string_data_bicubic

December 17, 2021

1 String data

In this notebook, we will explore a hands-on example of learning a Ricci-flat metric using the new **cymetric** library.

github.com/pythoncymetric/cymetric

You can install it directly from your colab or jupyter environment with

[1]:
```
!pip install -q git+https://github.com/pythoncymetric/cymetric.git
```

```
    |                 | 6.2 MB 8.4 MB/s
    |                 | 837 kB 55.8 MB/s
    |                 | 1.1 MB 57.6 MB/s
    |                 | 160 kB 72.4 MB/s
    |                 | 271 kB 71.8 MB/s
    |                 | 192 kB 73.7 MB/s
  Building wheel for cymetric (setup.py) … done
```

The package currently consists of two modules with various submodules,

1. one for the generation of points based on a *PointGenerator* class. Check out this notebook for all the different sub-modules.
2. one for the learning of Ricci-flat metrics, based on corrections to the Fubini-Study metric. Check out this notebook for some of the different sub-modules.

There are further notebooks for the Mathematica and SageMath interface. You can find more infos in the tutorials directory of the repository.

[2]:
```
import numpy as np
import os as os
```

1.1 The PointGenerator

In a first step we will use the PointGenerator to create points on a Calabi-Yau manifold. The point sampling is done with a theorem due to Shiffman and Zelditch, which has been discussed in detail yesterday.

Note if you have a CICY with $K > 1$ or a toric hypersurface, you'll need to import the *CICYPoint-Generator* or use the Mathematica package for the *ToricPointGenerator*.

[3]:
```
from cymetric.pointgen.pointgen import PointGenerator
```

The package is documented, which can be accessed as usual in python

[4]:
```
help(PointGenerator)
```

```
Help on class PointGenerator in module cymetric.pointgen.pointgen:

class PointGenerator(builtins.object)
 |  PointGenerator(monomials, coefficients, kmoduli, ambient, vol_j_norm=1,
verbose=2, backend='multiprocessing')
 |
 |  The PointGenerator class.
 |
 |  The numerics are entirely done in numpy; sympy is used for taking
 |  (implicit) derivatives.
 |
 |  Use this one if you want to generate points and data on a CY given by
 |  one hypersurface.
 |
 |  All other PointGenerators inherit from this class.
 |
 |  Example:
 |      We consider the Fermat quintic given by
 |
 |      .. math::
 |
 |          Q(z) = z_1^5 + z_2^5 + z_3^5 + z_4^5 + z_5^5
 |
 |      and set it up with:
 |
 |      >>> import numpy as np
 |      >>> from cymetric.pointgen.pointgen import PointGenerator
 |      >>> monomials = 5*np.eye(5, dtype=np.int)
 |      >>> coefficients = np.ones(5)
 |      >>> kmoduli = np.ones(1)
 |      >>> ambient = np.array([4])
 |      >>> pg = PointGenerator(monomials, coefficients, kmoduli, ambient)
 |
 |      Once the PointGenerator is initialized you can generate a training
 |      dataset with
 |
 |      >>> pg.prepare_dataset(number_of_points, dir_name)
 |
 |      and prepare the required tensorflow model data with
 |
 |      >>> pg.prepare_basis(dir_name)
 |
 |  Methods defined here:
 |
```

```
 |  __call__(self, points, vol_js=None)
 |      Computes the FS metric at each point.
 |
 |      Args:
 |          points (ndarray[(n_p, ncoords), np.complex128]): Points.
 |          vol_js (ndarray[(h^{(1,1)}), np.complex128]): vol_j factors.
 |              Defaults to None.
 |
 |      Returns:
 |          ndarray[(n_p, ncoords, ncoords), np.complex128]: g^FS
 |
 |  __init__(self, monomials, coefficients, kmoduli, ambient, vol_j_norm=1,
verbose=2, backend='multiprocessing')
 |      The PointGenerator uses the *joblib* module to parallelize
 |      computations.
 |
 |      Args:
 |          monomials (ndarray[(nMonomials, ncoords), np.int]): monomials
 |          coefficients (ndarray[(nMonomials)]): coefficients in front of each
 |              monomial.
 |          kmoduli (ndarray[(nProj)]): the kaehler moduli.
 |          ambient (ndarray[(nProj), np.int]): the direct product of projective
 |              spaces making up the ambient space.
 |          vol_j_norm (float, optional): Normalization of the volume of the
 |              Calabi-Yau X as computed from
 |
 |              .. math:: \int_X J^n \; \text{ at } \; t_1=t_2=…=t_n = 1.
 |
 |              Defaults to 1.
 |          verbose (int, optional): Controls logging. 1-Debug, 2-Info,
 |              else Warning. Defaults to 2.
 |          backend (str, optional): Backend for Parallel. Defaults to
 |              'multiprocessing'. 'loky' makes issues with pickle5.
 |
 |  compute_kappa(self, pw=[])
 |      We compute kappa from the Monge-Ampère equation
 |
 |      .. math:: J^3 = \kappa |\Omega|^2
 |
 |      such that after integrating we find
 |
 |      .. math::
 |
 |          \kappa = \frac{J^3}{|\Omega|^2} =
 |              \frac{\text{Vol}_K}{\text{Vol}_{\text{CY}}}
 |
 |      Args:
 |          pw (ndarray[(points, weight, omega)], optional):
```

```
 |              point weights generated from:
 |
 |              >>> self.generate_point_weights(np, omega=True)
 |
 |              Defaults to [], which then generates 10000 pws.
 |
 |      Returns:
 |          np.float: kappa
 |
 |  cy_condition(self, points)
 |      Computes the CY condition at each point.
 |
 |      Args:
 |          points (ndarray([n_p, ncoords], np.complex128)): Points (on the CY).
 |
 |      Returns:
 |          ndarray(n_p, np.complex128): CY condition
 |
 |  fubini_study_metrics(self, points, vol_js=None)
 |      Computes the FS metric at each point.
 |
 |      Args:
 |          points (ndarray[(n_p, ncoords), np.complex128]): Points.
 |          vol_js (ndarray[(h^{(1,1)}), np.complex128]): vol_j factor.
 |              Defaults to None.
 |
 |      Returns:
 |          ndarray[(n_p, ncoords, ncoords), np.complex128]: g^FS
 |
 |  generate_pn_points(self, n_p, n)
 |      Generates points on the sphere :math:`S^{2n+1}`.
 |
 |      Args:
 |          n_p (int): number of points.
 |          n (int): degree of projective space.
 |
 |      Returns:
 |          ndarray[(np, n+1), np.complex128]: complex points
 |
 |  generate_point_weights(self, n_pw, omega=False)
 |      Generates a numpy dictionary of point weights.
 |
 |      Args:
 |          n_pw (int): # of point weights.
 |          omega (bool, optional): If True adds Omega to dict.
 |              Defaults to False.
 |
 |      Returns:
```

```
|         np.dict: point weights
|
|  generate_points(self, n_p, nproc=-1, batch_size=5000)
|      Generates complex points on the CY.
|
|      The points are automatically scaled, such that the largest
|      coordinate in each projective space is 1+0.j.
|
|      Args:
|          n_p (int): # of points.
|          nproc (int, optional): # of jobs used. Defaults to -1. Then
|              uses all available resources.
|          batch_size (int, optional): batch_size of Parallel.
|              Defaults to 5000.
|
|      Returns:
|          ndarray[(n_p, ncoords), np.complex128]: rescaled points
|
|  holomorphic_volume_form(self, points, j_elim=None)
|      We compute the holomorphic volume form
|      at all points by solving the residue theorem:
|
|      .. math::
|
|          \Omega &= \int_\rho \frac{1}{Q} \wedge^n dz_i \\
|                 &= \frac{1}{\frac{\partial Q}{\partial z_j}}\wedge^{n-1} dz_a
|
|      where the index a runs over the local n-fold good coordinates.
|
|      Args:
|          points (ndarray[(n_p, ncoords), np.complex128]): Points.
|          j_elim (ndarray([n_p], np.int64)): index to be eliminated.
|              Defaults not None. If None eliminates max(dQdz).
|
|      Returns:
|          ndarray[(n_p), np.complex128]: Omega evaluated at each point.
|
|  point_weight(self, points, normalize_to_vol_j=False, j_elim=None)
|      We compute the weight/mass of each point:
|
|      .. math::
|
|          w &= \frac{d\text{Vol}_\text{cy}}{dA}|_p \\
|            &\sim \frac{|\Omega|^2}{\det(g^\text{FS}_{ab})}|_p
|
|      the weight depends on the distribution of free parameters during
|      point sampling. We employ a theorem due to Shiffman and Zelditch.
|      See also: [9803052].
```

```
 |
 |      Args:
 |          points (ndarray([n_p, ncoords], np.complex128)): Points.
 |          normalize_to_vol_j (bool, optional): Normalize such that
 |
 |              .. math::
 |
 |                  \int_X \det(g) &= \sum_i \sqrt{\det(g)} \cdot w|_{x_i}\\
 |                              &= d^{ijk} t_i t_j t_k.
 |
 |              Defaults to False.
 |          j_elim (ndarray([n_p, nhyper], np.int64)): Index to be eliminated.
 |              Defaults to None. If None eliminates max(dQdz).
 |
 |      Returns:
 |          ndarray([n_p], np.float64): weight at each point.
 |
 |  prepare_basis(self, dirname)
 |      Prepares pickled monomial basis for the tensorflow models.
 |
 |      Args:
 |          dirname (str): dir name to save
 |
 |      Returns:
 |          int: 0
 |
 |  prepare_dataset(self, n_p, dirname, val_split=0.1, ltails=0, rtails=0)
 |      Prepares training and validation data.
 |
 |      Args:
 |          n_p (int): Number of points to generate.
 |          dirname (str): Directory name to save dataset in.
 |          val_split (float, optional): train-val split. Defaults to 0.1.
 |          ltails (float, optional): Percentage discarded on the left tail
 |              of weight distribution. Defaults to 0.
 |          rtails (float, optional): Percentage discarded on the right tail
 |              of weight distribution. Defaults to 0.
 |
 |      Returns:
 |          int: 0
 |
 |  pullbacks(self, points, j_elim=None)
 |      Computes the pullback from ambient space to local CY coordinates
 |      at each point.
 |
 |      Denote the ambient space coordinates with z_i and the CY
 |      coordinates with x_a then
 |
```

```
|      .. math::
|
|          J^i_a = \frac{dz_i}{dx_a}
|
|      Args:
|          points (ndarray([n_p, ncoords], np.complex128)): Points.
|          j_elim (ndarray([n_p, nhyper], np.int64)): Index to be eliminated.
|              Defaults to None. If None eliminates max(dQdz).
|
|      Returns:
|          ndarray([n_p, nfold, ncoords], np.complex128): Pullback tensor
|              at each point.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

In this notebook we will consider a special member of the bicubic family. The Bicubic is given by a degree (3,3) polynomial in $\mathbb{P}^2 \times \mathbb{P}^2$. We chose a member of this family invariant under $\mathbb{Z}_3 \times \mathbb{Z}_3$ with the following coefficients:

$$-0.368597398314 18x_1^3x_4^3+0.05273312318807256x_1^3x_4x_5x_6+0.3777578902423778x_1^3x_5^3-0.2276208490552241x_1^3x_6^3-0.5$$

If you are looking for CICYs admitting some freely acting symmetries, check out Andre's webpage with the extended CICYlist containing all symmetries found by Braun.

Let's feed this data to the computer.

