Colab - Workshop
"An introduction to Tensorflow"

Gerald Corzo

5/26/2020

Contents

Workshop Google Machine Learning tools (services)

Motivation

In general there is a world of alternatives for Machine Learning and is not so easy to follow one path. New libraries are appearing in the arena of ML, however, there is little or no legacy (compatibility) among them.

New learning algorithms are presented in environments with new ways to design model. This links also to ways to connect models and data, and the facilities present to develop on the cloud. However, time finding the "path" for solving a problem could be quite cumbersome and long.

So where can we start to learn? A suggestion is to follow these links.

Overview

This exercise is meant to cover a short description of the websites that provide data and Machine learning tools as well as to introduce some of the main principles of using this tools.

This workshop does not aim at describing the algorithms as this was already covered in the lectures.

As an introduction to this workshop it is recommended to explore the following tutorial from zero to Hero part 1.

https://www.youtube.com/watch?v=KNAWp2S3w94

** Tensorflow version** Tensorflow version 2.0 is relatively new, and links found on google could be pointing to the old version.

For this Colab-Workshop we will have:

Part 1: Overview of sites and introduction to tensorflow Part 2: Exercise on reading data Part 3: Creating a forecasting model using a ANN

Reference information

MIT has the following site with some good slides and links to examples:

https://cbmm.mit.edu/sites/default/files/documents/Tensorflow%202_0%20slides.pdf

https://machinelearningmastery.com/tensorflow-tutorial-deep-learning-with-tf-keras/

Google sites for ML

As part of the overview these sites will be described in the online session

Firebase - Mobile and cloud ML https://www.youtube.com/watch?v=ejrn_JHksws&feature=youtu.be&list=PLl-K7zZEsYLmOF_07IayrTntevxtbUxDL

Earth Engine https://earthengine.google.com/

Kaggle (Google) https://www.kaggle.com/

Google AI cloud https://cloud.google.com/ai-platform

Kubeflow (ML on Kubernets) https://www.kubeflow.org/

Google

https://ai.google/tools/

Courses

these links contains short courses https://developers.google.com/machine-learning/crash-course

https://aischool.microsoft.com/en-us

##Competition of Machine Learning (Water resources in Colombia)

Kaggle https://www.kaggle.com/ https://www.kaggle.com/corzogac/magdalena-colombia-data

Other sources of Machine Learning

Microsoft

https://www.microsoft.com/en-us/ai/ai-platform

https://dotnet.microsoft.com/apps/machinelearning-ai

https://azure.microsoft.com/en-us/overview/ai-platform/

Interesting blog to read https://medium.com/@Mareks_082/ml-net-machine-learning-library-from-microsoft-39d265761b34

Alternative and complements for tensorflow

https://developer.nvidia.com/tensorrt https://onnx.ai/ https://github.com/onnx

Part 1 : Tensorflow basics

This practice can be found in google colab following this link:

https://colab.research.google.com/drive/1d_scKNMkvPeHT_jtiBM7yzDIuD2EHG2r#scrollTo=K_uwKYPiXFIM

Tensorflow library

Tensorflow is a library that is meant to facilitate the operations that are common in processing large amounts of data. It was built on the concept of a tensor as a vector or array of multiple dimensions. Also, it can be seen as a type of mapping flows of tensors in different type of parallel processes, optimizing the use of the available infrastructures (most of the time proper for GPU type of operations). In this sense it is pretty fast, but in general algorithms have to be built on top of tensorflow. Keras is the most common estimators library that allows to use tensorflow types and integrates theirs processes into Machine Learning algorithms.

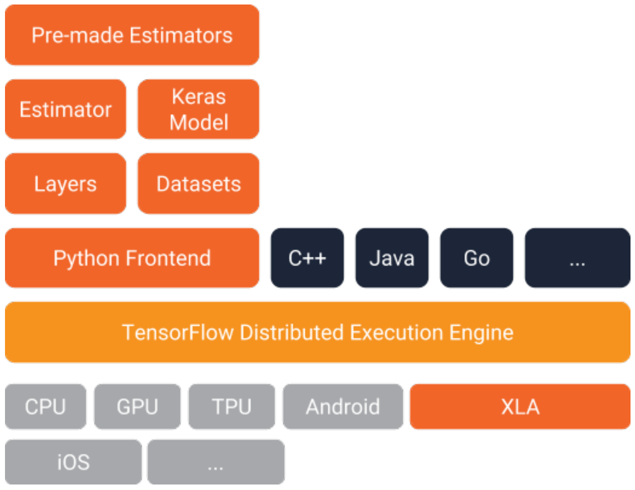


Figure 1: Tensorflow and estimatos in a compilation proces

The figure ...... shows the from bottom-up how the gray boxes which are representing the hardware infrastructure used to process the machine learning algorithms are the base of the task. The tensorflow infrastructure on top of it deal with a data and process such that can distribute in the "optimal" way the tasks.

CPU = Central processing unit **GPU** = Graphical processing unit **TPU** = Tensor processing unit **XLA** = eXtended Linear algebra units (integration of TPUs)

Installation

In a linux system this command will work well.

pip install tensorflow

In windows there are sometimes drawbacks associated to the hardware used and the required compiled packages.

