dense\_2(x)  mlp = SparseMLP(1)  y = mlp(tf.ones((10, 10)))  print(mlp.losses) # List containing one float32 scalar  These losses are cleared by the top-level layer at the start of each forward pass -- they don't accumulate. So layer.losses always contain only the losses created during the last forward pass. You would typically use these losses by summing them before computing your gradients when writing a training loop.  # Losses correspond to the \*last\* forward pass.  mlp = SparseMLP(1)  mlp(tf.ones((10, 10)))  assert len(mlp.losses) == 1  mlp(tf.ones((10, 10)))  assert len(mlp.losses) == 1 # No accumulation.  When using model.fit(), such loss terms are handled automatically.  When writing a custom training loop, you should retrieve these terms by hand from model.losses, like this:  loss\_fn = tf.keras.losses.CategoricalCrossentropy(from\_logits=True)  optimizer = tf.keras.optimizers.Adam()  # Iterate over the batches of a dataset.  for x, y in dataset:  with tf.GradientTape() as tape:  # Forward pass.  logits = model(x)  # Loss value for this batch.  loss\_value = loss\_fn(y, logits)  # Add extra loss terms to the loss value.  loss\_value += sum(model.losses)  # Update the weights of the model to minimize the loss value.  gradients = tape.gradient(loss\_value, model.trainable\_weights)  optimizer.apply\_gradients(zip(gradients, model.trainable\_weights))  See [the add\_loss() documentation](https://keras.io/api/layers/base_layer/#add_loss-method) for more details.  6.Similarly, a custom metric can be defined in a function or a subclass of keras.metrics.Metric. When would you use each option?  Metrics  A metric is a function that is used to judge the performance of your model.  Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. Note that you may use any loss function as a metric.  Available metrics  [Accuracy metrics](https://keras.io/api/metrics/accuracy_metrics)  [Accuracy class](https://keras.io/api/metrics/accuracy_metrics/#accuracy-class)  [BinaryAccuracy class](https://keras.io/api/metrics/accuracy_metrics/#binaryaccuracy-class)  [CategoricalAccuracy class](https://keras.io/api/metrics/accuracy_metrics/#categoricalaccuracy-class)  [SparseCategoricalAccuracy class](https://keras.io/api/metrics/accuracy_metrics/#sparsecategoricalaccuracy-class)  [TopKCategoricalAccuracy class](https://keras.io/api/metrics/accuracy_metrics/#topkcategoricalaccuracy-class)  [SparseTopKCategoricalAccuracy class](https://keras.io/api/metrics/accuracy_metrics/#sparsetopkcategoricalaccuracy-class)  [Probabilistic metrics](https://keras.io/api/metrics/probabilistic_metrics)  [BinaryCrossentropy class](https://keras.io/api/metrics/probabilistic_metrics/#binarycrossentropy-class)  [CategoricalCrossentropy class](https://keras.io/api/metrics/probabilistic_metrics/#categoricalcrossentropy-class)  [SparseCategoricalCrossentropy class](https://keras.io/api/metrics/probabilistic_metrics/#sparsecategoricalcrossentropy-class)  [KLDivergence class](https://keras.io/api/metrics/probabilistic_metrics/#kldivergence-class)  [Poisson class](https://keras.io/api/metrics/probabilistic_metrics/#poisson-class)  [Regression metrics](https://keras.io/api/metrics/regression_metrics)  [MeanSquaredError class](https://keras.io/api/metrics/regression_metrics/#meansquarederror-class)  [RootMeanSquaredError class](https://keras.io/api/metrics/regression_metrics/#rootmeansquarederror-class)  [MeanAbsoluteError class](https://keras.io/api/metrics/regression_metrics/#meanabsoluteerror-class)  [MeanAbsolutePercentageError class](https://keras.io/api/metrics/regression_metrics/#meanabsolutepercentageerror-class)  [MeanSquaredLogarithmicError class](https://keras.io/api/metrics/regression_metrics/#meansquaredlogarithmicerror-class)  [CosineSimilarity class](https://keras.io/api/metrics/regression_metrics/#cosinesimilarity-class)  [LogCoshError class](https://keras.io/api/metrics/regression_metrics/#logcosherror-class)  [Classification metrics based on True/False positives & negatives](https://keras.io/api/metrics/classification_metrics)  [AUC class](https://keras.io/api/metrics/classification_metrics/#auc-class)  [Precision class](https://keras.io/api/metrics/classification_metrics/#precision-class)  [Recall class](https://keras.io/api/metrics/classification_metrics/#recall-class)  [TruePositives class](https://keras.io/api/metrics/classification_metrics/#truepositives-class)  [TrueNegatives class](https://keras.io/api/metrics/classification_metrics/#truenegatives-class)  [FalsePositives class](https://keras.io/api/metrics/classification_metrics/#falsepositives-class)  [FalseNegatives class](https://keras.io/api/metrics/classification_metrics/#falsenegatives-class)  [PrecisionAtRecall class](https://keras.io/api/metrics/classification_metrics/#precisionatrecall-class)  [SensitivityAtSpecificity class](https://keras.io/api/metrics/classification_metrics/#sensitivityatspecificity-class)  [SpecificityAtSensitivity class](https://keras.io/api/metrics/classification_metrics/#specificityatsensitivity-class)  [Image segmentation metrics](https://keras.io/api/metrics/segmentation_metrics)  [MeanIoU class](https://keras.io/api/metrics/segmentation_metrics/#meaniou-class)  [Hinge metrics for "maximum-margin" classification](https://keras.io/api/metrics/hinge_metrics)  [Hinge class](https://keras.io/api/metrics/hinge_metrics/#hinge-class)  [SquaredHinge class](https://keras.io/api/metrics/hinge_metrics/#squaredhinge-class)  [CategoricalHinge class](https://keras.io/api/metrics/hinge_metrics/#categoricalhinge-class)  Usage with compile() & fit()  The compile() method takes a metrics argument, which is a list of metrics:  model.compile(  optimizer='adam',  loss='mean\_squared\_error',  metrics=[  metrics.MeanSquaredError(),  metrics.AUC(),  ]  )  Metric values are displayed during fit() and logged to the History object returned by fit(). They are also returned by model.evaluate().  Note that the best way to monitor your metrics during training is via [TensorBoard](https://keras.io/api/callbacks/tensorboard).  To track metrics under a specific name, you can pass the name argument to the metric constructor:  model.compile(  optimizer='adam',  loss='mean\_squared\_error',  metrics=[  metrics.MeanSquaredError(name='my\_mse'),  metrics.AUC(name='my\_auc'),  ]  )  All built-in metrics may also be passed via their string identifier (in this case, default constructor argument values are used, including a default metric name):  model.compile(  optimizer='adam',  loss='mean\_squared\_error',  metrics=[  'MeanSquaredError',  'AUC',  ]  )  Standalone usage  Unlike losses, metrics are stateful. You update their state using the update\_state() method, and you query the scalar metric result using the result() method:  m = tf.keras.metrics.AUC()  m.update\_state([0, 1, 1, 1], [0, 1, 0, 0])  print('Intermediate result:', float(m.result()))  m.update\_state([1, 1, 1, 1], [0, 1, 1, 0])  print('Final result:', float(m.result()))  The internal state can be cleared via metric.reset\_states().  Here's how you would use a metric as part of a simple custom training loop:  accuracy = tf.keras.metrics.CategoricalAccuracy()  loss\_fn = tf.keras.losses.CategoricalCrossentropy(from\_logits=True)  optimizer = tf.keras.optimizers.Adam()  # Iterate over the batches of a dataset.  for step, (x, y) in enumerate(dataset):  with tf.GradientTape() as tape:  logits = model(x)  # Compute the loss value for this batch.  loss\_value = loss\_fn(y, logits)  # Update the state of the `accuracy` metric.  accuracy.update\_state(y, logits)  # Update the weights of the model to minimize the loss value.  gradients = tape.gradient(loss\_value, model.trainable\_weights)  optimizer.apply\_gradients(zip(gradients, model.trainable\_weights))  # Logging the current accuracy value so far.  if step % 100 == 0:  print('Step:', step)  print('Total running accuracy so far: %.3f' % accuracy.result())  Creating custom metrics  As simple callables (stateless)  Much like loss functions, any callable with signature metric\_fn(y\_true, y\_pred) that returns an array of losses (one of sample in the input batch) can be passed to compile() as a metric. Note that sample weighting is automatically supported for any such metric.  Here's a simple example:  def my\_metric\_fn(y\_true, y\_pred):  squared\_difference = tf.square(y\_true - y\_pred)  return tf.reduce\_mean(squared\_difference, axis=-1) # Note the `axis=-1`  model.compile(optimizer='adam', loss='mean\_squared\_error', metrics=[my\_metric\_fn])  In this case, the scalar metric value you are tracking during training and evaluation is the average of the per-batch metric values for all batches see during a given epoch (or during a given call to model.evaluate()).  