```
[5]: monomials = np.array([[0, 0, 3, 0, 0, 3],
            [0, 0, 3, 0, 3, 0],
            [0, 0, 3, 1, 1, 1],
            [0, 0, 3, 3, 0, 0],
            [0, 1, 2, 0, 2, 1],
            [0, 1, 2, 1, 0, 2],
            [0, 1, 2, 2, 1, 0],
            [0, 2, 1, 0, 1, 2],
            [0, 2, 1, 1, 2, 0],
            [0, 2, 1, 2, 0, 1],
            [0, 3, 0, 0, 0, 3],
            [0, 3, 0, 0, 3, 0],
            [0, 3, 0, 1, 1, 1],
            [0, 3, 0, 3, 0, 0],
```

```
        [1, 0, 2, 0, 1, 2],
        [1, 0, 2, 1, 2, 0],
        [1, 0, 2, 2, 0, 1],
        [1, 1, 1, 0, 0, 3],
        [1, 1, 1, 0, 3, 0],
        [1, 1, 1, 1, 1, 1],
        [1, 1, 1, 3, 0, 0],
        [1, 2, 0, 0, 2, 1],
        [1, 2, 0, 1, 0, 2],
        [1, 2, 0, 2, 1, 0],
        [2, 0, 1, 0, 2, 1],
        [2, 0, 1, 1, 0, 2],
        [2, 0, 1, 2, 1, 0],
        [2, 1, 0, 0, 1, 2],
        [2, 1, 0, 1, 2, 0],
        [2, 1, 0, 2, 0, 1],
        [3, 0, 0, 0, 0, 3],
        [3, 0, 0, 0, 3, 0],
        [3, 0, 0, 1, 1, 1],
        [3, 0, 0, 3, 0, 0]])
coefficients = np.array([-0.3685974 , -0.22762085,  0.05273312,  0.37775789,  0.
 ↪44197442,
       -0.27154207,  0.37192549,  0.52839796, -0.52824873,  1.73375885,
        0.37775789, -0.3685974 ,  0.05273312, -0.22762085, -0.52824873,
        1.73375885,  0.52839796, -0.20844236, -0.20844236, -0.10526534,
       -0.20844236, -0.27154207,  0.37192549,  0.44197442,  0.37192549,
        0.44197442, -0.27154207,  1.73375885,  0.52839796, -0.52824873,
       -0.22762085,  0.37775789,  0.05273312, -0.3685974 ])
kmoduli = np.ones(2)
ambient = np.array([2,2])
```

and can then initialise the *PointGenerator*, which requires the information about monominals, coefficient (cmoduli), ambient space and Kähler moduli (wrt to the FS metric).

[6]:
```
pg = PointGenerator(monomials, coefficients, kmoduli, ambient)
```

To create a dataset, we give the *PointGenerator* a directory and the number of points:

[7]:
```
help(pg.prepare_dataset)
```

```
Help on method prepare_dataset in module cymetric.pointgen.pointgen:

prepare_dataset(n_p, dirname, val_split=0.1, ltails=0, rtails=0) method of
cymetric.pointgen.pointgen.PointGenerator instance
    Prepares training and validation data.

    Args:
        n_p (int): Number of points to generate.
        dirname (str): Directory name to save dataset in.
```

```
        val_split (float, optional): train-val split. Defaults to 0.1.
        ltails (float, optional): Percentage discarded on the left tail
            of weight distribution. Defaults to 0.
        rtails (float, optional): Percentage discarded on the right tail
            of weight distribution. Defaults to 0.

    Returns:
        int: 0
```

[8]:
```
dirname = 'bicubic'
n_p = 77000# so that it finishes within the 25 mins
pg.prepare_dataset(n_p, dirname)
```

[8]:
```
0
```

The dataset can subsequently be loaded with NumPy.

[9]:
```
data = np.load(os.path.join(dirname, 'dataset.npz'))
for entry in data:
    print(entry, data[entry].shape)
```

```
X_train (69300, 12)
y_train (69300, 2)
X_val (7700, 12)
y_val (7700, 2)
val_pullbacks (7700, 3, 6)
```

The training data contains real points

[10]:
```
print(data['X_train'][0:5])
```

```
[[ 1.00000000e+00 -3.42877599e-01 -3.84113210e-01  1.58526139e-01
   1.00000000e+00  4.47923984e-01  2.77555756e-17 -7.81247238e-02
  -1.85793594e-01  5.73089957e-01 -1.38777878e-17  4.30812818e-02]
 [ 1.06363520e-01  3.00956309e-02  1.00000000e+00  1.58526139e-01
   1.00000000e+00  4.47923984e-01 -2.65341976e-01 -4.04525238e-01
   5.55111512e-17  5.73089957e-01 -1.38777878e-17  4.30812818e-02]
 [ 5.72591140e-01 -1.80183185e-01  1.00000000e+00  1.58526139e-01
   1.00000000e+00  4.47923984e-01 -7.47168948e-01 -4.04628919e-01
   5.55111512e-17  5.73089957e-01 -1.38777878e-17  4.30812818e-02]
 [-9.73883928e-02  1.00000000e+00 -2.29005058e-01 -5.32178220e-03
   5.24514544e-01  1.00000000e+00 -4.87785052e-01  0.00000000e+00
   1.70522815e-01  3.04006310e-03  1.42763493e-01  0.00000000e+00]
 [ 2.04770938e-01  1.00000000e+00  4.13629351e-01 -5.32178220e-03
   5.24514544e-01  1.00000000e+00  6.79606434e-01  0.00000000e+00
  -3.06608850e-01  3.04006310e-03  1.42763493e-01  0.00000000e+00]]
```

since our neural nets will be implemented with real weights. They can be made complex by adding the first half of cols to the second half x i.

[11]:
```
cpoints = data['X_train'][0:5,0:pg.ncoords] + 1.j*data['X_train'][0:5,pg.
 ↪ncoords:]
print(cpoints)
```

```
[[ 1.        +2.77555756e-17j -0.3428776 -7.81247238e-02j
  -0.38411321-1.85793594e-01j  0.15852614+5.73089957e-01j
   1.        -1.38777878e-17j  0.44792398+4.30812818e-02j]
 [ 0.10636352-2.65341976e-01j  0.03009563-4.04525238e-01j
   1.        +5.55111512e-17j  0.15852614+5.73089957e-01j
   1.        -1.38777878e-17j  0.44792398+4.30812818e-02j]
 [ 0.57259114-7.47168948e-01j -0.18018319-4.04628919e-01j
   1.        +5.55111512e-17j  0.15852614+5.73089957e-01j
   1.        -1.38777878e-17j  0.44792398+4.30812818e-02j]
 [-0.09738839-4.87785052e-01j  1.        +0.00000000e+00j
  -0.22900506+1.70522815e-01j -0.00532178+3.04006310e-03j
   0.52451454+1.42763493e-01j  1.        +0.00000000e+00j]
 [ 0.20477094+6.79606434e-01j  1.        +0.00000000e+00j
   0.41362935-3.06608850e-01j -0.00532178+3.04006310e-03j
   0.52451454+1.42763493e-01j  1.        +0.00000000e+00j]]
```

We can see how the patch information is encoded by using the scaling relations to set $\max(x) = 1$. Note, the two 1s coming from the scaling relations of each $\mathbb{P}^2$.

But are these points on the CY?

[12]:
```
pg.cy_condition(cpoints)
```

[12]:
```
array([-3.40005801e-16+8.74300632e-16j, -3.00215582e-15+2.45289900e-15j,
       -1.39471767e-14+8.18789481e-16j, -2.75746120e-16-3.27490829e-16j,
        1.60454497e-16+3.84292615e-16j])
```

Yes! The PointGenerator has various other convenience functions for pullbacks, holomorphic n-form Ω, FS metric, … .

Finally, we store all the data encoding the information of the underlying CY in a *basis.pickle* file. Those are the defining monomials, their derivatives, as well as complex and Kähler moduli and a bunch of other stuff. The MetricModels later will require all this information in their internal computations of the pullback tensors and transition matrices.

[13]:
```
pg.prepare_basis(dirname)
```

[13]:
```
0
```

We inspect this data by loading it from the file

[14]:
```
BASIS = np.load(os.path.join(dirname, 'basis.pickle'), allow_pickle=True)
for entry in BASIS:
    print(entry)
```

```
DQDZB0
DQDZF0
```

```
QBO
QFO
NFOLD
AMBIENT
KMODULI
NHYPER
```

1.2 The MetricModel

In the second part of this tutorial we will train a neural net to learn the Ricci-flat metric on the Bicubic specified above. We will utilise the ϕ-model introduced yesterday by Fabian. Recall that it is given by the following Ansatz:

$$g_{CY} = g_{FS} + \partial\bar{\partial}\phi$$

and thus by construction Kähler, and moreover in the same Kähler class as the reference Fubini-Study metric. There are also other possible Ansätze which can be imported and modified. For example:

Model	Ansatz	Class
free	gNN	FreeModel, ToricModel
addition	gFS + gNN	AddFSModel, AddFSModelToric
mult	gFS + (gFS $\odot$ gNN)	MultFSModel, MultFSModelToric
matrix	gFS + (gFS $\cdot$ gNN)	MatrixFSModel, MatrixFSModelToric

However as shown in our NeurIPS paper the ϕ-model works best. First, we import tensorflow, some utility functions, the ϕ-model (*PhiFSModel* or *PhiFSModelToric*), and callbacks + metrics to keep track of the training process.

[15]:
```
import tensorflow as tf
import logging
# trying and failing to silence tensorflow
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
logging.getLogger('tensorflow').setLevel(logging.FATAL)
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.FATAL)
tf.get_logger().setLevel('ERROR')
tfk = tf.keras
from cymetric.models.callbacks import RicciCallback, SigmaCallback, VolkCallback
from cymetric.models.tfmodels import PhiFSModel
from cymetric.models.metrics import SigmaLoss, TransitionLoss, TotalLoss
from plot_cb import PlotLearning
```

again documentation exists and sometimes even includes minimal working examples

[16]:
```
help(PhiFSModel)
```

```
Help on class PhiFSModel in module cymetric.models.tfmodels:

class PhiFSModel(FreeModel)
 |  PhiFSModel(*args, **kwargs)
 |
 |  PhiFSModel inherits from :py:class:`FreeModel`.
 |
 |  The PhiModel learns the scalar potential correction to some Kaehler metric
 |  to make it the Ricci-flat metric. The Kaehler metric is taken to be the
 |  Fubini-Study metric.
 |
 |  Example:
 |      Is similar to :py:class:`FreeModel`. Replace the nn accordingly.
 |
 |      >>> nn = tfk.Sequential(
 |      …     [
 |      …         tfk.layers.Input(shape=(ncoords)),
 |      …         tfk.layers.Dense(64, activation="gelu"),
 |      …         tfk.layers.Dense(1),
 |      …     ]
 |      … )
 |      >>> model = PhiFSModel(nn, BASIS)
 |
 |  You have to use this model if you want to remain in the same Kaehler class
 |  specified by the Kaehler moduli.
 |
 |  Method resolution order:
 |      PhiFSModel
 |      FreeModel
 |      cymetric.models.fubinistudy.FSModel
 |      keras.engine.training.Model
 |      keras.engine.base_layer.Layer
 |      tensorflow.python.module.module.Module
 |      tensorflow.python.training.tracking.tracking.AutoTrackable
 |      tensorflow.python.training.tracking.base.Trackable
 |      keras.utils.version_utils.LayerVersionSelector
 |      keras.utils.version_utils.ModelVersionSelector
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *args, **kwargs)
 |      PhiFSModel is a tensorflow model predicting CY metrics.
 |
 |      The output of this model has the following Ansatz
 |
 |      .. math::
 |
```

```
 |          g_{\text{out}} = g_{\text{FS}} +
 |              \partial \bar{\partial} \phi_{\text{NN}}
 |
 |      and returns a hermitian (nfold, nfold) tensor. The model is by
 |      defintion Kaehler and thus this loss contribution is by default
 |      disabled. For similar reasons the Volk loss is also disabled if
 |      the last layer does not contain a bias. Otherwise it is required
 |      for successful tracing.
 |
 |  call(self, input_tensor, training=True, j_elim=None)
 |      Prediction of the model.
 |
 |      .. math::
 |
 |          g_{\text{out}; ij} = g_{\text{FS}; ij} + \
 |              partial_i \bar{\partial}_j \phi_{\text{NN}}
 |
 |      Args:
 |          input_tensor (tf.tensor([bSize, 2*ncoords], tf.float32)): Points.
 |          training (bool, optional): Not used. Defaults to True.
 |          j_elim (tf.tensor([bSize, nHyper], tf.int64), optional):
 |              Coordinates(s) to be eliminated in the pullbacks.
 |              If None will take max(dQ/dz). Defaults to None.
 |
 |      Returns:
 |          tf.tensor([bSize, nfold, nfold], tf.complex64):
 |              Prediction at each point.
 |
 |  compute_volk_loss(self, input_tensor, weights, pred=None)
 |      Computes volk loss.
 |
 |      .. math::
 |
 |          \mathcal{L}_{\text{vol}_k} = \int_X \phi
 |
 |      The last term is constant over the whole batch. Thus, the volk loss
 |      is *batch dependent*. This loss contribution should be satisfied by
 |      construction but is included for tracing purposes.
 |
 |      Args:
 |          input_tensor (tf.tensor([bSize, 2*ncoords], tf.float32)): Points.
 |          weights (tf.tensor([bSize], tf.float32)): Weights.
 |          pred (tf.tensor([bSize, nfold, nfold], tf.complex64), optional):
 |              Prediction from `self(input_tensor)`.
 |              If None will be calculated. Defaults to None.
 |
 |      Returns:
 |          tf.tensor([bSize], tf.float32): Volk loss.
```