Tensorflow variables

Tensorflow relates highly to numpy, and also to other libraries like google earth. The variable for tensor flow are created with the tensor flow function. Like the following example on how to create a constant.

```
import tensorflow as tf
# As an example of a constant
hello=tf.constant("My first tensor constant")
print(hello)
```

```
## tf.Tensor(b'My first tensor constant', shape=(), dtype=string)
```

The same is for numbers:

```
x=1
y=x+9
print("A normal variable x is "+str(x))
```

```
## A normal variable x is 1
```

```
print("A normal variable 9 is "+ str(y))

#Now with tensorflow the same as
```

```
## A normal variable 9 is 10
```

```
x=tf.constant(1,name='x')
y=tf.Variable(x+9,name='y')
print(y)
```

```
## <tf.Variable 'y:0' shape=() dtype=int32, numpy=10>
```

A tensor is sean as a vector so we can make the following analogy

a=1 (0 Dimension tensor) a=[1,3,5,6,7] (1 dimension tensor) a=[[1,2,3],[5,4,3]] (2 dimension tensor)

Part 2 : Machine Learning

Exercise 1 (Learning process)

Assume you have five input and output samples of an experiment. From a mathematical point of view you could represent the data from the experiment as a vector of the values.

Input

$x = [x_1, x_2, x_3, x_4, x_5]$

output

$y = [x_1, x_2, x_3, x_4, x_5]$

If you would like to know how the output relates to the input you could think about how to obtain a function such that $y = f(x)$. This could be solved with a standard mathematical analysis of the samples using a liner regression fit.

However, the if you do not know the way to do the linear regression, how would you learn the relation between the input vector x and the output vector y (firs in your mind).

Your task is : Guess the equation that relates x and y from the following example

Keras ANN

We will make a neural network that will help us to solve the above relation. A step by step description will follow

Import libraries

```
from tensorflow import keras
import numpy as np
import os
import datetime
import tensorflow as tf
#from tensorflow.keras.callbacks import TensorBoard
tf.__version__
```

```
## '2.2.0'
```

Define the variables

```
# Create the variables in numpy with size float to fit tensorflow requirements
xs=np.array([-1,0,1,2,3 ],dtype=float)
ys=np.array([-7,-2,3,8,13],dtype=float)
```

Build the model

Sequential = represents the sequential link between layers **Layers.Dense** = Represents a layer, and unit is the number of nodes in the layer.

```
#logdir=os.path.join("logs",datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
#tensorflow_callback=TensorBoard(log_dir=logdir,histogram_freq=1)

model=keras.Sequential(name="MyfirstModel")

model.add(keras.layers.Dense(units=1,input_shape=[1],name="MyLayer"))

model.compile(optimizer='sgd',loss='mean_squared_error')

model.fit(xs,ys,epochs=50) #,callbacks=[tensorflow_callback])
```

Making a prediction

Now after you trained your model

```
#the prediction is done by
p=model.predict([10])
print("The prediction of 10 is " +str(p))
print("The mathematical solution of 10 is 48")
```

The result should show something like this

The prediction of 10 is [[46.09436]] The mathematical solution of 10 is 48

Model visualization

To view the contents of your model you can use **get__weights** and **get__config**

```
m=model.get_weights()
print(m)
```

[array([[4.7488227]], dtype=float32), array([-1.393868], dtype=float32)]

```
model.get_config()

#result will be:
{'layers': [{'class_name': 'Dense',
   'config': {'activation': 'linear',
    'activity_regularizer': None,
    'batch_input_shape': (None, 1),
    'bias_constraint': None,
    'bias_initializer': {'class_name': 'Zeros', 'config': {}},
    'bias_regularizer': None,
    'dtype': 'float32',
    'kernel_constraint': None,
    'kernel_initializer': {'class_name': 'VarianceScaling',
     'config': {'distribution': 'uniform',
      'mode': 'fan_avg',
      'scale': 1.0,
      'seed': None}},
    'kernel_regularizer': None,
    'name': 'MyLayer',
    'trainable': True,
    'units': 1,
    'use_bias': True}}],
 'name': 'MyfirstModel'}
```

Visualize your model structure

This can be done by sing the function plot_model inside the utilities section of keras library, as follows:

```
tf.keras.utils.plot_model(
    model,
    to_file="model.png",
    show_shapes=True,
    show_layer_names=True,
    rankdir="LR",
    expand_nested=True,
    dpi=96,
)
```

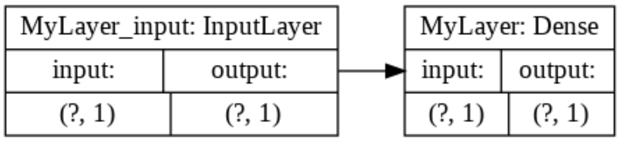
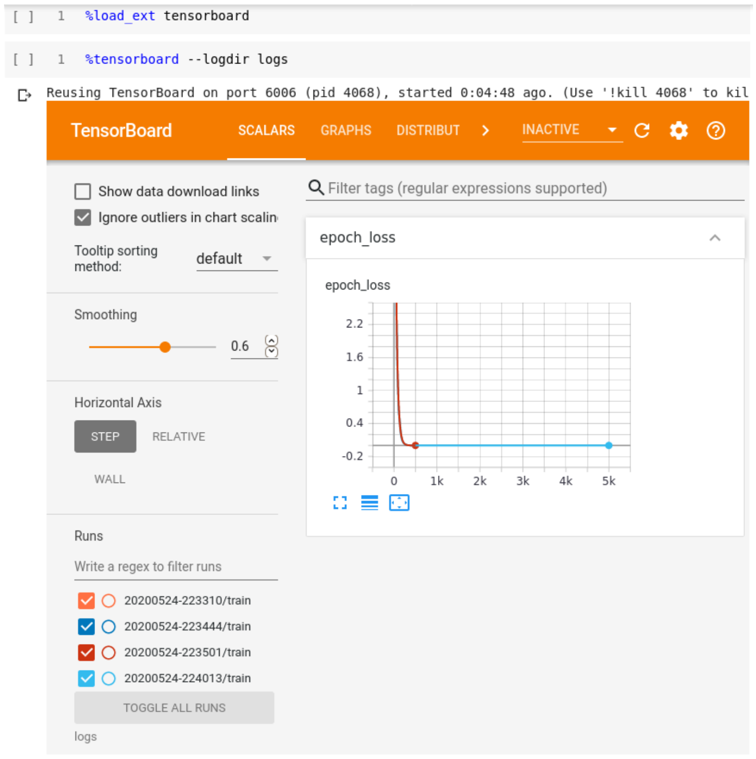