As subclasses of Metric (stateful)  Not all metrics can be expressed via stateless callables, because metrics are evaluated for each batch during training and evaluation, but in some cases the average of the per-batch values is not what you are interested in.  Let's say that you want to compute AUC over a given evaluation dataset: the average of the per-batch AUC values isn't the same as the AUC over the entire dataset.  For such metrics, you're going to want to subclass the Metric class, which can maintain a state across batches. It's easy:  Create the state variables in \_\_init\_\_  Update the variables given y\_true and y\_pred in update\_state()  Return the scalar metric result in result()  Clear the state in reset\_states()  Here's a simple example computing binary true positives:  class BinaryTruePositives(tf.keras.metrics.Metric):  def \_\_init\_\_(self, name='binary\_true\_positives', \*\*kwargs):  super(BinaryTruePositives, self).\_\_init\_\_(name=name, \*\*kwargs)  self.true\_positives = self.add\_weight(name='tp', initializer='zeros')  def update\_state(self, y\_true, y\_pred, sample\_weight=None):  y\_true = tf.cast(y\_true, tf.bool)  y\_pred = tf.cast(y\_pred, tf.bool)  values = tf.logical\_and(tf.equal(y\_true, True), tf.equal(y\_pred, True))  values = tf.cast(values, self.dtype)  if sample\_weight is not None:  sample\_weight = tf.cast(sample\_weight, self.dtype)  values = tf.multiply(values, sample\_weight)  self.true\_positives.assign\_add(tf.reduce\_sum(values))  def result(self):  return self.true\_positives  def reset\_states(self):  self.true\_positives.assign(0)  m = BinaryTruePositives()  m.update\_state([0, 1, 1, 1], [0, 1, 0, 0])  print('Intermediate result:', float(m.result()))  m.update\_state([1, 1, 1, 1], [0, 1, 1, 0])  print('Final result:', float(m.result()))  The add\_metric() API  When writing the forward pass of a custom layer or a subclassed model, you may sometimes want to log certain quantities on the fly, as metrics. In such cases, you can use the add\_metric() method.  Let's say you want to log as metric the mean of the activations of a Dense-like custom layer. You could do the following:  class DenseLike(Layer):  """y = w.x + b"""  ...  def call(self, inputs):  output = tf.matmul(inputs, self.w) + self.b  self.add\_metric(tf.reduce\_mean(output), aggregation='mean', name='activation\_mean')  return output  The quantity will then tracked under the name "activation\_mean". The value tracked will be the average of the per-batch metric metric values (as specified by aggregation).  7.When should you create a custom layer versus a custom model?  If you are building a new model architecture using existing keras/tf layers then build a custom model. If you are implementing your own custom tensor operations with in a layer, then build a custom layer.  Layers: common sets of useful operations  Most of the time when writing code for machine learning models you want to operate at a higher level of abstraction than individual operations and manipulation of individual variables.  Many machine learning models are expressible as the composition and stacking of relatively simple layers, and TensorFlow provides both a set of many common layers as well as easy ways for you to write your own application-specific layers either from scratch or as the composition of existing layers.  TensorFlow includes the full [Keras](https://keras.io/) API in the tf.keras package, and the Keras layers are very useful when building your own models.  # In the tf.keras.layers package, layers are objects. To construct a layer, # simply construct the object. Most layers take as a first argument the number # of output dimensions / channels. layer = tf.keras.layers.Dense(100) # The number of input dimensions is often unnecessary, as it can be inferred # the first time the layer is used, but it can be provided if you want to # specify it manually, which is useful in some complex models. layer = tf.keras.layers.Dense(10, input\_shape=(None, 5))  The full list of pre-existing layers can be seen in [the documentation](https://www.tensorflow.org/api_docs/python/tf/keras/layers). It includes Dense (a fully-connected layer), Conv2D, LSTM, BatchNormalization, Dropout, and many others.  # To use a layer, simply call it. layer(tf.zeros([10, 5]))  <tf.Tensor: shape=(10, 10), dtype=float32, numpy=  array([[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., 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., 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.]], dtype=float32)>  # Layers have many useful methods. For example, you can inspect all variables # in a layer using `layer.variables` and trainable variables using # `layer.trainable\_variables`. In this case a fully-connected layer # will have variables for weights and biases. layer.variables  [<tf.Variable 'dense\_1/kernel:0' shape=(5, 10) dtype=float32, numpy=  array([[ 0.23369527, 0.6311286 , 0.3390423 , -0.18410993, -0.24590981,  0.35261846, -0.34033006, -0.09950155, -0.2522891 , 0.0554117 ],  [-0.18508238, 0.53696483, -0.12825328, -0.1419357 , 0.4807914 ,  0.4538824 , -0.36918357, -0.11204702, -0.45423228, -0.5898261 ],  [-0.12760967, -0.3141594 , 0.08622831, 0.11372548, -0.2741867 ,  0.57630974, -0.18668345, -0.48103276, 0.44950897, -0.17167208],  [ 0.6198546 , -0.13317591, 0.53580827, -0.5101569 , -0.06103915,  0.00936204, 0.53685576, 0.44844192, -0.4400819 , 0.34407234],  [ 0.05848145, -0.02604926, 0.2680108 , 0.08222908, 0.12862706,  0.2059567 , 0.29267555, -0.19178253, -0.03098279, -0.20128557]],  dtype=float32)>,  <tf.Variable 'dense\_1/bias:0' shape=(10,) dtype=float32, numpy=array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)>]  # The variables are also accessible through nice accessors layer.kernel, layer.bias  (<tf.Variable 'dense\_1/kernel:0' shape=(5, 10) dtype=float32, numpy=  array([[ 0.23369527, 0.6311286 , 0.3390423 , -0.18410993, -0.24590981,  0.35261846, -0.34033006, -0.09950155, -0.2522891 , 0.0554117 ],  [-0.18508238, 0.53696483, -0.12825328, -0.1419357 , 0.4807914 ,  0.4538824 , -0.36918357, -0.11204702, -0.45423228, -0.5898261 ],  [-0.12760967, -0.3141594 , 0.08622831, 0.11372548, -0.2741867 ,  0.57630974, -0.18668345, -0.48103276, 0.44950897, -0.17167208],  [ 0.6198546 , -0.13317591, 0.53580827, -0.5101569 , -0.06103915,  0.00936204, 0.53685576, 0.44844192, -0.4400819 , 0.34407234],  [ 0.05848145, -0.02604926, 0.2680108 , 0.08222908, 0.12862706,  0.2059567 , 0.29267555, -0.19178253, -0.03098279, -0.20128557]],  dtype=float32)>,  <tf.Variable 'dense\_1/bias:0' shape=(10,) dtype=float32, numpy=array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)>)  Implementing custom layers  The best way to implement your own layer is extending the tf.keras.Layer class and implementing:  \_\_init\_\_ , where you can do all input-independent initialization  build, where you know the shapes of the input tensors and can do the rest of the initialization  call, where you do the forward computation  Note that you don't have to wait until build is called to create your variables, you can also create them in \_\_init\_\_. However, the advantage of creating them in build is that it enables late variable creation based on the shape of the inputs the layer will operate on. On the other hand, creating variables in \_\_init\_\_ would mean that shapes required to create the variables will need to be explicitly specified.  class MyDenseLayer(tf.keras.layers.Layer):   def \_\_init\_\_(self, num\_outputs):     super(MyDenseLayer, self).\_\_init\_\_()     self.num\_outputs = num\_outputs    def build(self, input\_shape):     self.kernel = self.add\_weight("kernel",                                   shape=[int(input\_shape[-1]),                                          self.num\_outputs])    def call(self, inputs):     return tf.matmul(inputs, self.kernel)  layer = MyDenseLayer(10)  \_ = layer(tf.zeros([10, 5])) # Calling the layer `.builds` it.  print([var.name for var in layer.trainable\_variables])  ['my\_dense\_layer/kernel:0']  Overall code is easier to read and maintain if it uses standard layers whenever possible, as other readers will be familiar with the behavior of standard layers. If you want to use a layer which is not present in [tf.keras.layers](https://www.tensorflow.org/api_docs/python/tf/keras/layers), consider filing a [github issue](http://github.com/tensorflow/tensorflow/issues/new) or, even better, sending us a pull request!  8.What are some use cases that require writing your own custom training loop?  How do you write a training loop?  How to write a training loop in Chainer  Prepare a dataset.  Create a dataset iterator.  Define a network.  Select an optimization algorithm.  Write a training loop. ...  Save the trained model.  Perform classification by the saved model and check the network performance on validation/test sets.  