```
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from FreeModel:
 |
 |  compile(self, custom_metrics=None, **kwargs)
 |      Compiles the model.
 |
 |      kwargs takes any argument of regular `tf.model.compile()`
 |
 |      Example:
 |          >>> model = FreeModel(nn, BASIS)
 |          >>> from cymetric.models.metrics import TotalLoss
 |          >>> metrics = [TotalLoss()]
 |          >>> opt = tfk.optimizers.Adam()
 |          >>> model.compile(custom_metrics = metrics, optimizer = opt)
 |
 |      Args:
 |          custom_metrics (list, optional): List of custom metrics.
 |              See also :py:mod:`cymetric.models.metrics`. If None, no metrics
 |              are tracked during training. Defaults to None.
 |
 |  save(self, filepath, **kwargs)
 |      Saves the underlying neural network to filepath.
 |
 |      NOTE:
 |          Currently does not save the whole custom model.
 |
 |      Args:
 |          filepath (str): filepath
 |
 |  test_step(self, data)
 |      Same as train_step without the outer gradient tape.
 |      Does *not* update the NN weights.
 |
 |      NOTE:
 |          1. Computes the exaxt same losses as train_step
 |
 |          2. Ricci loss val can be separately enabled with
 |
 |              >>> model.learn_ricci_val = True
 |
 |          3. Requires additional tracing.
 |
 |      Args:
 |          data (tuple): test_data (x,y, sample_weight)
 |
 |      Returns:
 |          dict: metrics
```

```
 |
 |  to_hermitian(self, x)
 |      Returns a hermitian tensor.
 |
 |      Takes a tensor of length (-1,nfold**2) and transforms it
 |      into a (-1,nfold,nfold) hermitian matrix.
 |
 |      Args:
 |          x (tensor[(-1,nfold**2), tf.float]): input tensor
 |
 |      Returns:
 |          tensor[(-1,nfold,nfold), tf.float]: hermitian matrix
 |
 |  train_step(self, data)
 |      Train step of a single batch in model.fit().
 |
 |      NOTE:
 |          1. The first epoch will take additional time, due to tracing.
 |
 |          2. Warnings are plentiful. Disable on your own risk with
 |
 |              >>> tf.get_logger().setLevel('ERROR')
 |
 |          3. The conditionals need to be set before tracing.
 |
 |          4. We employ under the hood gradient clipping.
 |
 |      Args:
 |          data (tuple): test_data (x,y, sample_weight)
 |
 |      Returns:
 |          dict: metrics
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from FreeModel:
 |
 |  metrics
 |      Returns the models metrics including custom metrics.
 |
 |      Returns:
 |          list: metrics
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from cymetric.models.fubinistudy.FSModel:
 |
 |  compute_kaehler_loss(self, x)
 |      Computes Kähler loss.
 |
```

```
 |      .. math::
 |          \cal{L}_{\text{dJ}} = \sum_{ijk} ||Re(c_{ijk})||_n +
 |                  ||Im(c_{ijk})||_n \\
 |              \text{with: } c_{ijk} = g_{i\bar{j},k} - g_{k\bar{j},i}
 |
 |      Args:
 |          x (tf.tensor([bSize, 2*ncoords], tf.float)): Points.
 |
 |      Returns:
 |          tf.tensor([bSize, 1], tf.float): \sum_ijk abs(cijk)**n
 |
 |  compute_ricci_loss(self, points, pb=None)
 |      Computes the absolute value of the Ricci scalar for each point,
 |      then takes the norm.
 |
 |      .. seealso:: method :py:meth:`.compute_ricci_scalar`.
 |
 |      Args:
 |          points (tf.tensor([bSize, 2*ncoords], tf.float)): Points.
 |          pb (tf.tensor([bSize, nfold, ncoords], tf.float), optional):
 |              Pullback tensor at each point. Defaults to None.
 |
 |      Returns:
 |          tf.tensor([bSize], tf.float): |R|_n.
 |
 |  compute_ricci_scalar(self, points, pb=None)
 |      Computes the Ricci scalar for each point.
 |
 |      .. math::
 |
 |          R = g^{ij} J_i^a \bar{J}_j^b \partial_a \bar{\partial}_b
 |              \log \det g
 |
 |      Args:
 |          points (tf.tensor([bSize, 2*ncoords], tf.float)): Points.
 |          pb (tf.tensor([bSize, nfold, ncoords], tf.float), optional):
 |              Pullback tensor at each point. Defaults to None.
 |
 |      Returns:
 |          tf.tensor([bSize], tf.float): R|_p.
 |
 |  compute_transition_loss(self, points)
 |      Computes transition loss at each point.
 |
 |      .. math::
 |
 |          \mathcal{L} = \frac{1}{d} \sum_{k,j}
 |              ||g^k - T_{jk} \cdot g^j T^\dagger_{jk}||_n
```

```
 |
 |      Args:
 |          points (tf.tensor([bSize, 2*ncoords], tf.float32)): Points.
 |
 |      Returns:
 |          tf.tensor([bSize], tf.float32): Transition loss at each point.
 |
 |  fubini_study_pb(self, points, pb=None, j_elim=None)
 |      Computes the pullbacked Fubini-Study metric.
 |
 |      NOTE:
 |          The pb argument overwrites j_elim.
 |
 |      .. math::
 |
 |          g_{ij} = \frac{1}{\pi} J_i^a \bar{J}_j^b \partial_a
 |              \bar{\partial}_b \ln |\vec{z}|^2
 |
 |
 |      Args:
 |          points (tf.tensor([bSize, 2*ncoords], tf.float32)): Points.
 |          pb (tf.tensor([bSize, nfold, ncoords], tf.float32)):
 |              Pullback at each point. Overwrite j_elim. Defaults to None.
 |          j_elim (tf.array([bSize], tf.int64)): index to be eliminated.
 |              Coordinates(s) to be eliminated in the pullbacks.
 |              If None will take max(dQ/dz). Defaults to None.
 |
 |      Returns:
 |          tf.tensor([bSize, nfold, nfold], tf.complex64):
 |              FS-metric at each point.
 |
 |  get_transition_matrix(self, points, i_mask, j_mask, fixed)
 |      Computes transition matrix between patch i and j
 |      for each point in points where fixed is the coordinate,
 |      which is being eliminated.
 |
 |      Example (by hand):
 |          Consider the bicubic with:
 |
 |          .. math::
 |
 |              P_1^2 [a_0 : a_1 : a_2] \text{ and } P_2^2 [b_0 : b_1 : b_2].
 |
 |          Assume we eliminate :math:`b_2` and keep it fixed. Then we
 |          consider two patches.
 |          Patch 1 where :math:`a_0 = b_0 = 1` with new coordinates
 |          :math:`(x_1, x_2, x_3) = (a_1/a_0, a_2/a_0, b_1/b_0)`
 |          Patch 2 where :math:`a_1=b_1=1` with new coordinates
```

```
|           :math:`(w_1, w_2, w_3) = (a_0/a_1, a_2/a_1, b_0/b_1)`
|           such that we can reexpress w in terms of x:
|           :math:`w_1(x)=1/x_1,\; w_2(x)=x_2/x_1,\; w_3(x)=1/x_3`
|           from which follows:
|
|           .. math::
|
|               T_{11} &= \frac{\partial w_1}{\partial x_1} =
|                   -1/x_1^2 = -a_0^2/a_1^2 \\
|               T_{12} &= \frac{\partial w_2}{\partial x_1} =
|                   -x_2/x_1^2 = -a_2 a_0/a_1^2 \\
|               T_{13} &= \frac{\partial w_3}{\partial x_1} = 0 \\
|               T_{21} &= \frac{\partial w_1}{\partial x_2} = 0 \\
|                      & \dots
|
|       Args:
|           points (tf.tensor([bSize, 2*ncoords], tf.float32)): Points.
|           i_mask (tf.tensor([bSize, ncoords], tf.bool)): Mask of pi-indices.
|           j_mask (tf.tensor([bSize, ncoords], tf.bool)): Mask of pi-indices.
|           fixed (tf.tensor([bSize, 1], tf.int64)): Elimination indices.
|
|       Returns:
|           tf.tensor([bSize, nfold, nfold], tf.complex64): T_ij on the CY.
|
|   pullbacks(self, points, j_elim=None)
|       Computes the pullback tensor at each point.
|
|       NOTE:
|           Scatter-nd uses a while loop when creating the graph.
|
|       .. math::
|
|           J^i_a = \frac{dz_i}{dx_a}
|
|       where x_a are the nfold good coordinates after eliminating j_elim.
|
|       Args:
|           points (tf.tensor([bSize, 2*ncoords], tf.float32)): Points.
|           j_elim (tf.tensor([bSize, nHyper], tf.int64), optional):
|               Coordinates(s) to be eliminated in the pullbacks.
|               If None will take max(dQ/dz). Defaults to None.
|
|       Returns:
|           tf.tensor([bSize, nfold, ncoords], tf.complex64): Pullback at each
|               point.
|
|   transition_loss_matrices(self, gj, gi, Tij)
|       Computes transition loss matrix between metric
```

```
 |      in patches i and j with transition matrix Tij.
 |
 |      Args:
 |          gj (tf.tensor([bSize, nfold, nfold], tf.complex64)):
 |              Metric in patch j.
 |          gi (tf.tensor([bSize, nfold, nfold], tf.complex64)):
 |              Metric in patch i.
 |          Tij (tf.tensor([bSize, nfold, nfold], tf.complex64)):
 |              Transition matrix from patch i to patch j.
 |
 |      Returns:
 |          tf.tensor([bSize, nfold, nfold], tf.complex64):
 |              .. math::`g_j - T^{ij} g_i T^{ij,\dagger}`
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from keras.engine.training.Model:
 |
 |  __copy__(self)
 |
 |  __deepcopy__(self, memo)
 |
 |  __reduce__(self)
 |      Helper for pickle.
 |
 |  __setattr__(self, name, value)
 |      Support self.foo = trackable syntax.
 |
 |  build(self, input_shape)
 |      Builds the model based on input shapes received.
 |
 |      This is to be used for subclassed models, which do not know at
instantiation
 |      time what their inputs look like.
 |
 |      This method only exists for users who want to call `model.build()` in a
 |      standalone way (as a substitute for calling the model on real data to
 |      build it). It will never be called by the framework (and thus it will
 |      never throw unexpected errors in an unrelated workflow).
 |
 |      Args:
 |       input_shape: Single tuple, `TensorShape` instance, or list/dict of
shapes,
 |         where shapes are tuples, integers, or `TensorShape` instances.
 |
 |      Raises:
 |        ValueError:
 |          1. In case of invalid user-provided data (not of type tuple,
 |             list, `TensorShape`, or dict).
```

