string_data_bicubic_cleared

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

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

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.

[ ]:
```
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 *CICYPointGenerator* or use the Mathematica package for the *ToricPointGenerator*.

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

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

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

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:

$-0.36859739831418x_1^3x_4^3+0.05273312318807256x_1^3x_4x_5x_6+0.3777578902423778x_1^3x_5^3-0.2276208490552241x_1^3x_6^3-0.5$

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.

[ ]:
```
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).

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

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

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

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

The dataset can subsequently be loaded with NumPy.

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

The training data contains real points

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

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.

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

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?

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

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.

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

We inspect this data by loading it from the file

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

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.

```
[ ]: 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

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

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

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

[ ]:
```
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

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

which introduces some additional hyperparameters α_i to our model.

[ ]:
```
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.

[ ]:
```
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()
```

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

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

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

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

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

[ ]:
```
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.

[ ]:
```
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)
```

1.3 Using the Metric

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

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

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, ...