Figure 2: Model with one layer and one neuron, for one output

A more visual analysis of the model structure an its results can be obtained using the tensorboard.

```
%load_ext tensorboard
%tensorboard --logdir logs
```



```
[ ]  %load_ext tensorboard
[ ]  %tensorboard --logdir logs
Reusing TensorBoard on port 6006 (pid 4068), started 0:04:48 ago. (Use '!kill 4068' to kil
```

Figure 3: Tensorboard visualization of the previous runs in the log file

Keras dense layer

A way to understand mode the keras.layer can be done by comparing the matric multiplication done in numpy and the one in keras. See the following examples an try to analyze it in tensorflow.

matrix multiplication in numpy

```
x = np.arange(10).reshape(1, 5, 2)
print(x)

y = np.arange(10, 20).reshape(1, 2, 5)
print(y)

print("Matrix Multiplication")
z=np.matmul(y,x)
print(z)
```

matrix multiplication in tensorflow

```
x = np.arange(10).reshape(1, 5, 2)
print(x)

y = np.arange(10, 20).reshape(1, 2, 5)
print(y)

tf.keras.layers.Dot(axes=(1, 2))([x, y])
```

Exercise 2 (Build your ANN)

How many epochs do you need to build an ANN that replicate a sine function with 2 neurons.

1. Generate an input vector x of values from 0 to 6 with 0.1 intervals
2. Estimate the $y = sin(x)$ function for all the values of the input vector x.
3. Build an ANN model that estimates y

Run the same example with only one neuron and discuss the result.