```
 |          2. If the model requires call arguments that are agnostic
 |             to the input shapes (positional or keyword arg in call
signature).
 |          3. If not all layers were properly built.
 |          4. If float type inputs are not supported within the layers.
 |
 |        In each of these cases, the user should build their model by calling
it
 |        on real tensor data.
 |
 |  evaluate(self, x=None, y=None, batch_size=None, verbose=1,
sample_weight=None, steps=None, callbacks=None, max_queue_size=10, workers=1,
use_multiprocessing=False, return_dict=False, **kwargs)
 |      Returns the loss value & metrics values for the model in test mode.
 |
 |      Computation is done in batches (see the `batch_size` arg.)
 |
 |      Args:
 |          x: Input data. It could be:
 |            - A Numpy array (or array-like), or a list of arrays
 |              (in case the model has multiple inputs).
 |            - A TensorFlow tensor, or a list of tensors
 |              (in case the model has multiple inputs).
 |            - A dict mapping input names to the corresponding array/tensors,
 |              if the model has named inputs.
 |            - A `tf.data` dataset. Should return a tuple
 |              of either `(inputs, targets)` or
 |              `(inputs, targets, sample_weights)`.
 |            - A generator or `keras.utils.Sequence` returning `(inputs,
targets)`
 |              or `(inputs, targets, sample_weights)`.
 |            A more detailed description of unpacking behavior for iterator
types
 |            (Dataset, generator, Sequence) is given in the `Unpacking behavior
 |            for iterator-like inputs` section of `Model.fit`.
 |          y: Target data. Like the input data `x`, it could be either Numpy
 |            array(s) or TensorFlow tensor(s). It should be consistent with `x`
 |            (you cannot have Numpy inputs and tensor targets, or inversely).
If
 |            `x` is a dataset, generator or `keras.utils.Sequence` instance,
`y`
 |            should not be specified (since targets will be obtained from the
 |            iterator/dataset).
 |          batch_size: Integer or `None`. Number of samples per batch of
 |            computation. If unspecified, `batch_size` will default to 32. Do
not
 |            specify the `batch_size` if your data is in the form of a dataset,
 |            generators, or `keras.utils.Sequence` instances (since they
```

```
generate
 |              batches).
 |          verbose: 0 or 1. Verbosity mode. 0 = silent, 1 = progress bar.
 |          sample_weight: Optional Numpy array of weights for the test samples,
 |            used for weighting the loss function. You can either pass a flat
(1D)
 |            Numpy array with the same length as the input samples
 |              (1:1 mapping between weights and samples), or in the case of
 |                temporal data, you can pass a 2D array with shape `(samples,
 |                sequence_length)`, to apply a different weight to every
timestep
 |                of every sample. This argument is not supported when `x` is a
 |                dataset, instead pass sample weights as the third element of
`x`.
 |          steps: Integer or `None`. Total number of steps (batches of samples)
 |            before declaring the evaluation round finished. Ignored with the
 |            default value of `None`. If x is a `tf.data` dataset and `steps`
is
 |            None, 'evaluate' will run until the dataset is exhausted. This
 |            argument is not supported with array inputs.
 |          callbacks: List of `keras.callbacks.Callback` instances. List of
 |            callbacks to apply during evaluation. See
 |            [callbacks](/api_docs/python/tf/keras/callbacks).
 |          max_queue_size: Integer. Used for generator or
`keras.utils.Sequence`
 |            input only. Maximum size for the generator queue. If unspecified,
 |            `max_queue_size` will default to 10.
 |          workers: Integer. Used for generator or `keras.utils.Sequence` input
 |            only. Maximum number of processes to spin up when using process-
based
 |            threading. If unspecified, `workers` will default to 1.
 |          use_multiprocessing: Boolean. Used for generator or
 |            `keras.utils.Sequence` input only. If `True`, use process-based
 |            threading. If unspecified, `use_multiprocessing` will default to
 |            `False`. Note that because this implementation relies on
 |            multiprocessing, you should not pass non-picklable arguments to
the
 |            generator as they can't be passed easily to children processes.
 |          return_dict: If `True`, loss and metric results are returned as a
dict,
 |            with each key being the name of the metric. If `False`, they are
 |            returned as a list.
 |          **kwargs: Unused at this time.
 |
 |      See the discussion of `Unpacking behavior for iterator-like inputs` for
 |      `Model.fit`.
 |
 |      `Model.evaluate` is not yet supported with
```

```
 |      `tf.distribute.experimental.ParameterServerStrategy`.
 |
 |      Returns:
 |          Scalar test loss (if the model has a single output and no metrics)
 |          or list of scalars (if the model has multiple outputs
 |          and/or metrics). The attribute `model.metrics_names` will give you
 |          the display labels for the scalar outputs.
 |
 |      Raises:
 |          RuntimeError: If `model.evaluate` is wrapped in a `tf.function`.
 |
 |  evaluate_generator(self, generator, steps=None, callbacks=None,
max_queue_size=10, workers=1, use_multiprocessing=False, verbose=0)
 |      Evaluates the model on a data generator.
 |
 |      DEPRECATED:
 |        `Model.evaluate` now supports generators, so there is no longer any
need
 |        to use this endpoint.
 |
 |  fit(self, x=None, y=None, batch_size=None, epochs=1, verbose='auto',
callbacks=None, validation_split=0.0, validation_data=None, shuffle=True,
class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None,
validation_steps=None, validation_batch_size=None, validation_freq=1,
max_queue_size=10, workers=1, use_multiprocessing=False)
 |      Trains the model for a fixed number of epochs (iterations on a dataset).
 |
 |      Args:
 |          x: Input data. It could be:
 |            - A Numpy array (or array-like), or a list of arrays
 |              (in case the model has multiple inputs).
 |            - A TensorFlow tensor, or a list of tensors
 |              (in case the model has multiple inputs).
 |            - A dict mapping input names to the corresponding array/tensors,
 |              if the model has named inputs.
 |            - A `tf.data` dataset. Should return a tuple
 |              of either `(inputs, targets)` or
 |              `(inputs, targets, sample_weights)`.
 |            - A generator or `keras.utils.Sequence` returning `(inputs,
targets)`
 |              or `(inputs, targets, sample_weights)`.
 |            - A `tf.keras.utils.experimental.DatasetCreator`, which wraps a
 |              callable that takes a single argument of type
 |              `tf.distribute.InputContext`, and returns a `tf.data.Dataset`.
 |              `DatasetCreator` should be used when users prefer to specify the
 |              per-replica batching and sharding logic for the `Dataset`.
 |              See `tf.keras.utils.experimental.DatasetCreator` doc for more
 |              information.
```

```
 |           A more detailed description of unpacking behavior for iterator
types
 |           (Dataset, generator, Sequence) is given below. If using
 |           `tf.distribute.experimental.ParameterServerStrategy`, only
 |           `DatasetCreator` type is supported for `x`.
 |         y: Target data. Like the input data `x`,
 |           it could be either Numpy array(s) or TensorFlow tensor(s).
 |           It should be consistent with `x` (you cannot have Numpy inputs and
 |           tensor targets, or inversely). If `x` is a dataset, generator,
 |           or `keras.utils.Sequence` instance, `y` should
 |           not be specified (since targets will be obtained from `x`).
 |         batch_size: Integer or `None`.
 |             Number of samples per gradient update.
 |             If unspecified, `batch_size` will default to 32.
 |             Do not specify the `batch_size` if your data is in the
 |             form of datasets, generators, or `keras.utils.Sequence`
instances
 |             (since they generate batches).
 |         epochs: Integer. Number of epochs to train the model.
 |             An epoch is an iteration over the entire `x` and `y`
 |             data provided
 |             (unless the `steps_per_epoch` flag is set to
 |             something other than None).
 |             Note that in conjunction with `initial_epoch`,
 |             `epochs` is to be understood as "final epoch".
 |             The model is not trained for a number of iterations
 |             given by `epochs`, but merely until the epoch
 |             of index `epochs` is reached.
 |         verbose: 'auto', 0, 1, or 2. Verbosity mode.
 |             0 = silent, 1 = progress bar, 2 = one line per epoch.
 |             'auto' defaults to 1 for most cases, but 2 when used with
 |             `ParameterServerStrategy`. Note that the progress bar is not
 |             particularly useful when logged to a file, so verbose=2 is
 |             recommended when not running interactively (eg, in a production
 |             environment).
 |         callbacks: List of `keras.callbacks.Callback` instances.
 |             List of callbacks to apply during training.
 |             See `tf.keras.callbacks`. Note
`tf.keras.callbacks.ProgbarLogger`
 |             and `tf.keras.callbacks.History` callbacks are created
automatically
 |             and need not be passed into `model.fit`.
 |             `tf.keras.callbacks.ProgbarLogger` is created or not based on
 |             `verbose` argument to `model.fit`.
 |             Callbacks with batch-level calls are currently unsupported with
 |             `tf.distribute.experimental.ParameterServerStrategy`, and users
are
 |             advised to implement epoch-level calls instead with an
```

```
appropriate
 |              `steps_per_epoch` value.
 |          validation_split: Float between 0 and 1.
 |              Fraction of the training data to be used as validation data.
 |              The model will set apart this fraction of the training data,
 |              will not train on it, and will evaluate
 |              the loss and any model metrics
 |              on this data at the end of each epoch.
 |              The validation data is selected from the last samples
 |              in the `x` and `y` data provided, before shuffling. This
argument is
 |              not supported when `x` is a dataset, generator or
 |             `keras.utils.Sequence` instance.
 |              `validation_split` is not yet supported with
 |              `tf.distribute.experimental.ParameterServerStrategy`.
 |          validation_data: Data on which to evaluate
 |              the loss and any model metrics at the end of each epoch.
 |              The model will not be trained on this data. Thus, note the fact
 |              that the validation loss of data provided using
`validation_split`
 |              or `validation_data` is not affected by regularization layers
like
 |              noise and dropout.
 |              `validation_data` will override `validation_split`.
 |              `validation_data` could be:
 |                - A tuple `(x_val, y_val)` of Numpy arrays or tensors.
 |                - A tuple `(x_val, y_val, val_sample_weights)` of NumPy
arrays.
 |                - A `tf.data.Dataset`.
 |                - A Python generator or `keras.utils.Sequence` returning
 |                `(inputs, targets)` or `(inputs, targets, sample_weights)`.
 |              `validation_data` is not yet supported with
 |              `tf.distribute.experimental.ParameterServerStrategy`.
 |          shuffle: Boolean (whether to shuffle the training data
 |              before each epoch) or str (for 'batch'). This argument is
ignored
 |              when `x` is a generator or an object of tf.data.Dataset.
 |              'batch' is a special option for dealing
 |              with the limitations of HDF5 data; it shuffles in batch-sized
 |              chunks. Has no effect when `steps_per_epoch` is not `None`.
 |          class_weight: Optional dictionary mapping class indices (integers)
 |              to a weight (float) value, used for weighting the loss function
 |              (during training only).
 |              This can be useful to tell the model to
 |              "pay more attention" to samples from
 |              an under-represented class.
 |          sample_weight: Optional Numpy array of weights for
 |              the training samples, used for weighting the loss function
```

```
 |              (during training only). You can either pass a flat (1D)
 |              Numpy array with the same length as the input samples
 |              (1:1 mapping between weights and samples),
 |              or in the case of temporal data,
 |              you can pass a 2D array with shape
 |              `(samples, sequence_length)`,
 |              to apply a different weight to every timestep of every sample.
This
 |              argument is not supported when `x` is a dataset, generator, or
 |             `keras.utils.Sequence` instance, instead provide the
sample_weights
 |              as the third element of `x`.
 |          initial_epoch: Integer.
 |              Epoch at which to start training
 |              (useful for resuming a previous training run).
 |          steps_per_epoch: Integer or `None`.
 |              Total number of steps (batches of samples)
 |              before declaring one epoch finished and starting the
 |              next epoch. When training with input tensors such as
 |              TensorFlow data tensors, the default `None` is equal to
 |              the number of samples in your dataset divided by
 |              the batch size, or 1 if that cannot be determined. If x is a
 |              `tf.data` dataset, and 'steps_per_epoch'
 |              is None, the epoch will run until the input dataset is
exhausted.
 |              When passing an infinitely repeating dataset, you must specify
the
 |              `steps_per_epoch` argument. If `steps_per_epoch=-1` the training
 |              will run indefinitely with an infinitely repeating dataset.
 |              This argument is not supported with array inputs.
 |              When using `tf.distribute.experimental.ParameterServerStrategy`:
 |                * `steps_per_epoch=None` is not supported.
 |          validation_steps: Only relevant if `validation_data` is provided and
 |              is a `tf.data` dataset. Total number of steps (batches of
 |              samples) to draw before stopping when performing validation
 |              at the end of every epoch. If 'validation_steps' is None,
validation
 |              will run until the `validation_data` dataset is exhausted. In
the
 |              case of an infinitely repeated dataset, it will run into an
 |              infinite loop. If 'validation_steps' is specified and only part
of
 |              the dataset will be consumed, the evaluation will start from the
 |              beginning of the dataset at each epoch. This ensures that the
same
 |              validation samples are used every time.
 |          validation_batch_size: Integer or `None`.
 |              Number of samples per validation batch.
```