In this tutorial section, we will learn how to train a deep neural network to classify images of hand-written digits in the popular MNIST dataset. This dataset contains 50,000 training examples and 10,000 test examples. Each example is a set of a 28 x 28 greyscale image and a corresponding class label. Since the digits from 0 to 9 are used, there are 10 classes for the labels.  Chainer provides a feature called [Trainer](https://docs.chainer.org/en/v3.2.0/reference/core/generated/chainer.training.Trainer.html#chainer.training.Trainer) that can simplify the training procedure of your model. However, it is also good to know how the training works in Chainer before starting to use the useful [Trainer](https://docs.chainer.org/en/v3.2.0/reference/core/generated/chainer.training.Trainer.html#chainer.training.Trainer) class that hides the actual processes. Writing your own training loop can be useful for learning how [Trainer](https://docs.chainer.org/en/v3.2.0/reference/core/generated/chainer.training.Trainer.html#chainer.training.Trainer) works or for implementing features not included in the standard trainer.  The complete training procedure consists of the following steps:  [Prepare a dataset](https://docs.chainer.org/en/v3.2.0/tutorial/train_loop.html#prepare-a-dataset)  [Create a dataset iterator](https://docs.chainer.org/en/v3.2.0/tutorial/train_loop.html#create-an-iterator)  [Define a network](https://docs.chainer.org/en/v3.2.0/tutorial/train_loop.html#define-a-network)  [Select an optimization algorithm](https://docs.chainer.org/en/v3.2.0/tutorial/train_loop.html#select-an-optimization-algorithm)  [Write a training loop](https://docs.chainer.org/en/v3.2.0/tutorial/train_loop.html#write-a-training-loop)  Retrieve a set of examples (mini-batch) from the training dataset.  Feed the mini-batch to your network.  Run a forward pass of the network and compute the loss.  Just call the [backward()](https://docs.chainer.org/en/v3.2.0/reference/core/generated/chainer.Variable.html#chainer.Variable.backward) method from the loss [Variable](https://docs.chainer.org/en/v3.2.0/reference/core/generated/chainer.Variable.html#chainer.Variable) to compute the gradients for all trainable parameters.  Run the optimizer to update those parameters.  [Save the trained model](https://docs.chainer.org/en/v3.2.0/tutorial/train_loop.html#save-the-trained-model)  [Perform classification by the saved model](https://docs.chainer.org/en/v3.2.0/tutorial/train_loop.html#perform-classification) and check the network performance on validation/test sets.  9.Can custom Keras components contain arbitrary Python code, or must they be convertible to TF Functions?  If you know calculus, you can analytically find that the partial derivative of this function with regard to w1 is 6 \* w1 + 2 \* w2. You can also find that its partial derivative with regard to w2 is 2 \* w1. For example, at the point (w1, w2) = (5, 3), these partial derivatives are equal to 36 and 10, respectively, so the gradient vector at this point is (36, 10). But if this were a neural network, the function would be much more complex, typically with tens of thousands of parameters, and finding the partial derivatives analytically by hand would be an almost impossible task. One solution could be to compute an approximation of each partial derivative by measuring how much the function’s output changes when you tweak the corresponding parameter:  w1, w2 = 5,3  eps = 1e-6  ( f(w1+eps, w2)-f(w1,w2) )/eps, ( f(w1, w2+eps)-f(w1, w2) )/eps    Looks about right! This works rather well and is easy to implement, but it is just an approximation, and importantly you need to call f() at least once per parameter (not twice, since we could compute f(w1, w2) just once). Needing to call f() at least once  per parameter makes this approach intractable for large neural networks. So instead, we should use autodiff. TensorFlow makes this pretty simple:  w1, w2 = tf.Variable(5.), tf.Variable(3.)  with tf.GradientTape() as tape:  z = f(w1,w2)    gradients = tape.gradient(z, [w1,w2])  gradients  We first define two variables w1 and w2, then we create a tf.GradientTape context that will automatically record every operation that involves a variable, and finally we ask this tape to compute the gradients of the result z with regard to both variables [w1, w2]. Let’s take a look at the gradients that TensorFlow computed:  Perfect! Not only is the result accurate (the precision is only limited by the floatingpoint errors), but the gradient() method only goes through the recorded computations once (in reverse order), no matter how many variables there are, so it is  incredibly efficient. It’s like magic!  