Appendix

Help on a function for Dense layer

To obtain help on a function you can

```
help(tf.keras.layers.Dense)
```

```
## Help on class Dense in module tensorflow.python.keras.layers.core:
##
## class Dense(tensorflow.python.keras.engine.base_layer.Layer)
##  |  Just your regular densely-connected NN layer.
##  |
##  |  `Dense` implements the operation:
##  |  `output = activation(dot(input, kernel) + bias)`
##  |  where `activation` is the element-wise activation function
##  |  passed as the `activation` argument, `kernel` is a weights matrix
##  |  created by the layer, and `bias` is a bias vector created by the layer
## |  (only applicable if `use_bias` is `True`).
## |
## |  Note: If the input to the layer has a rank greater than 2, then `Dense`
## |  computes the dot product between the `inputs` and the `kernel` along the
## |  last axis of the `inputs` and axis 1 of the `kernel` (using `tf.tensordot`).
## |  For example, if input has dimensions `(batch_size, d0, d1)`,
## |  then we create a `kernel` with shape `(d1, units)`, and the `kernel` operates
## |  along axis 2 of the `input`, on every sub-tensor of shape `(1, 1, d1)`
## |  (there are `batch_size * d0` such sub-tensors).
## |  The output in this case will have shape `(batch_size, d0, units)`.
## |
## |  Besides, layer attributes cannot be modified after the layer has been called
## |  once (except the `trainable` attribute).
## |
## |  Example:
## |
## |  ```python
## |  # as first layer in a sequential model:
## |  model = Sequential()
## |  model.add(Dense(32, input_shape=(16,)))
## |  # now the model will take as input arrays of shape (*, 16)
## |  # and output arrays of shape (*, 32)
## |
## |  # after the first layer, you don't need to specify
## |  # the size of the input anymore:
## |  model.add(Dense(32))
## |  ```
## |
## |  Arguments:
## |    units: Positive integer, dimensionality of the output space.
## |    activation: Activation function to use.
## |      If you don't specify anything, no activation is applied
## |      (ie. "linear" activation: `a(x) = x`).
## |    use_bias: Boolean, whether the layer uses a bias vector.
## |    kernel_initializer: Initializer for the `kernel` weights matrix.
## |    bias_initializer: Initializer for the bias vector.
## |    kernel_regularizer: Regularizer function applied to
## |      the `kernel` weights matrix.
## |    bias_regularizer: Regularizer function applied to the bias vector.
## |    activity_regularizer: Regularizer function applied to
## |      the output of the layer (its "activation")..
## |    kernel_constraint: Constraint function applied to
## |      the `kernel` weights matrix.
## |    bias_constraint: Constraint function applied to the bias vector.
## |
## |  Input shape:
## |    N-D tensor with shape: `(batch_size, ..., input_dim)`.
## |    The most common situation would be
## |    a 2D input with shape `(batch_size, input_dim)`.
## |
## |  Output shape:
## |    N-D tensor with shape: `(batch_size, ..., units)`.
## |    For instance, for a 2D input with shape `(batch_size, input_dim)`,
## |    the output would have shape `(batch_size, units)`.
```

```
## |
## | Method resolution order:
## |     Dense
## |     tensorflow.python.keras.engine.base_layer.Layer
## |     tensorflow.python.module.module.Module
## |     tensorflow.python.training.tracking.tracking.AutoTrackable
## |     tensorflow.python.training.tracking.base.Trackable
## |     tensorflow.python.keras.utils.version_utils.LayerVersionSelector
## |     builtins.object
## |
## | Methods defined here:
## |
## | __init__(self, units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_
## |
## | build(self, input_shape)
## |     Creates the variables of the layer (optional, for subclass implementers).
## |
## |     This is a method that implementers of subclasses of `Layer` or `Model`
## |     can override if they need a state-creation step in-between
## |     layer instantiation and layer call.
## |
## |     This is typically used to create the weights of `Layer` subclasses.
## |
## |     Arguments:
## |       input_shape: Instance of `TensorShape`, or list of instances of
## |         `TensorShape` if the layer expects a list of inputs
## |         (one instance per input).
## |
## | call(self, inputs)
## |     This is where the layer's logic lives.
## |
## |     Arguments:
## |         inputs: Input tensor, or list/tuple of input tensors.
## |         **kwargs: Additional keyword arguments.
## |
## |     Returns:
## |         A tensor or list/tuple of tensors.
## |
## | compute_output_shape(self, input_shape)
## |     Computes the output shape of the layer.
## |
## |     If the layer has not been built, this method will call `build` on the
## |     layer. This assumes that the layer will later be used with inputs that
## |     match the input shape provided here.
## |
## |     Arguments:
## |         input_shape: Shape tuple (tuple of integers)
## |             or list of shape tuples (one per output tensor of the layer).
## |             Shape tuples can include None for free dimensions,
## |             instead of an integer.
## |
## |     Returns:
## |         An input shape tuple.
## |
## |  get_config(self)
## |      Returns the config of the layer.
## |
## |      A layer config is a Python dictionary (serializable)
## |      containing the configuration of a layer.
## |      The same layer can be reinstantiated later
## |      (without its trained weights) from this configuration.
## |
## |      The config of a layer does not include connectivity
## |      information, nor the layer class name. These are handled
## |      by `Network` (one layer of abstraction above).
## |
## |      Returns:
## |          Python dictionary.
## |
## |  ----------------------------------------------------------------------
## |  Methods inherited from tensorflow.python.keras.engine.base_layer.Layer:
## |
## |  __call__(self, *args, **kwargs)
## |      Wraps `call`, applying pre- and post-processing steps.
## |
## |      Arguments:
## |        *args: Positional arguments to be passed to `self.call`.
## |        **kwargs: Keyword arguments to be passed to `self.call`.
## |
## |      Returns:
## |        Output tensor(s).
## |
## |      Note:
## |        - The following optional keyword arguments are reserved for specific uses:
## |          * `training`: Boolean scalar tensor of Python boolean indicating
## |            whether the `call` is meant for training or inference.
## |          * `mask`: Boolean input mask.
## |        - If the layer's `call` method takes a `mask` argument (as some Keras
## |          layers do), its default value will be set to the mask generated
## |          for `inputs` by the previous layer (if `input` did come from
## |          a layer that generated a corresponding mask, i.e. if it came from
## |          a Keras layer with masking support.
## |
## |      Raises:
## |        ValueError: if the layer's `call` method returns None (an invalid value).