```
 |              If unspecified, will default to `batch_size`.
 |              Do not specify the `validation_batch_size` if your data is in
the
 |              form of datasets, generators, or `keras.utils.Sequence`
instances
 |              (since they generate batches).
 |          validation_freq: Only relevant if validation data is provided.
Integer
 |              or `collections.abc.Container` instance (e.g. list, tuple,
etc.).
 |              If an integer, specifies how many training epochs to run before
a
 |              new validation run is performed, e.g. `validation_freq=2` runs
 |              validation every 2 epochs. If a Container, specifies the epochs
on
 |              which to run validation, e.g. `validation_freq=[1, 2, 10]` runs
 |              validation at the end of the 1st, 2nd, and 10th epochs.
 |          max_queue_size: Integer. Used for generator or
`keras.utils.Sequence`
 |              input only. Maximum size for the generator queue.
 |              If unspecified, `max_queue_size` will default to 10.
 |          workers: Integer. Used for generator or `keras.utils.Sequence` input
 |              only. Maximum number of processes to spin up
 |              when using process-based threading. If unspecified, `workers`
 |              will default to 1.
 |          use_multiprocessing: Boolean. Used for generator or
 |              `keras.utils.Sequence` input only. If `True`, use process-based
 |              threading. If unspecified, `use_multiprocessing` will default to
 |              `False`. Note that because this implementation relies on
 |              multiprocessing, you should not pass non-picklable arguments to
 |              the generator as they can't be passed easily to children
processes.
 |
 |      Unpacking behavior for iterator-like inputs:
 |          A common pattern is to pass a tf.data.Dataset, generator, or
 |        tf.keras.utils.Sequence to the `x` argument of fit, which will in fact
 |        yield not only features (x) but optionally targets (y) and sample
weights.
 |        Keras requires that the output of such iterator-likes be unambiguous.
The
 |        iterator should return a tuple of length 1, 2, or 3, where the
optional
 |        second and third elements will be used for y and sample_weight
 |        respectively. Any other type provided will be wrapped in a length one
 |        tuple, effectively treating everything as 'x'. When yielding dicts,
they
 |        should still adhere to the top-level tuple structure.
 |        e.g. `({"x0": x0, "x1": x1}, y)`. Keras will not attempt to separate
```

```
 |          features, targets, and weights from the keys of a single dict.
 |            A notable unsupported data type is the namedtuple. The reason is
that
 |          it behaves like both an ordered datatype (tuple) and a mapping
 |          datatype (dict). So given a namedtuple of the form:
 |              `namedtuple("example_tuple", ["y", "x"])`
 |          it is ambiguous whether to reverse the order of the elements when
 |          interpreting the value. Even worse is a tuple of the form:
 |              `namedtuple("other_tuple", ["x", "y", "z"])`
 |          where it is unclear if the tuple was intended to be unpacked into x,
y,
 |          and sample_weight or passed through as a single element to `x`. As a
 |          result the data processing code will simply raise a ValueError if it
 |          encounters a namedtuple. (Along with instructions to remedy the
issue.)
 |
 |      Returns:
 |          A `History` object. Its `History.history` attribute is
 |          a record of training loss values and metrics values
 |          at successive epochs, as well as validation loss values
 |          and validation metrics values (if applicable).
 |
 |      Raises:
 |          RuntimeError: 1. If the model was never compiled or,
 |          2. If `model.fit` is  wrapped in `tf.function`.
 |
 |          ValueError: In case of mismatch between the provided input data
 |              and what the model expects or when the input data is empty.
 |
 |  fit_generator(self, generator, steps_per_epoch=None, epochs=1, verbose=1,
callbacks=None, validation_data=None, validation_steps=None, validation_freq=1,
class_weight=None, max_queue_size=10, workers=1, use_multiprocessing=False,
shuffle=True, initial_epoch=0)
 |      Fits the model on data yielded batch-by-batch by a Python generator.
 |
 |      DEPRECATED:
 |        `Model.fit` now supports generators, so there is no longer any need to
use
 |        this endpoint.
 |
 |  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).
 |
 |      Note that `get_config()` does not guarantee to return a fresh copy of
dict
 |      every time it is called. The callers should make a copy of the returned
dict
 |      if they want to modify it.
 |
 |      Returns:
 |          Python dictionary.
 |
 |  get_layer(self, name=None, index=None)
 |      Retrieves a layer based on either its name (unique) or index.
 |
 |      If `name` and `index` are both provided, `index` will take precedence.
 |      Indices are based on order of horizontal graph traversal (bottom-up).
 |
 |      Args:
 |          name: String, name of layer.
 |          index: Integer, index of layer.
 |
 |      Returns:
 |          A layer instance.
 |
 |  get_weights(self)
 |      Retrieves the weights of the model.
 |
 |      Returns:
 |          A flat list of Numpy arrays.
 |
 |  load_weights(self, filepath, by_name=False, skip_mismatch=False,
options=None)
 |      Loads all layer weights, either from a TensorFlow or an HDF5 weight
file.
 |
 |      If `by_name` is False weights are loaded based on the network's
 |      topology. This means the architecture should be the same as when the
weights
 |      were saved.  Note that layers that don't have weights are not taken into
 |      account in the topological ordering, so adding or removing layers is
fine as
 |      long as they don't have weights.
 |
 |      If `by_name` is True, weights are loaded into layers only if they share
the
 |      same name. This is useful for fine-tuning or transfer-learning models
```

```
where
 |      some of the layers have changed.
 |
 |      Only topological loading (`by_name=False`) is supported when loading
weights
 |      from the TensorFlow format. Note that topological loading differs
slightly
 |      between TensorFlow and HDF5 formats for user-defined classes inheriting
from
 |      `tf.keras.Model`: HDF5 loads based on a flattened list of weights, while
the
 |      TensorFlow format loads based on the object-local names of attributes to
 |      which layers are assigned in the `Model`'s constructor.
 |
 |      Args:
 |          filepath: String, path to the weights file to load. For weight files
in
 |              TensorFlow format, this is the file prefix (the same as was
passed
 |              to `save_weights`). This can also be a path to a SavedModel
 |              saved from `model.save`.
 |          by_name: Boolean, whether to load weights by name or by topological
 |              order. Only topological loading is supported for weight files in
 |              TensorFlow format.
 |          skip_mismatch: Boolean, whether to skip loading of layers where
there is
 |              a mismatch in the number of weights, or a mismatch in the shape
of
 |              the weight (only valid when `by_name=True`).
 |          options: Optional `tf.train.CheckpointOptions` object that specifies
 |              options for loading weights.
 |
 |      Returns:
 |          When loading a weight file in TensorFlow format, returns the same
status
 |          object as `tf.train.Checkpoint.restore`. When graph building,
restore
 |          ops are run automatically as soon as the network is built (on first
call
 |          for user-defined classes inheriting from `Model`, immediately if it
is
 |          already built).
 |
 |          When loading weights in HDF5 format, returns `None`.
 |
 |      Raises:
 |          ImportError: If `h5py` is not available and the weight file is in
HDF5
```

```
 |              format.
 |          ValueError: If `skip_mismatch` is set to `True` when `by_name` is
 |            `False`.
 |
 |  make_predict_function(self, force=False)
 |      Creates a function that executes one step of inference.
 |
 |      This method can be overridden to support custom inference logic.
 |      This method is called by `Model.predict` and `Model.predict_on_batch`.
 |
 |      Typically, this method directly controls `tf.function` and
 |      `tf.distribute.Strategy` settings, and delegates the actual evaluation
 |      logic to `Model.predict_step`.
 |
 |      This function is cached the first time `Model.predict` or
 |      `Model.predict_on_batch` is called. The cache is cleared whenever
 |      `Model.compile` is called. You can skip the cache and generate again the
 |      function with `force=True`.
 |
 |      Args:
 |        force: Whether to regenerate the predict function and skip the cached
 |          function if available.
 |
 |      Returns:
 |        Function. The function created by this method should accept a
 |        `tf.data.Iterator`, and return the outputs of the `Model`.
 |
 |  make_test_function(self, force=False)
 |      Creates a function that executes one step of evaluation.
 |
 |      This method can be overridden to support custom evaluation logic.
 |      This method is called by `Model.evaluate` and `Model.test_on_batch`.
 |
 |      Typically, this method directly controls `tf.function` and
 |      `tf.distribute.Strategy` settings, and delegates the actual evaluation
 |      logic to `Model.test_step`.
 |
 |      This function is cached the first time `Model.evaluate` or
 |      `Model.test_on_batch` is called. The cache is cleared whenever
 |      `Model.compile` is called. You can skip the cache and generate again the
 |      function with `force=True`.
 |
 |      Args:
 |        force: Whether to regenerate the test function and skip the cached
 |          function if available.
 |
 |      Returns:
 |        Function. The function created by this method should accept a
```

```
 |        `tf.data.Iterator`, and return a `dict` containing values that will
 |        be passed to `tf.keras.Callbacks.on_test_batch_end`.
 |
 |  make_train_function(self, force=False)
 |      Creates a function that executes one step of training.
 |
 |      This method can be overridden to support custom training logic.
 |      This method is called by `Model.fit` and `Model.train_on_batch`.
 |
 |      Typically, this method directly controls `tf.function` and
 |      `tf.distribute.Strategy` settings, and delegates the actual training
 |      logic to `Model.train_step`.
 |
 |      This function is cached the first time `Model.fit` or
 |      `Model.train_on_batch` is called. The cache is cleared whenever
 |      `Model.compile` is called. You can skip the cache and generate again the
 |      function with `force=True`.
 |
 |      Args:
 |        force: Whether to regenerate the train function and skip the cached
 |          function if available.
 |
 |      Returns:
 |        Function. The function created by this method should accept a
 |        `tf.data.Iterator`, and return a `dict` containing values that will
 |        be passed to `tf.keras.Callbacks.on_train_batch_end`, such as
 |        `{'loss': 0.2, 'accuracy': 0.7}`.
 |
 |  predict(self, x, batch_size=None, verbose=0, steps=None, callbacks=None,
max_queue_size=10, workers=1, use_multiprocessing=False)
 |      Generates output predictions for the input samples.
 |
 |      Computation is done in batches. This method is designed for performance
in
 |      large scale inputs. For small amount of inputs that fit in one batch,
 |      directly using `__call__()` is recommended for faster execution, e.g.,
 |      `model(x)`, or `model(x, training=False)` if you have layers such as
 |      `tf.keras.layers.BatchNormalization` that behaves differently during
 |      inference. Also, note the fact that test loss is not affected by
 |      regularization layers like noise and dropout.
 |
 |      Args:
 |          x: Input samples. It could be:
 |            - A Numpy array (or array-like), or a list of arrays
 |              (in case the model has multiple inputs).
 |            - A TensorFlow tensor, or a list of tensors
 |              (in case the model has multiple inputs).
 |            - A `tf.data` dataset.
```