To save memory, only put the strict minimum inside the tf.GradientTape() block. Alternatively, pause recording by creating a  with tape.stop\_recording() block inside the tf.GradientTape() block.  The tape is automatically erased immediately after you call its gradient() method, so you will get an exception if you try to call gradient() twice:  with tf.GradientTape() as tape:  z=f(w1, w2)  dz\_dw1 = tape.gradient(z, w1) # => tensor 36.0  print(dz\_dw1)  dz\_dw2 = tape.gradient(z, w2) # RuntimeError  print(dz\_dw2)  Reason: GradientTape.gradient can only be called once on non-persistent tapes.  If you need to call gradient() more than once, you must make the tape persistent and delete it each time you are done with it to free resources:  with tf.GradientTape(persistent=True) as tape:  z=f(w1, w2)  dz\_dw1 = tape.gradient(z, w1)  print(dz\_dw1)  dz\_dw2 = tape.gradient(z, w2)## works now!  print(dz\_dw2)  del tape  By default, the tape will only track operations involving variables, so if you try to compute the gradient of z with regard to anything other than a variable, the result will be None:  c1, c2 = tf.constant(5.), tf.constant(3.)  with tf.GradientTape() as tape:  z = f(c1, c2)  gradients = tape.gradient(z,[c1, c2])  gradients  However, you can force the tape to watch any tensors you like, to record every operation that involves them. You can then compute gradients with regard to these tensors, as if they were variables:  c1, c2 = tf.constant(5.), tf.constant(3.)  with tf.GradientTape() as tape:  tape.watch(c1) ###  tape.watch(c2) ###  z = f(c1, c2)    gradients = tape.gradient(z,[c1, c2])  gradients  with tf.GradientTape(persistent=True) as tape:  z1 = f(w1, w2+2.)  z2 = f(w1, w2+5.)  z3 = f(w1, w2+7.)    x=tf.stack([tape.gradient(z, [w1,w2])  for z in (z1, z2, z3)  ])  x  tf.reduce\_sum(x, axis=0)  This can be useful in some cases, like if you want to implement a regularization loss that penalizes activations that vary a lot when the inputs vary little: the loss will be based on the gradient of the activations with regard to the inputs. Since the inputs are not variables, you would need to tell the tape to watch them.  Most of the time a gradient tape is used to compute the gradients of a single value (usually the loss) with regard to a set of values (usually the model parameters). This is where reverse-mode autodiff shines, as it just needs to do one forward pass and one reverse pass to get all the gradients at once. If you try to compute the gradients of a vector, for example a vector containing multiple losses, then TensorFlow will compute the gradients of the vector’s sum. So if you ever need to get the individual gradients (e.g., the gradients of each loss with regard to the model parameters), you must call the tape’s jacobian() method: it will perform reverse-mode autodiff once for each loss in the vector (all in parallel by default). It is even possible to compute second-order partial derivatives (the Hessians, i.e., the partial derivatives of the partial derivatives), but this is rarely needed in practice.  10.What are the main rules to respect if you want a function to be convertible to a TF Function?  In TensorFlow 2, [eager execution](https://www.tensorflow.org/guide/eager) is turned on by default. The user interface is intuitive and flexible (running one-off operations is much easier and faster), but this can come at the expense of performance and deployability.  You can use [tf.function](https://www.tensorflow.org/api_docs/python/tf/function) to make graphs out of your programs. It is a transformation tool that creates Python-independent dataflow graphs out of your Python code. This will help you create performant and portable models, and it is required to use SavedModel.  This guide will help you conceptualize how [tf.function](https://www.tensorflow.org/api_docs/python/tf/function) works under the hood, so you can use it effectively.  The main takeaways and recommendations are:  Debug in eager mode, then decorate with [@tf.function](https://www.tensorflow.org/api_docs/python/tf/function).  Don't rely on Python side effects like object mutation or list appends.  [tf.function](https://www.tensorflow.org/api_docs/python/tf/function) works best with TensorFlow ops; NumPy and Python calls are converted to constants.  Setup  # Update TensorFlow, as this notebook requires version 2.9 or later !pip install -q -U tensorflow>=2.9.0 import tensorflow as tf  Define a helper function to demonstrate the kinds of errors you might encounter:  import traceback import contextlib  # Some helper code to demonstrate the kinds of errors you might encounter. @contextlib.contextmanager def assert\_raises(error\_class):   try:     yield   except error\_class as e:     print('Caught expected exception \n  {}:'.format(error\_class))     traceback.print\_exc(limit=2)   except Exception as e:     raise e   else:     raise Exception('Expected {} to be raised but no error was raised!'.format(         error\_class))  Basics  Usage  A Function you define (for example by applying the [@tf.function](https://www.tensorflow.org/api_docs/python/tf/function) decorator) is just like a core TensorFlow operation: You can execute it eagerly; you can compute gradients; and so on.  @tf.function  # The decorator converts `add` into a `Function`. def add(a, b):   return a + b  add(tf.ones([2, 2]), tf.ones([2, 2]))  #  [[2., 2.], [2., 2.]]  <tf.Tensor: shape=(2, 2), dtype=float32, numpy=  array([[2., 2.],  [2., 2.]], dtype=float32)>  v = tf.Variable(1.0) with tf.GradientTape() as tape:   result = add(v, 1.0) tape.gradient(result, v)  <tf.Tensor: shape=(), dtype=float32, numpy=1.0>  You can use Functions inside other Functions.  @tf.function def dense\_layer(x, w, b):   return add(tf.matmul(x, w), b)  dense\_layer(tf.ones([3, 2]), tf.ones([2, 2]), tf.ones([2]))  <tf.Tensor: shape=(3, 2), dtype=float32, numpy=  array([[3., 3.],  [3., 3.],  [3., 3.]], dtype=float32)>  Functions can be faster than eager code, especially for graphs with many small ops. But for graphs with a few expensive ops (like convolutions), you may not see much speedup.  import timeit conv\_layer = tf.keras.layers.Conv2D(100, 3)  @tf.function def conv\_fn(image):   return conv\_layer(image)  image = tf.zeros([1, 200, 200, 100]) # Warm up conv\_layer(image); conv\_fn(image) print("Eager conv:", timeit.timeit(lambda: conv\_layer(image), number=10)) print("Function conv:", timeit.timeit(lambda: conv\_fn(image), number=10)) print("Note how there's not much difference in performance for convolutions")  Eager conv: 0.004911784999990232  Function conv: 0.005145685999991656  Note how there's not much difference in performance for convolutions  Tracing  This section exposes how Function works under the hood, including implementation details which may change in the future. However, once you understand why and when tracing happens, it's much easier to use [tf.function](https://www.tensorflow.org/api_docs/python/tf/function) effectively!  What is "tracing"?  A Function runs your program in a [TensorFlow Graph](https://www.tensorflow.org/guide/intro_to_graphs#what_are_graphs). However, a [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph) cannot represent all the things that you'd write in an eager TensorFlow program. For instance, Python supports polymorphism, but [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph) requires its inputs to have a specified data type and dimension. Or you may perform side tasks like reading command-line arguments, raising an error, or working with a more complex Python object; none of these things can run in a [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph).  Function bridges this gap by separating your code in two stages:  1) In the first stage, referred to as "tracing", Function creates a new [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph). Python code runs normally, but all TensorFlow operations (like adding two Tensors) are deferred: they are captured by the [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph) and not run.  2) In the second stage, a [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph) which contains everything that was deferred in the first stage is run. This stage is much faster than the tracing stage.  Depending on its inputs, Function will not always run the first stage when it is called. See ["Rules of tracing"](https://www.tensorflow.org/guide/function#rules_of_tracing) below to get a better sense of how it makes that determination. Skipping the first stage and only executing the second stage is what gives you TensorFlow's high performance.  