## |        RuntimeError: if `super().__init__()` was not called in the constructor.
## |
## |  __delattr__(self, name)
## |      Implement delattr(self, name).
## |
## |  __getstate__(self)
## |
## |  __setattr__(self, name, value)
## |      Support self.foo = trackable syntax.
## |
## |  __setstate__(self, state)
## |
## |  add_loss(self, losses, inputs=None)
## |      Add loss tensor(s), potentially dependent on layer inputs.
## |
## |      Some losses (for instance, activity regularization losses) may be dependent
## |      on the inputs passed when calling a layer. Hence, when reusing the same
## |      layer on different inputs `a` and `b`, some entries in `layer.losses` may
## |      be dependent on `a` and some on `b`. This method automatically keeps track
## |      of dependencies.
## |
## |      This method can be used inside a subclassed layer or model's `call`
## |      function, in which case `losses` should be a Tensor or list of Tensors.
## |
## |      Example:
## |
## |      ```python
## |      class MyLayer(tf.keras.layers.Layer):
## |        def call(inputs, self):
## |          self.add_loss(tf.abs(tf.reduce_mean(inputs)), inputs=True)
## |          return inputs
## |      ```
## |
## |      This method can also be called directly on a Functional Model during
## |      construction. In this case, any loss Tensors passed to this Model must
## |      be symbolic and be able to be traced back to the model's `Input`s. These
## |      losses become part of the model's topology and are tracked in `get_config`.
## |
## |      Example:
## |
## |      ```python
## |      inputs = tf.keras.Input(shape=(10,))
## |      x = tf.keras.layers.Dense(10)(inputs)
## |      outputs = tf.keras.layers.Dense(1)(x)
## |      model = tf.keras.Model(inputs, outputs)
## |      # Activity regularization.
## |      model.add_loss(tf.abs(tf.reduce_mean(x)))
## |      ```
## |
## |      If this is not the case for your loss (if, for example, your loss references
## |      a `Variable` of one of the model's layers), you can wrap your loss in a
## |      zero-argument lambda. These losses are not tracked as part of the model's
## |      topology since they can't be serialized.
## |
## |      Example:
## |
## |      ```python
## |      inputs = tf.keras.Input(shape=(10,))
## |      x = tf.keras.layers.Dense(10)(inputs)
## |      outputs = tf.keras.layers.Dense(1)(x)
## |      model = tf.keras.Model(inputs, outputs)
## |      # Weight regularization.
## |      model.add_loss(lambda: tf.reduce_mean(x.kernel))
## |      ```
## |
## |      The `get_losses_for` method allows to retrieve the losses relevant to a
## |      specific set of inputs.
```

```
## |
## |      Arguments:
## |        losses: Loss tensor, or list/tuple of tensors. Rather than tensors, losses
## |          may also be zero-argument callables which create a loss tensor.
## |        inputs: Ignored when executing eagerly. If anything other than None is
## |          passed, it signals the losses are conditional on some of the layer's
## |          inputs, and thus they should only be run where these inputs are
## |          available. This is the case for activity regularization losses, for
## |          instance. If `None` is passed, the losses are assumed
## |          to be unconditional, and will apply across all dataflows of the layer
## |          (e.g. weight regularization losses).
## |
## |  add_metric(self, value, aggregation=None, name=None)
## |      Adds metric tensor to the layer.
## |
## |      Args:
## |        value: Metric tensor.
## |        aggregation: Sample-wise metric reduction function. If `aggregation=None`,
## |          it indicates that the metric tensor provided has been aggregated
## |          already. eg, `bin_acc = BinaryAccuracy(name='acc')` followed by
## |          `model.add_metric(bin_acc(y_true, y_pred))`. If aggregation='mean', the
## |          given metric tensor will be sample-wise reduced using `mean` function.
## |          eg, `model.add_metric(tf.reduce_sum(outputs), name='output_mean',
## |          aggregation='mean')`.
## |        name: String metric name.
## |
## |      Raises:
## |        ValueError: If `aggregation` is anything other than None or `mean`.
## |
## |  add_update(self, updates, inputs=None)
## |      Add update op(s), potentially dependent on layer inputs. (deprecated arguments)
## |
## |      Warning: SOME ARGUMENTS ARE DEPRECATED: `(inputs)`. They will be removed in a future version
## |      Instructions for updating:
## |      `inputs` is now automatically inferred
## |
## |      Weight updates (for instance, the updates of the moving mean and variance
## |      in a BatchNormalization layer) may be dependent on the inputs passed
## |      when calling a layer. Hence, when reusing the same layer on
## |      different inputs `a` and `b`, some entries in `layer.updates` may be
## |      dependent on `a` and some on `b`. This method automatically keeps track
## |      of dependencies.
## |
## |      The `get_updates_for` method allows to retrieve the updates relevant to a
## |      specific set of inputs.
## |
## |      This call is ignored when eager execution is enabled (in that case, variable
## |      updates are run on the fly and thus do not need to be tracked for later
## |      execution).
## |
## |      Arguments:
## |        updates: Update op, or list/tuple of update ops, or zero-arg callable
## |          that returns an update op. A zero-arg callable should be passed in
## |          order to disable running the updates by setting `trainable=False`
##  |          on this Layer, when executing in Eager mode.
##  |        inputs: Deprecated, will be automatically inferred.
##  |
##  |  add_variable(self, *args, **kwargs)
##  |      Deprecated, do NOT use! Alias for `add_weight`. (deprecated)
##  |
##  |      Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version.
##  |      Instructions for updating:
##  |      Please use `layer.add_weight` method instead.
##  |
##  |  add_weight(self, name=None, shape=None, dtype=None, initializer=None, regularizer=None, trainabl
##  |      Adds a new variable to the layer.
##  |
##  |      Arguments:
##  |        name: Variable name.
##  |        shape: Variable shape. Defaults to scalar if unspecified.
##  |        dtype: The type of the variable. Defaults to `self.dtype` or `float32`.
##  |        initializer: Initializer instance (callable).
##  |        regularizer: Regularizer instance (callable).
##  |        trainable: Boolean, whether the variable should be part of the layer's
##  |          "trainable_variables" (e.g. variables, biases)
##  |          or "non_trainable_variables" (e.g. BatchNorm mean and variance).
##  |          Note that `trainable` cannot be `True` if `synchronization`
##  |          is set to `ON_READ`.
##  |        constraint: Constraint instance (callable).
##  |        partitioner: Partitioner to be passed to the `Trackable` API.