```
 |              - A generator or `keras.utils.Sequence` instance.
 |              A more detailed description of unpacking behavior for iterator
types
 |              (Dataset, generator, Sequence) is given in the `Unpacking behavior
 |              for iterator-like inputs` section of `Model.fit`.
 |            batch_size: Integer or `None`.
 |                Number of samples per batch.
 |                If unspecified, `batch_size` will default to 32.
 |                Do not specify the `batch_size` if your data is in the
 |                form of dataset, generators, or `keras.utils.Sequence` instances
 |                (since they generate batches).
 |            verbose: Verbosity mode, 0 or 1.
 |            steps: Total number of steps (batches of samples)
 |                before declaring the prediction round finished.
 |                Ignored with the default value of `None`. If x is a `tf.data`
 |                dataset and `steps` is None, `predict()` will
 |                run until the input dataset is exhausted.
 |            callbacks: List of `keras.callbacks.Callback` instances.
 |                List of callbacks to apply during prediction.
 |                See [callbacks](/api_docs/python/tf/keras/callbacks).
 |            max_queue_size: Integer. Used for generator or
`keras.utils.Sequence`
 |                input only. Maximum size for the generator queue.
 |                If unspecified, `max_queue_size` will default to 10.
 |            workers: Integer. Used for generator or `keras.utils.Sequence` input
 |                only. Maximum number of processes to spin up when using
 |                process-based threading. If unspecified, `workers` will default
 |                to 1.
 |            use_multiprocessing: Boolean. Used for generator or
 |                `keras.utils.Sequence` input only. If `True`, use process-based
 |                threading. If unspecified, `use_multiprocessing` will default to
 |                `False`. Note that because this implementation relies on
 |                multiprocessing, you should not pass non-picklable arguments to
 |                the generator as they can't be passed easily to children
processes.
 |
 |        See the discussion of `Unpacking behavior for iterator-like inputs` for
 |        `Model.fit`. Note that Model.predict uses the same interpretation rules
as
 |        `Model.fit` and `Model.evaluate`, so inputs must be unambiguous for all
 |        three methods.
 |
 |        Returns:
 |            Numpy array(s) of predictions.
 |
 |        Raises:
 |            RuntimeError: If `model.predict` is wrapped in a `tf.function`.
 |            ValueError: In case of mismatch between the provided
```

```
 |              input data and the model's expectations,
 |              or in case a stateful model receives a number of samples
 |              that is not a multiple of the batch size.
 |
 |  predict_generator(self, generator, steps=None, callbacks=None,
max_queue_size=10, workers=1, use_multiprocessing=False, verbose=0)
 |      Generates predictions for the input samples from a data generator.
 |
 |      DEPRECATED:
 |        `Model.predict` now supports generators, so there is no longer any
need
 |        to use this endpoint.
 |
 |  predict_on_batch(self, x)
 |      Returns predictions for a single batch of samples.
 |
 |      Args:
 |          x: Input data. It could be:
 |            - A Numpy array (or array-like), or a list of arrays (in case the
 |                model has multiple inputs).
 |            - A TensorFlow tensor, or a list of tensors (in case the model has
 |                multiple inputs).
 |
 |      Returns:
 |          Numpy array(s) of predictions.
 |
 |      Raises:
 |          RuntimeError: If `model.predict_on_batch` is wrapped in a
`tf.function`.
 |
 |  predict_step(self, data)
 |      The logic for one inference step.
 |
 |      This method can be overridden to support custom inference logic.
 |      This method is called by `Model.make_predict_function`.
 |
 |      This method should contain the mathematical logic for one step of
inference.
 |      This typically includes the forward pass.
 |
 |      Configuration details for *how* this logic is run (e.g. `tf.function`
and
 |      `tf.distribute.Strategy` settings), should be left to
 |      `Model.make_predict_function`, which can also be overridden.
 |
 |      Args:
 |        data: A nested structure of `Tensor`s.
 |
```

```
 |      Returns:
 |        The result of one inference step, typically the output of calling the
 |        `Model` on data.
 |
 |  reset_metrics(self)
 |      Resets the state of all the metrics in the model.
 |
 |      Examples:
 |
 |      >>> inputs = tf.keras.layers.Input(shape=(3,))
 |      >>> outputs = tf.keras.layers.Dense(2)(inputs)
 |      >>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
 |      >>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
 |
 |      >>> x = np.random.random((2, 3))
 |      >>> y = np.random.randint(0, 2, (2, 2))
 |      >>> _ = model.fit(x, y, verbose=0)
 |      >>> assert all(float(m.result()) for m in model.metrics)
 |
 |      >>> model.reset_metrics()
 |      >>> assert all(float(m.result()) == 0 for m in model.metrics)
 |
 |  reset_states(self)
 |
 |  save_spec(self, dynamic_batch=True)
 |      Returns the `tf.TensorSpec` of call inputs as a tuple `(args, kwargs)`.
 |
 |      This value is automatically defined after calling the model for the
first
 |      time. Afterwards, you can use it when exporting the model for serving:
 |
 |      ```python
 |      model = tf.keras.Model(…)
 |
 |      @tf.function
 |      def serve(*args, **kwargs):
 |        outputs = model(*args, **kwargs)
 |        # Apply postprocessing steps, or add additional outputs.
 |        …
 |        return outputs
 |
 |      # arg_specs is `[tf.TensorSpec(…), …]`. kwarg_specs, in this
example, is
 |      # an empty dict since functional models do not use keyword arguments.
 |      arg_specs, kwarg_specs = model.save_spec()
 |
 |      model.save(path, signatures={
 |        'serving_default': serve.get_concrete_function(*arg_specs,
```

```
**kwarg_specs)
 |      })
 |      ```
 |
 |      Args:
 |        dynamic_batch: Whether to set the batch sizes of all the returned
 |          `tf.TensorSpec` to `None`. (Note that when defining functional or
 |          Sequential models with `tf.keras.Input([…], batch_size=X)`, the
 |          batch size will always be preserved). Defaults to `True`.
 |      Returns:
 |        If the model inputs are defined, returns a tuple `(args, kwargs)`. All
 |        elements in `args` and `kwargs` are `tf.TensorSpec`.
 |        If the model inputs are not defined, returns `None`.
 |        The model inputs are automatically set when calling the model,
 |        `model.fit`, `model.evaluate` or `model.predict`.
 |
 |  save_weights(self, filepath, overwrite=True, save_format=None, options=None)
 |      Saves all layer weights.
 |
 |      Either saves in HDF5 or in TensorFlow format based on the `save_format`
 |      argument.
 |
 |      When saving in HDF5 format, the weight file has:
 |        - `layer_names` (attribute), a list of strings
 |            (ordered names of model layers).
 |        - For every layer, a `group` named `layer.name`
 |            - For every such layer group, a group attribute `weight_names`,
 |                a list of strings
 |                (ordered names of weights tensor of the layer).
 |            - For every weight in the layer, a dataset
 |                storing the weight value, named after the weight tensor.
 |
 |      When saving in TensorFlow format, all objects referenced by the network
are
 |      saved in the same format as `tf.train.Checkpoint`, including any `Layer`
 |      instances or `Optimizer` instances assigned to object attributes. For
 |      networks constructed from inputs and outputs using
`tf.keras.Model(inputs,
 |      outputs)`, `Layer` instances used by the network are tracked/saved
 |      automatically. For user-defined classes which inherit from
`tf.keras.Model`,
 |      `Layer` instances must be assigned to object attributes, typically in
the
 |      constructor. See the documentation of `tf.train.Checkpoint` and
 |      `tf.keras.Model` for details.
 |
 |      While the formats are the same, do not mix `save_weights` and
 |      `tf.train.Checkpoint`. Checkpoints saved by `Model.save_weights` should
```

```
be
 |      loaded using `Model.load_weights`. Checkpoints saved using
 |      `tf.train.Checkpoint.save` should be restored using the corresponding
 |      `tf.train.Checkpoint.restore`. Prefer `tf.train.Checkpoint` over
 |      `save_weights` for training checkpoints.
 |
 |      The TensorFlow format matches objects and variables by starting at a
root
 |      object, `self` for `save_weights`, and greedily matching attribute
 |      names. For `Model.save` this is the `Model`, and for `Checkpoint.save`
this
 |      is the `Checkpoint` even if the `Checkpoint` has a model attached. This
 |      means saving a `tf.keras.Model` using `save_weights` and loading into a
 |      `tf.train.Checkpoint` with a `Model` attached (or vice versa) will not
match
 |      the `Model`'s variables. See the
 |      [guide to training
checkpoints](https://www.tensorflow.org/guide/checkpoint)
 |      for details on the TensorFlow format.
 |
 |      Args:
 |          filepath: String or PathLike, path to the file to save the weights
to.
 |              When saving in TensorFlow format, this is the prefix used for
 |              checkpoint files (multiple files are generated). Note that the
'.h5'
 |              suffix causes weights to be saved in HDF5 format.
 |          overwrite: Whether to silently overwrite any existing file at the
 |              target location, or provide the user with a manual prompt.
 |          save_format: Either 'tf' or 'h5'. A `filepath` ending in '.h5' or
 |              '.keras' will default to HDF5 if `save_format` is `None`.
Otherwise
 |              `None` defaults to 'tf'.
 |          options: Optional `tf.train.CheckpointOptions` object that specifies
 |              options for saving weights.
 |
 |      Raises:
 |          ImportError: If `h5py` is not available when attempting to save in
HDF5
 |              format.
 |
 |  summary(self, line_length=None, positions=None, print_fn=None,
expand_nested=False)
 |      Prints a string summary of the network.
 |
 |      Args:
 |          line_length: Total length of printed lines
 |              (e.g. set this to adapt the display to different
```

```
 |              terminal window sizes).
 |          positions: Relative or absolute positions of log elements
 |              in each line. If not provided,
 |              defaults to `[.33, .55, .67, 1.]`.
 |          print_fn: Print function to use. Defaults to `print`.
 |              It will be called on each line of the summary.
 |              You can set it to a custom function
 |              in order to capture the string summary.
 |          expand_nested: Whether to expand the nested models.
 |              If not provided, defaults to `False`.
 |
 |      Raises:
 |          ValueError: if `summary()` is called before the model is built.
 |
 |  test_on_batch(self, x, y=None, sample_weight=None, reset_metrics=True,
return_dict=False)
 |      Test the model on a single batch of samples.
 |
 |      Args:
 |          x: Input data. It could be:
 |            - A Numpy array (or array-like), or a list of arrays (in case the
 |                model has multiple inputs).
 |            - A TensorFlow tensor, or a list of tensors (in case the model has
 |                multiple inputs).
 |            - A dict mapping input names to the corresponding array/tensors,
if
 |                the model has named inputs.
 |          y: Target data. Like the input data `x`, it could be either Numpy
 |            array(s) or TensorFlow tensor(s). It should be consistent with `x`
 |            (you cannot have Numpy inputs and tensor targets, or inversely).
 |          sample_weight: Optional array of the same length as x, containing
 |            weights to apply to the model's loss for each sample. In the case
of
 |            temporal data, you can pass a 2D array with shape (samples,
 |            sequence_length), to apply a different weight to every timestep of
 |            every sample.
 |          reset_metrics: If `True`, the metrics returned will be only for this
 |            batch. If `False`, the metrics will be statefully accumulated
across
 |            batches.
 |          return_dict: If `True`, loss and metric results are returned as a
dict,
 |            with each key being the name of the metric. If `False`, they are
 |            returned as a list.
 |
 |      Returns:
 |          Scalar test loss (if the model has a single output and no metrics)
 |          or list of scalars (if the model has multiple outputs
```

```
 |          and/or metrics). The attribute `model.metrics_names` will give you
 |          the display labels for the scalar outputs.
 |
 |      Raises:
 |          RuntimeError: If `model.test_on_batch` is wrapped in a
`tf.function`.
 |
 |  to_json(self, **kwargs)
 |      Returns a JSON string containing the network configuration.
 |
 |      To load a network from a JSON save file, use
 |      `keras.models.model_from_json(json_string, custom_objects={})`.
 |
 |      Args:
 |          **kwargs: Additional keyword arguments
 |              to be passed to `json.dumps()`.
 |
 |      Returns:
 |          A JSON string.
 |
 |  to_yaml(self, **kwargs)
 |      Returns a yaml string containing the network configuration.
 |
 |      Note: Since TF 2.6, this method is no longer supported and will raise a
 |      RuntimeError.
 |
 |      To load a network from a yaml save file, use
 |      `keras.models.model_from_yaml(yaml_string, custom_objects={})`.
 |
 |      `custom_objects` should be a dictionary mapping
 |      the names of custom losses / layers / etc to the corresponding
 |      functions / classes.
 |
 |      Args:
 |          **kwargs: Additional keyword arguments
 |              to be passed to `yaml.dump()`.
 |
 |      Returns:
 |          A YAML string.
 |
 |      Raises:
 |          RuntimeError: announces that the method poses a security risk
 |
 |  train_on_batch(self, x, y=None, sample_weight=None, class_weight=None,
reset_metrics=True, return_dict=False)
 |      Runs a single gradient update on a single batch of data.
 |
 |      Args:
```