When Function does decide to trace, the tracing stage is immediately followed by the second stage, so calling the Function both creates and runs the [tf.Graph](https://www.tensorflow.org/api_docs/python/tf/Graph). Later you will see how you can run only the tracing stage with [get\_concrete\_function](https://www.tensorflow.org/guide/function" \l "obtaining_concrete_functions).  11.When would you need to create a dynamic Keras model? How do you do that? Why not make all your models dynamic?  Keras is a neural network Application Programming Interface (API) for Python that is tightly integrated with TensorFlow, which is used to build machine learning models. Keras’ models offer a simple, user-friendly way to define a neural network, which will then be built for you by TensorFlow.  What’s the Difference Between Tensorflow and Keras?  TensorFlow is an open-source set of libraries for creating and working with neural networks, such as those used in Machine Learning (ML) and Deep Learning projects.  Keras, on the other hand, is a high-level API that runs on top of TensorFlow. Keras simplifies the implementation of complex neural networks with its easy to use framework.  When to Use Keras vs TensorFlow  TensorFlow provides a comprehensive machine learning platform that offers both high level and low level capabilities for building and deploying machine learning models. However, it does have a steep learning curve. It’s best used when you have a need for:  Deep learning research  Complex neural networks  Working with large datasets  High performance models  Keras, on the other hand, is perfect for those that do not have a strong background in Deep Learning, but still want to work with neural networks. Using Keras, you can build a neural network model quickly and easily using minimal code, allowing for rapid prototyping. For example:  # Import the Keras libraries required in this example:  from keras.models import Sequential  from keras.layers import Dense, Activation  # Create a Sequential model:  model = Sequential()  # Add layers with the add() method:  model.add(Dense(32, input\_dim=784))  model.add(Activation('relu'))  Keras is less error prone than TensorFlow, and models are more likely to be accurate with Keras than with TensorFlow. This is because Keras operates within the limitations of its framework, which include:  Computation speed: Keras sacrifices speed for user-friendliness.  Low-level Errors: sometimes you’ll get TensorFlow backend error messages that Keras was not designed to handle.  Algorithm Support – Keras is not well suited for working with certain basic machine learning algorithms and models like clustering and Principal Component Analysis (PCM).  Dynamic Charts – Keras has no support for dynamic chart creation.  Keras Model Overview  Models are the core entity you’ll be working with when using Keras. The models are used to define TensorFlow neural networks by specifying the attributes, functions, and layers you want.  Keras offers a number of APIs you can use to define your neural network, including:  Sequential API, which lets you create a model layer by layer for most problems. It’s straightforward (just a simple list of layers), but it’s limited to single-input, single-output stacks of layers.  Functional API, which is a full-featured API that supports arbitrary model architectures. It’s more flexible and complex than the sequential API.  Model Subclassing, which lets you implement everything from scratch. Suitable for research and highly complex use cases, but rarely used in practice.  How to Define a Neural Network with Keras’ Sequential API  The Sequential API is a framework for creating models based on instances of the sequential() class. The model has one input variable, a hidden layer with two neurons, and an output layer with one binary output. Additional layers can be created and added to the model.  # Define the model:  from keras.models import Sequential  from keras.layers import Dense  model = Sequential()  model.add(Dense(2, input\_dim=1, activation='relu'))  model.add(Dense(1, activation='sigmoid'))  The model includes the following information:  Layers and their order in the model.  Output shape (number of elements in each dimension of output data)  of each layer.  Number of parameters (weights) in each layer.  Total number of parameters in the model. | |