##  |        use_resource: Whether to use `ResourceVariable`.
##  |        synchronization: Indicates when a distributed a variable will be
##  |          aggregated. Accepted values are constants defined in the class
##  |          `tf.VariableSynchronization`. By default the synchronization is set to
##  |          `AUTO` and the current `DistributionStrategy` chooses
##  |          when to synchronize. If `synchronization` is set to `ON_READ`,
##  |          `trainable` must not be set to `True`.
##  |        aggregation: Indicates how a distributed variable will be aggregated.
##  |          Accepted values are constants defined in the class
##  |          `tf.VariableAggregation`.
##  |        **kwargs: Additional keyword arguments. Accepted values are `getter`,
##  |          `collections`, `experimental_autocast` and `caching_device`.
##  |
##  |      Returns:
##  |        The created variable. Usually either a `Variable` or `ResourceVariable`
##  |        instance. If `partitioner` is not `None`, a `PartitionedVariable`
##  |        instance is returned.
##  |
##  |      Raises:
##  |        RuntimeError: If called with partitioned variable regularization and
##  |          eager execution is enabled.
##  |        ValueError: When giving unsupported dtype and no initializer or when
##  |          trainable has been set to True with synchronization set as `ON_READ`.
##  |
##  |  apply(self, inputs, *args, **kwargs)
##  |      Deprecated, do NOT use! (deprecated)
##  |
##  |      Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version.
## |      Instructions for updating:
## |      Please use `layer.__call__` method instead.
## |
## |      This is an alias of `self.__call__`.
## |
## |      Arguments:
## |        inputs: Input tensor(s).
## |        *args: additional positional arguments to be passed to `self.call`.
## |        **kwargs: additional keyword arguments to be passed to `self.call`.
## |
## |      Returns:
## |        Output tensor(s).
## |
## |  compute_mask(self, inputs, mask=None)
## |      Computes an output mask tensor.
## |
## |      Arguments:
## |          inputs: Tensor or list of tensors.
## |          mask: Tensor or list of tensors.
## |
## |      Returns:
## |          None or a tensor (or list of tensors,
## |              one per output tensor of the layer).
## |
## |  compute_output_signature(self, input_signature)
## |      Compute the output tensor signature of the layer based on the inputs.
## |
## |      Unlike a TensorShape object, a TensorSpec object contains both shape
## |      and dtype information for a tensor. This method allows layers to provide
## |      output dtype information if it is different from the input dtype.
## |      For any layer that doesn't implement this function,
## |      the framework will fall back to use `compute_output_shape`, and will
## |      assume that the output dtype matches the input dtype.
## |
## |      Args:
## |        input_signature: Single TensorSpec or nested structure of TensorSpec
## |          objects, describing a candidate input for the layer.
## |
## |      Returns:
## |        Single TensorSpec or nested structure of TensorSpec objects, describing
## |          how the layer would transform the provided input.
## |
## |      Raises:
## |        TypeError: If input_signature contains a non-TensorSpec object.
## |
## |  count_params(self)
## |      Count the total number of scalars composing the weights.
## |
## |      Returns:
## |          An integer count.
## |
## |      Raises:
## |          ValueError: if the layer isn't yet built
## |            (in which case its weights aren't yet defined).
## |
## | get_input_at(self, node_index)
## |     Retrieves the input tensor(s) of a layer at a given node.
## |
## |     Arguments:
## |         node_index: Integer, index of the node
## |             from which to retrieve the attribute.
## |             E.g. `node_index=0` will correspond to the
## |             first time the layer was called.
## |
## |     Returns:
## |         A tensor (or list of tensors if the layer has multiple inputs).
## |
## |     Raises:
## |       RuntimeError: If called in Eager mode.
## |
## | get_input_mask_at(self, node_index)
## |     Retrieves the input mask tensor(s) of a layer at a given node.
## |
## |     Arguments:
## |         node_index: Integer, index of the node
## |             from which to retrieve the attribute.
## |             E.g. `node_index=0` will correspond to the
## |             first time the layer was called.
## |
## |     Returns:
## |         A mask tensor
## |         (or list of tensors if the layer has multiple inputs).
## |
## | get_input_shape_at(self, node_index)
## |     Retrieves the input shape(s) of a layer at a given node.
## |
## |     Arguments:
## |         node_index: Integer, index of the node
## |             from which to retrieve the attribute.
## |             E.g. `node_index=0` will correspond to the
## |             first time the layer was called.
## |
## |     Returns:
## |         A shape tuple
## |         (or list of shape tuples if the layer has multiple inputs).
## |
## |     Raises:
## |       RuntimeError: If called in Eager mode.
## |
## | get_losses_for(self, inputs)
## |     Retrieves losses relevant to a specific set of inputs.
## |
## |     Arguments:
## |       inputs: Input tensor or list/tuple of input tensors.
## |
## |     Returns:
## |       List of loss tensors of the layer that depend on `inputs`.
## |
##  |  get_output_at(self, node_index)
##  |      Retrieves the output tensor(s) of a layer at a given node.
##  |
##  |      Arguments:
##  |          node_index: Integer, index of the node
##  |              from which to retrieve the attribute.
##  |              E.g. `node_index=0` will correspond to the
##  |              first time the layer was called.
##  |
##  |      Returns:
##  |          A tensor (or list of tensors if the layer has multiple outputs).
##  |
##  |      Raises:
##  |        RuntimeError: If called in Eager mode.
##  |
##  |  get_output_mask_at(self, node_index)
##  |      Retrieves the output mask tensor(s) of a layer at a given node.
##  |
##  |      Arguments:
##  |          node_index: Integer, index of the node
##  |              from which to retrieve the attribute.
##  |              E.g. `node_index=0` will correspond to the
##  |              first time the layer was called.
##  |
##  |      Returns:
##  |          A mask tensor
##  |          (or list of tensors if the layer has multiple outputs).
##  |
##  |  get_output_shape_at(self, node_index)
##  |      Retrieves the output shape(s) of a layer at a given node.
##  |
##  |      Arguments:
##  |          node_index: Integer, index of the node
##  |              from which to retrieve the attribute.
##  |              E.g. `node_index=0` will correspond to the
##  |              first time the layer was called.
##  |
##  |      Returns:
##  |          A shape tuple
##  |          (or list of shape tuples if the layer has multiple outputs).