```
 |          x: Input data. It could be:
 |            - A Numpy array (or array-like), or a list of arrays
 |                (in case the model has multiple inputs).
 |            - A TensorFlow tensor, or a list of tensors
 |                (in case the model has multiple inputs).
 |            - A dict mapping input names to the corresponding array/tensors,
 |                if the model has named inputs.
 |          y: Target data. Like the input data `x`, it could be either Numpy
 |            array(s) or TensorFlow tensor(s). It should be consistent with `x`
 |            (you cannot have Numpy inputs and tensor targets, or inversely).
 |          sample_weight: Optional array of the same length as x, containing
 |            weights to apply to the model's loss for each sample. In the case
of
 |            temporal data, you can pass a 2D array with shape (samples,
 |            sequence_length), to apply a different weight to every timestep of
 |            every sample.
 |          class_weight: Optional dictionary mapping class indices (integers)
to a
 |            weight (float) to apply to the model's loss for the samples from
this
 |            class during training. This can be useful to tell the model to
"pay
 |            more attention" to samples from an under-represented class.
 |          reset_metrics: If `True`, the metrics returned will be only for this
 |            batch. If `False`, the metrics will be statefully accumulated
across
 |            batches.
 |          return_dict: If `True`, loss and metric results are returned as a
dict,
 |            with each key being the name of the metric. If `False`, they are
 |            returned as a list.
 |
 |      Returns:
 |          Scalar training loss
 |          (if the model has a single output and no metrics)
 |          or list of scalars (if the model has multiple outputs
 |          and/or metrics). The attribute `model.metrics_names` will give you
 |          the display labels for the scalar outputs.
 |
 |      Raises:
 |        RuntimeError: If `model.train_on_batch` is wrapped in a `tf.function`.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from keras.engine.training.Model:
 |
 |  from_config(config, custom_objects=None) 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`).
|
|      Args:
|          config: A Python dictionary, typically the
|              output of get_config.
|
|      Returns:
|          A layer instance.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from keras.engine.training.Model:
|
|  __new__(cls, *args, **kwargs)
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from keras.engine.training.Model:
|
|  distribute_strategy
|      The `tf.distribute.Strategy` this model was created under.
|
|  layers
|
|  metrics_names
|      Returns the model's display labels for all outputs.
|
|      Note: `metrics_names` are available only after a `keras.Model` has been
|      trained/evaluated on actual data.
|
|      Examples:
|
|      >>> inputs = tf.keras.layers.Input(shape=(3,))
|      >>> outputs = tf.keras.layers.Dense(2)(inputs)
|      >>> model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
|      >>> model.compile(optimizer="Adam", loss="mse", metrics=["mae"])
|      >>> model.metrics_names
|      []
|
|      >>> x = np.random.random((2, 3))
|      >>> y = np.random.randint(0, 2, (2, 2))
|      >>> model.fit(x, y)
|      >>> model.metrics_names
|      ['loss', 'mae']
|
|      >>> inputs = tf.keras.layers.Input(shape=(3,))
```

```
 |      >>> d = tf.keras.layers.Dense(2, name='out')
 |      >>> output_1 = d(inputs)
 |      >>> output_2 = d(inputs)
 |      >>> model = tf.keras.models.Model(
 |      …    inputs=inputs, outputs=[output_1, output_2])
 |      >>> model.compile(optimizer="Adam", loss="mse", metrics=["mae", "acc"])
 |      >>> model.fit(x, (y, y))
 |      >>> model.metrics_names
 |      ['loss', 'out_loss', 'out_1_loss', 'out_mae', 'out_acc', 'out_1_mae',
 |      'out_1_acc']
 |
 |  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.
 |
 |  run_eagerly
 |      Settable attribute indicating whether the model should run eagerly.
 |
 |      Running eagerly means that your model will be run step by step,
 |      like Python code. Your model might run slower, but it should become
easier
 |      for you to debug it by stepping into individual layer calls.
 |
 |      By default, we will attempt to compile your model to a static graph to
 |      deliver the best execution performance.
 |
 |      Returns:
 |        Boolean, whether the model should run eagerly.
 |
 |  state_updates
 |      Deprecated, do NOT use!
 |
 |      Returns the `updates` from all layers that are stateful.
 |
 |      This is useful for separating training updates and
 |      state updates, e.g. when we need to update a layer's internal state
 |      during prediction.
 |
 |      Returns:
 |          A list of update ops.
 |
 |  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.
 |
 |  weights
 |      Returns the list of all layer variables/weights.
 |
 |      Note: This will not track the weights of nested `tf.Modules` that are
not
 |      themselves Keras layers.
 |
 |      Returns:
 |        A list of variables.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from keras.engine.base_layer.Layer:
 |
 |  __call__(self, *args, **kwargs)
 |      Wraps `call`, applying pre- and post-processing steps.
 |
 |      Args:
 |        *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.
 |        - If the layer is not built, the method will call `build`.
 |
 |      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)
 |
 |  __setstate__(self, state)
 |
 |  add_loss(self, losses, **kwargs)
 |      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(self, inputs):
 |          self.add_loss(tf.abs(tf.reduce_mean(inputs)))
 |          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,))
 |      d = tf.keras.layers.Dense(10)
 |      x = d(inputs)
 |      outputs = tf.keras.layers.Dense(1)(x)
 |      model = tf.keras.Model(inputs, outputs)
 |      # Weight regularization.
 |      model.add_loss(lambda: tf.reduce_mean(d.kernel))
 |      ```
 |
 |      Args:
 |        losses: Loss tensor, or list/tuple of tensors. Rather than tensors,
losses
 |          may also be zero-argument callables which create a loss tensor.
 |        **kwargs: Additional keyword arguments for backward compatibility.
 |          Accepted values:
 |            inputs - Deprecated, will be automatically inferred.
 |
 |  add_metric(self, value, name=None, **kwargs)
 |      Adds metric tensor to the layer.
 |
 |      This method can be used inside the `call()` method of a subclassed layer
 |      or model.
 |
 |      ```python
 |      class MyMetricLayer(tf.keras.layers.Layer):
 |        def __init__(self):
 |          super(MyMetricLayer, self).__init__(name='my_metric_layer')
 |          self.mean = tf.keras.metrics.Mean(name='metric_1')
 |
 |        def call(self, inputs):
 |          self.add_metric(self.mean(inputs))
 |          self.add_metric(tf.reduce_sum(inputs), name='metric_2')
 |          return inputs
 |      ```
 |
 |      This method can also be called directly on a Functional Model during
 |      construction. In this case, any tensor passed to this Model must
```

```
 |      be symbolic and be able to be traced back to the model's `Input`s. These
 |      metrics become part of the model's topology and are tracked when you
 |      save the model via `save()`.
 |
 |      ```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)
 |      model.add_metric(math_ops.reduce_sum(x), name='metric_1')
 |      ```
 |
 |      Note: Calling `add_metric()` with the result of a metric object on a
 |      Functional Model, as shown in the example below, is not supported. This
is
 |      because we cannot trace the metric result tensor back to the model's
inputs.
 |
 |      ```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)
 |      model.add_metric(tf.keras.metrics.Mean()(x), name='metric_1')
 |      ```
 |
 |      Args:
 |        value: Metric tensor.
 |        name: String metric name.
 |        **kwargs: Additional keyword arguments for backward compatibility.
 |          Accepted values:
 |          `aggregation` - When the `value` tensor provided is not the result
of
 |          calling a `keras.Metric` instance, it will be aggregated by default
 |          using a `keras.Metric.Mean`.
 |
 |  add_update(self, updates, inputs=None)
 |      Add update op(s), potentially dependent on layer inputs.
 |
 |      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.
 |
 |      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).
 |
 |      Args:
 |        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`.
 |
 |  add_weight(self, name=None, shape=None, dtype=None, initializer=None,
regularizer=None, trainable=None, constraint=None, use_resource=None,
synchronization=<VariableSynchronization.AUTO: 0>,
aggregation=<VariableAggregationV2.NONE: 0>, **kwargs)
 |      Adds a new variable to the layer.
 |
 |      Args:
 |        name: Variable name.
 |        shape: Variable shape. Defaults to scalar if unspecified.
 |        dtype: The type of the variable. Defaults to `self.dtype`.
 |        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).
 |        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 variable created.
```

```
 |
 |      Raises:
 |        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!
 |
 |      This is an alias of `self.__call__`.
 |
 |      Args:
 |        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.
 |
 |      Args:
 |          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_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.
 |
 |      Args:
 |          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.
 |
 |  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).
 |
 |  finalize_state(self)
 |      Finalizes the layers state after updating layer weights.
 |
 |      This function can be subclassed in a layer and will be called after
updating
 |      a layer weights. It can be overridden to finalize any additional layer
state
 |      after a weight update.
 |
 |  get_input_at(self, node_index)
 |      Retrieves the input tensor(s) of a layer at a given node.
 |
 |      Args:
 |          node_index: Integer, index of the node
 |              from which to retrieve the attribute.
 |              E.g. `node_index=0` will correspond to the
 |              first input node of the layer.
 |
```

```
 |      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.
 |
 |      Args:
 |          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.
 |
 |      Args:
 |          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)
 |      Deprecated, do NOT use!
 |
 |      Retrieves losses relevant to a specific set of inputs.
 |
 |      Args:
 |        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.
```

```
 |
 |      Args:
 |          node_index: Integer, index of the node
 |              from which to retrieve the attribute.
 |              E.g. `node_index=0` will correspond to the
 |              first output node of the layer.
 |
 |      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.
 |
 |      Args:
 |          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.
 |
 |      Args:
 |          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)
 |      Deprecated, do NOT use!
 |
 |      Retrieves updates relevant to a specific set of inputs.
 |
 |      Args:
```

```
 |        inputs: Input tensor or list/tuple of input tensors.
 |
 |      Returns:
 |        List of update ops of the layer that depend on `inputs`.
 |
 |  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: the kernel
matrix
 |      and the bias vector. These can be used to set the weights of another
 |      `Dense` layer:
 |
 |      >>> layer_a = tf.keras.layers.Dense(1,
 |      …   kernel_initializer=tf.constant_initializer(1.))
 |      >>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
 |      >>> layer_a.get_weights()
 |      [array([[1.],
 |             [1.],
 |             [1.]], dtype=float32), array([0.], dtype=float32)]
 |      >>> layer_b = tf.keras.layers.Dense(1,
 |      …   kernel_initializer=tf.constant_initializer(2.))
 |      >>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
 |      >>> layer_b.get_weights()
 |      [array([[2.],
 |             [2.],
 |             [2.]], dtype=float32), array([0.], dtype=float32)]
 |      >>> layer_b.set_weights(layer_a.get_weights())
 |      >>> layer_b.get_weights()
 |      [array([[1.],
 |             [1.],
 |             [1.]], dtype=float32), array([0.], dtype=float32)]
 |
 |      Args:
 |        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.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from keras.engine.base_layer.Layer:
 |
 |  activity_regularizer
 |      Optional regularizer function for the output of this layer.
 |
 |  compute_dtype
 |      The dtype of the layer's computations.
 |
 |      This is equivalent to `Layer.dtype_policy.compute_dtype`. Unless
 |      mixed precision is used, this is the same as `Layer.dtype`, the dtype of
 |      the weights.
 |
 |      Layers automatically cast their inputs to the compute dtype, which
causes
 |      computations and the output to be in the compute dtype as well. This is
done
 |      by the base Layer class in `Layer.__call__`, so you do not have to
insert
 |      these casts if implementing your own layer.
 |
 |      Layers often perform certain internal computations in higher precision
when
 |      `compute_dtype` is float16 or bfloat16 for numeric stability. The output
 |      will still typically be float16 or bfloat16 in such cases.
 |
 |      Returns:
 |        The layer's compute dtype.
 |
 |  dtype
 |      The dtype of the layer weights.
 |
 |      This is equivalent to `Layer.dtype_policy.variable_dtype`. Unless
 |      mixed precision is used, this is the same as `Layer.compute_dtype`, the
 |      dtype of the layer's computations.
 |
 |  dtype_policy
 |      The dtype policy associated with this layer.
 |
 |      This is an instance of a `tf.keras.mixed_precision.Policy`.
 |
 |  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
 |      List of losses added using the `add_loss()` API.
 |
 |      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.
 |
 |      Examples:
 |
 |      >>> class MyLayer(tf.keras.layers.Layer):
 |      …   def call(self, inputs):
 |      …     self.add_loss(tf.abs(tf.reduce_mean(inputs)))
 |      …     return inputs
 |      >>> l = MyLayer()
 |      >>> l(np.ones((10, 1)))
 |      >>> l.losses
 |      [1.0]
```

```
 |
 |      >>> 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.
 |      >>> len(model.losses)
 |      0
 |      >>> model.add_loss(tf.abs(tf.reduce_mean(x)))
 |      >>> len(model.losses)
 |      1
 |
 |      >>> inputs = tf.keras.Input(shape=(10,))
 |      >>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
 |      >>> x = d(inputs)
 |      >>> outputs = tf.keras.layers.Dense(1)(x)
 |      >>> model = tf.keras.Model(inputs, outputs)
 |      >>> # Weight regularization.
 |      >>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
 |      >>> model.losses
 |      [<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
 |
 |      Returns:
 |        A list of tensors.
 |
 |  name
 |      Name of the layer (string), set in the constructor.
 |
 |  non_trainable_variables
 |      Sequence of non-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).
 |
 |  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
 |
 |  supports_masking
 |      Whether this layer supports computing a mask using `compute_mask`.
 |
 |  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).
 |
 |  updates
 |
 |  variable_dtype
 |      Alias of `Layer.dtype`, the dtype of the weights.
 |
 |  variables
 |      Returns the list of all layer variables/weights.
 |
 |      Alias of `self.weights`.
 |
 |      Note: This will not track the weights of nested `tf.Modules` that are
not
 |      themselves Keras layers.
 |
 |      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)
```

We initialise the callbacks to track the performance of our model on the separate test data. Callbacks are python objects, which are being called after every epoch (or batch step if you want). We will check for Ricci-flatness after every epoch and whether the MA-equation is satisfied. Recall that quantities are integrated over the manifold with Monte-Carlo integration:

$$\int_X \mathrm{d}\ \mathrm{vol}_{CY} f = \int_X \frac{\mathrm{d}\ \mathrm{vol}_{CY}}{\mathrm{d}A} \mathrm{d}A f = \frac{1}{N} \sum_i w_i f|_{p_i}$$

where w_i are the integration weights given by Shiffman and Zelditch. The established benchmarks are σ-measure and $\mathcal{R}$-measure which are defined as

$$\sigma = \frac{1}{\mathrm{vol}_{CY}} \int_X \left| 1 - \frac{\frac{J^3}{\mathrm{vol}_K}}{\frac{\Omega \wedge \bar{\Omega}}{\mathrm{vol}_{CY}}} \right|$$

and

$$||R|| = \frac{\mathrm{vol}_K^{\frac{1}{\text{n-fold}}}}{\mathrm{vol}_{CY}} \int_X |R| \, .$$

[17]:
```
rcb = RicciCallback((data['X_val'], data['y_val']), data['val_pullbacks'])
scb = SigmaCallback((data['X_val'], data['y_val']))
volkcb = VolkCallback((data['X_val'], data['y_val']))
cb_list = [rcb, scb, volkcb, PlotLearning()]
```

In the next step we define the hyperparameters of our neural net. We recall that the total loss was given by

$$\mathcal{L} = \alpha_1 \mathcal{L}_{\mathrm{MA}} + \alpha_2 \mathcal{L}_{\mathrm{dJ}} + \alpha_3 \mathcal{L}_{\mathrm{transition}} + \alpha_4 \mathcal{L}_{\mathrm{Ricci}} + \alpha_5 \mathcal{L}_{\mathrm{vol\text{-}K}}$$

which introduces some additional hyperparameters α_i to our model.

[18]:
```
nlayer = 3
nHidden = 64
act = 'gelu'
nEpochs = 10
bSize = 64
alpha = [1., 1., 1., 1., 1.]#MA, kähler, transition, Ricci, volume
nfold = 3
n_in = 2*pg.ncoords
n_out = 1# phi is a scalar
kappa = 1/np.mean(data['y_train'][:,-2])#mean of integration weights
```

The neural net can be set up with Keras, which is another high-level API using tensorflow in the backend.

[19]:
```
nn = tfk.Sequential()
nn.add(tfk.Input(shape=(n_in)))
for i in range(nlayer):
    nn.add(tfk.layers.Dense(nHidden, activation=act))
nn.add(tfk.layers.Dense(n_out, use_bias = False))
nn.summary()
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                832

 dense_1 (Dense)             (None, 64)                4160

 dense_2 (Dense)             (None, 64)                4160

 dense_3 (Dense)             (None, 1)                 64

=================================================================
Total params: 9,216
Trainable params: 9,216
Non-trainable params: 0
_________________________________________________________________
```

About 10k parameters which is roughly the same as the number of parameters in the hbalanced metric at k=3 (99x99-99) Donaldson.

We convert the basis from the *basis.pickle*-file to tensorflow tensors

[20]:
```
from cymetric.models.tfhelper import prepare_tf_basis
BASIS = prepare_tf_basis(BASIS)
```

and finally provide all this stuff as arguments to the *PhiFSModel*:

[21]:
```
phimodel = PhiFSModel(nn, BASIS, kappa=kappa, alpha=alpha)
```

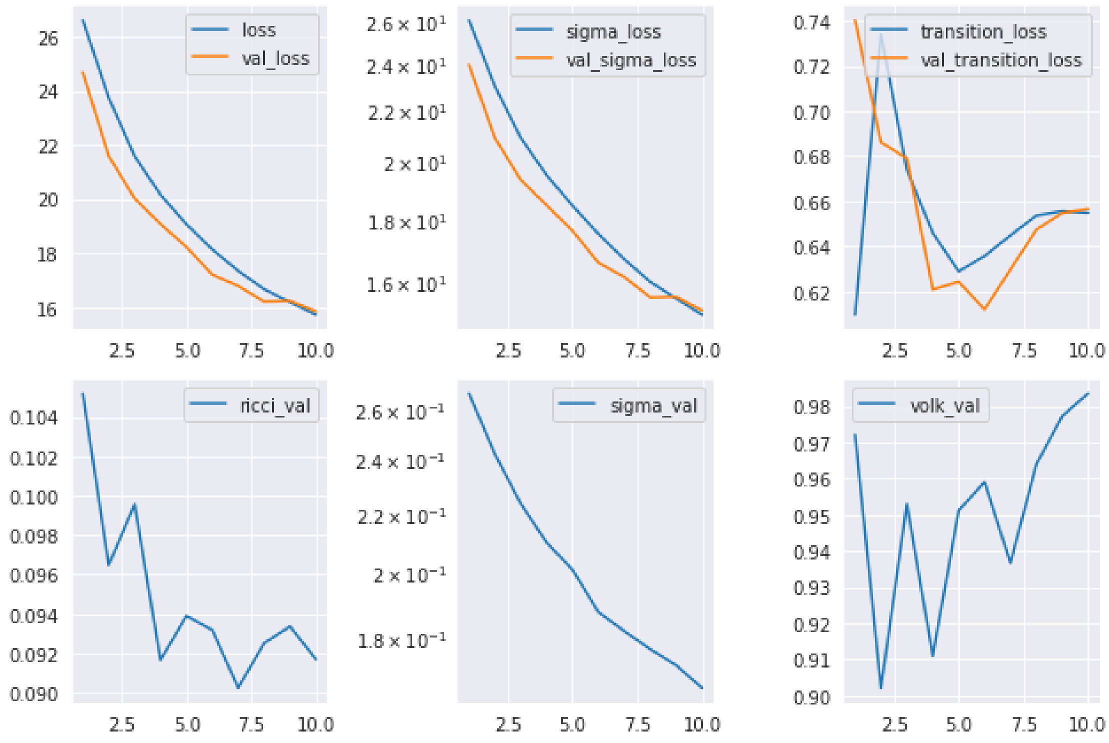
We compile the model, use Adam optimiser, introduce sample weights

[22]:
```
cmetrics = [TotalLoss(), SigmaLoss(), TransitionLoss()]
opt = tfk.optimizers.Adam()
phimodel.compile(custom_metrics = cmetrics, optimizer=opt)
sw = data['y_train'][:,0]
```

and, after all this effort, fit the model. Now the only thing that is left is watching the loss decrease. Every ML researcher's favourite activity.

[23]:
```
history = phimodel.fit(data['X_train'], data['y_train'],
                       epochs=nEpochs, batch_size=bSize,
                       validation_split=0.1, verbose=1,
```

```
                    callbacks=cb_list, sample_weight=sw)
```




```
975/975 [==============================] - 85s 88ms/step - loss: 15.7265 -
sigma_loss: 15.0716 - transition_loss: 0.6549 - val_loss: 15.8454 -
val_sigma_loss: 15.1889 - val_transition_loss: 0.6566 - ricci_val: 0.0917 -
sigma_val: 0.1660 - volk_val: 0.9836
```

1.3 Using the Metric

To get the metric at specific points x we just call the model with

[24]:
```
phimodel(data['X_val'][0:5])
```

[24]:
```
<tf.Tensor: shape=(5, 3, 3), dtype=complex64, numpy=
array([[[ 0.14943747-2.1827873e-10j, -0.01563393-4.3110428e-03j,
         -0.00962105-1.2034419e-02j],
        [-0.01563393+4.3110419e-03j,  0.20066918-8.3089829e-10j,
         -0.01484469-3.8823432e-03j],
        [-0.00962105+1.2034420e-02j, -0.01484469+3.8823434e-03j,
          0.17426145+7.5910811e-10j]],

       [[ 0.21871483-7.8400586e-10j,  0.05213756+3.5081454e-02j,
          0.02669672-7.9291999e-02j],
        [ 0.05213756-3.5081457e-02j,  0.22192816+1.8411191e-09j,
```

```
 -0.03742732-2.0945353e-02j],
[ 0.02669672+7.9291992e-02j, -0.03742732+2.0945355e-02j,
  0.20107032+1.6683718e-09j]],

[[ 0.25532487-1.8626451e-09j,  0.051225  -2.6202738e-02j,
  -0.01786895-4.4183038e-02j],
 [ 0.051225  +2.6202738e-02j,  0.20163487+1.4635930e-09j,
  -0.0135983 -7.9899179e-03j],
 [-0.01786895+4.4183031e-02j, -0.0135983 +7.9899170e-03j,
   0.17072347+1.8352735e-09j]],

[[ 0.18584731-5.7334546e-09j, -0.02462773-3.0770294e-02j,
   0.07919927+5.9154863e-03j],
 [-0.02462773+3.0770294e-02j,  0.17311345-1.2393772e-09j,
  -0.02587099+7.2506502e-02j],
 [ 0.07919926-5.9154881e-03j, -0.02587099-7.2506502e-02j,
   0.21203336-6.1901684e-10j]],

[[ 0.27743298+1.0150236e-09j,  0.0013    +9.6371248e-03j,
  -0.00960917-1.6544081e-02j],
 [ 0.00130001-9.6371267e-03j,  0.26463556-3.0856737e-10j,
  -0.02575707+3.5442214e-02j],
 [-0.00960917+1.6544081e-02j, -0.02575706-3.5442218e-02j,
   0.15399337-1.2318074e-09j]]], dtype=complex64)>
```

and have some numerical values.

1.4 Outlook

What's left to do? Here are some exercises for the listener:

1. Re-run the experiment with more points and more epochs.
2. Re-run the experiment with a different NN architecture. You can for example replace the dense network with some function ansatz for the Kähler potential such as the hbalanced metric on the section space. Maybe do some symbolic regression to get a symbolic (approximately) Ricci-flat metric.
3. Re-run the experiment with your favourite CY manifold.
4. Get involved. We welcome contributions. There is still a lot to be done, fixing Kähler class via integration over curves, writing an interface to CYtools, finding the ultimate nn-architecture, …