##  |
##  |      Raises:
##  |        RuntimeError: If called in Eager mode.
##  |
##  |  get_updates_for(self, inputs)
##  |      Retrieves updates relevant to a specific set of inputs.
##  |
##  |      Arguments:
##  |        inputs: Input tensor or list/tuple of input tensors.
##  |
##  |      Returns:
##  |        List of update ops of the layer that depend on `inputs`.
##  |
##  |  get_weights(self)
## |      Returns the current weights of the layer.
## |
## |      The weights of a layer represent the state of the layer. This function
## |      returns both trainable and non-trainable weight values associated with this
## |      layer as a list of Numpy arrays, which can in turn be used to load state
## |      into similarly parameterized layers.
## |
## |      For example, a Dense layer returns a list of two values-- per-output
## |      weights and the bias value. These can be used to set the weights of another
## |      Dense layer:
## |
## |      >>> a = tf.keras.layers.Dense(1,
## |      ...   kernel_initializer=tf.constant_initializer(1.))
## |      >>> a_out = a(tf.convert_to_tensor([[1., 2., 3.]]))
## |      >>> a.get_weights()
## |      [array([[1.],
## |             [1.],
## |             [1.]], dtype=float32), array([0.], dtype=float32)]
## |      >>> b = tf.keras.layers.Dense(1,
## |      ...   kernel_initializer=tf.constant_initializer(2.))
## |      >>> b_out = b(tf.convert_to_tensor([[10., 20., 30.]]))
## |      >>> b.get_weights()
## |      [array([[2.],
## |             [2.],
## |             [2.]], dtype=float32), array([0.], dtype=float32)]
## |      >>> b.set_weights(a.get_weights())
## |      >>> b.get_weights()
## |      [array([[1.],
## |             [1.],
## |             [1.]], dtype=float32), array([0.], dtype=float32)]
## |
## |      Returns:
## |          Weights values as a list of numpy arrays.
## |
## |  set_weights(self, weights)
## |      Sets the weights of the layer, from Numpy arrays.
## |
## |      The weights of a layer represent the state of the layer. This function
## |      sets the weight values from numpy arrays. The weight values should be
## |      passed in the order they are created by the layer. Note that the layer's
## |      weights must be instantiated before calling this function by calling
## |      the layer.
## |
## |      For example, a Dense layer returns a list of two values-- per-output
## |      weights and the bias value. These can be used to set the weights of another
## |      Dense layer:
## |
## |      >>> a = tf.keras.layers.Dense(1,
## |      ...   kernel_initializer=tf.constant_initializer(1.))
## |      >>> a_out = a(tf.convert_to_tensor([[1., 2., 3.]]))
## |      >>> a.get_weights()
## |      [array([[1.],
## |             [1.],
## |             [1.]], dtype=float32), array([0.], dtype=float32)]
##  |      >>> b = tf.keras.layers.Dense(1,
##  |      ...   kernel_initializer=tf.constant_initializer(2.))
##  |      >>> b_out = b(tf.convert_to_tensor([[10., 20., 30.]]))
##  |      >>> b.get_weights()
##  |      [array([[2.],
##  |             [2.],
##  |             [2.]], dtype=float32), array([0.], dtype=float32)]
##  |      >>> b.set_weights(a.get_weights())
##  |      >>> b.get_weights()
##  |      [array([[1.],
##  |             [1.],
##  |             [1.]], dtype=float32), array([0.], dtype=float32)]
##  |
##  |      Arguments:
##  |          weights: a list of Numpy arrays. The number
##  |              of arrays and their shape must match
##  |              number of the dimensions of the weights
##  |              of the layer (i.e. it should match the
##  |              output of `get_weights`).
##  |
##  |      Raises:
##  |          ValueError: If the provided weights list does not match the
##  |              layer's specifications.
##  |
##  |  ----------------------------------------------------------------------
##  |  Class methods inherited from tensorflow.python.keras.engine.base_layer.Layer:
##  |
##  |  from_config(config) from builtins.type
##  |      Creates a layer from its config.
##  |
##  |      This method is the reverse of `get_config`,
##  |      capable of instantiating the same layer from the config
##  |      dictionary. It does not handle layer connectivity
##  |      (handled by Network), nor weights (handled by `set_weights`).
##  |
##  |      Arguments:
##  |          config: A Python dictionary, typically the
##  |              output of get_config.
##  |
##  |      Returns:
##  |          A layer instance.
##  |
##  |  ----------------------------------------------------------------------
##  |  Data descriptors inherited from tensorflow.python.keras.engine.base_layer.Layer:
##  |
##  |  activity_regularizer
##  |      Optional regularizer function for the output of this layer.
##  |
##  |  dtype
##  |      Dtype used by the weights of the layer, set in the constructor.
##  |
##  |  dynamic
##  |      Whether the layer is dynamic (eager-only); set in the constructor.
##  |
## |  inbound_nodes
## |      Deprecated, do NOT use! Only for compatibility with external Keras.
## |
## |  input
## |      Retrieves the input tensor(s) of a layer.
## |
## |      Only applicable if the layer has exactly one input,
## |      i.e. if it is connected to one incoming layer.
## |
## |      Returns:
## |          Input tensor or list of input tensors.
## |
## |      Raises:
## |        RuntimeError: If called in Eager mode.
## |        AttributeError: If no inbound nodes are found.
## |
## |  input_mask
## |      Retrieves the input mask tensor(s) of a layer.
## |
## |      Only applicable if the layer has exactly one inbound node,
## |      i.e. if it is connected to one incoming layer.
## |
## |      Returns:
## |          Input mask tensor (potentially None) or list of input
## |          mask tensors.
## |
## |      Raises:
## |          AttributeError: if the layer is connected to
## |          more than one incoming layers.
## |
## |  input_shape
## |      Retrieves the input shape(s) of a layer.
## |
## |      Only applicable if the layer has exactly one input,
## |      i.e. if it is connected to one incoming layer, or if all inputs
## |      have the same shape.
## |
## |      Returns:
## |          Input shape, as an integer shape tuple
## |          (or list of shape tuples, one tuple per input tensor).
## |
## |      Raises:
## |          AttributeError: if the layer has no defined input_shape.
## |          RuntimeError: if called in Eager mode.
## |
## |  input_spec
## |      `InputSpec` instance(s) describing the input format for this layer.
## |
## |      When you create a layer subclass, you can set `self.input_spec` to enable
## |      the layer to run input compatibility checks when it is called.
## |      Consider a `Conv2D` layer: it can only be called on a single input tensor
## |      of rank 4. As such, you can set, in `__init__()`:
## |
## |      ```python
```

```
## |      self.input_spec = tf.keras.layers.InputSpec(ndim=4)
## |      ```
## |
## |      Now, if you try to call the layer on an input that isn't rank 4
## |      (for instance, an input of shape `(2,)`, it will raise a nicely-formatted
## |      error:
## |
## |      ```
## |      ValueError: Input 0 of layer conv2d is incompatible with the layer:
## |      expected ndim=4, found ndim=1. Full shape received: [2]
## |      ```
## |
## |      Input checks that can be specified via `input_spec` include:
## |      - Structure (e.g. a single input, a list of 2 inputs, etc)
## |      - Shape
## |      - Rank (ndim)
## |      - Dtype
## |
## |      For more information, see `tf.keras.layers.InputSpec`.
## |
## |      Returns:
## |        A `tf.keras.layers.InputSpec` instance, or nested structure thereof.
## |
## |  losses
## |      Losses which are associated with this `Layer`.
## |
## |      Variable regularization tensors are created when this property is accessed,
## |      so it is eager safe: accessing `losses` under a `tf.GradientTape` will
## |      propagate gradients back to the corresponding variables.
## |
## |      Returns:
## |        A list of tensors.
## |
## |  metrics
## |      List of `tf.keras.metrics.Metric` instances tracked by the layer.
## |
## |  name
## |      Name of the layer (string), set in the constructor.
## |
## |  non_trainable_variables
## |
## |  non_trainable_weights
## |      List of all non-trainable weights tracked by this layer.
## |
## |      Non-trainable weights are *not* updated during training. They are expected
## |      to be updated manually in `call()`.
## |
## |      Returns:
## |        A list of non-trainable variables.
## |
## |  outbound_nodes
## |      Deprecated, do NOT use! Only for compatibility with external Keras.
## |
## |  output
```

```
## |      Retrieves the output tensor(s) of a layer.
## |
## |      Only applicable if the layer has exactly one output,
## |      i.e. if it is connected to one incoming layer.
## |
## |      Returns:
## |        Output tensor or list of output tensors.
## |
## |      Raises:
## |        AttributeError: if the layer is connected to more than one incoming
## |          layers.
## |        RuntimeError: if called in Eager mode.
## |
## |  output_mask
## |      Retrieves the output mask tensor(s) of a layer.
## |
## |      Only applicable if the layer has exactly one inbound node,
## |      i.e. if it is connected to one incoming layer.
## |
## |      Returns:
## |          Output mask tensor (potentially None) or list of output
## |          mask tensors.
## |
## |      Raises:
## |          AttributeError: if the layer is connected to
## |          more than one incoming layers.
## |
## |  output_shape
## |      Retrieves the output shape(s) of a layer.
## |
## |      Only applicable if the layer has one output,
## |      or if all outputs have the same shape.
## |
## |      Returns:
## |          Output shape, as an integer shape tuple
## |          (or list of shape tuples, one tuple per output tensor).
## |
## |      Raises:
## |          AttributeError: if the layer has no defined output shape.
## |          RuntimeError: if called in Eager mode.
## |
## |  stateful
## |
## |  trainable
## |
## |  trainable_variables
## |      Sequence of trainable variables owned by this module and its submodules.
## |
## |      Note: this method uses reflection to find variables on the current instance
## |      and submodules. For performance reasons you may wish to cache the result
## |      of calling this method if you don't expect the return value to change.
## |
## |      Returns:
## |        A sequence of variables for the current module (sorted by attribute
## |        name) followed by variables from all submodules recursively (breadth
## |        first).
## |
## |  trainable_weights
## |      List of all trainable weights tracked by this layer.
## |
## |      Trainable weights are updated via gradient descent during training.
## |
## |      Returns:
## |        A list of trainable variables.
## |
## |  updates
## |
## |  variables
## |      Returns the list of all layer variables/weights.
## |
## |      Alias of `self.weights`.
## |
## |      Returns:
## |        A list of variables.
## |
## |  weights
## |      Returns the list of all layer variables/weights.
## |
## |      Returns:
## |        A list of variables.
## |
## |  ----------------------------------------------------------------------
## |  Class methods inherited from tensorflow.python.module.module.Module:
## |
## |  with_name_scope(method) from builtins.type
## |      Decorator to automatically enter the module name scope.
## |
## |      >>> class MyModule(tf.Module):
## |      ...   @tf.Module.with_name_scope
## |      ...   def __call__(self, x):
## |      ...     if not hasattr(self, 'w'):
## |      ...       self.w = tf.Variable(tf.random.normal([x.shape[1], 3]))
## |      ...     return tf.matmul(x, self.w)
## |
## |      Using the above module would produce `tf.Variable`s and `tf.Tensor`s whose
## |      names included the module name:
## |
## |      >>> mod = MyModule()
## |      >>> mod(tf.ones([1, 2]))
## |      <tf.Tensor: shape=(1, 3), dtype=float32, numpy=..., dtype=float32)>
## |      >>> mod.w
## |      <tf.Variable 'my_module/Variable:0' shape=(2, 3) dtype=float32,
## |      numpy=..., dtype=float32)>
## |
## |      Args:
## |        method: The method to wrap.
## |
## |      Returns:
```

```
## |       The original method wrapped such that it enters the module's name scope.
## |
## | ----------------------------------------------------------------------
## | Data descriptors inherited from tensorflow.python.module.module.Module:
## |
## | name_scope
## |     Returns a `tf.name_scope` instance for this class.
## |
## | submodules
## |     Sequence of all sub-modules.
## |
## |     Submodules are modules which are properties of this module, or found as
## |     properties of modules which are properties of this module (and so on).
## |
## |     >>> a = tf.Module()
## |     >>> b = tf.Module()
## |     >>> c = tf.Module()
## |     >>> a.b = b
## |     >>> b.c = c
## |     >>> list(a.submodules) == [b, c]
## |     True
## |     >>> list(b.submodules) == [c]
## |     True
## |     >>> list(c.submodules) == []
## |     True
## |
## |     Returns:
## |       A sequence of all submodules.
## |
## | ----------------------------------------------------------------------
## | Data descriptors inherited from tensorflow.python.training.tracking.base.Trackable:
## |
## | __dict__
## |     dictionary for instance variables (if defined)
## |
## | __weakref__
## |     list of weak references to the object (if defined)
## |
## | ----------------------------------------------------------------------
## | Static methods inherited from tensorflow.python.keras.utils.version_utils.LayerVersionSelector:
## |
## | __new__(cls, *args, **kwargs)
## |     Create and return a new object.  See help(type) for accurate signature.
```