deel-lip

Release 1.0.0

Mathieu Serrurier (mathieu.serrurier@irt-saintexupery.com), Franc

Mar 01, 2021

CONTENTS:

1 The library contains: 3

2 Example and usage 5

3 Installation 7

4 Cite this work 9

5 Contributing 11

6 Indices and tables 13

6.1 Example and usage . . . 13
6.1.1 Which layers are safe to use? . . . 13
6.1.2 How to use it? . . . 14
6.2 Demo 1: Wasserstein distance estimation on toy example . . . 15
6.2.1 Parameters input images . . . 15
6.2.2 Generate images . . . 15
6.2.3 Expe parameters . . . 20
6.2.3.1 KR dual formulation . . . 20
6.2.3.2 Build lipschitz Model . . . 20
6.2.3.3 Learn on toy dataset . . . 21
6.3 Demo 2: HKR Classifier on toy dataset . . . 22
6.3.1 Parameters . . . 23
6.3.2 Relation with optimal transport . . . 24
6.3.3 KR dual formulation . . . 24
6.3.4 Hinge-KR classification . . . 24
6.3.4.1 HKR-Classifier . . . 24
6.3.4.2 Build lipschitz Model . . . 25
6.3.4.3 Learn classification on toy dataset . . . 26
6.3.4.4 Plot output countour line . . . 27
6.3.4.5 Transfer network to a classical MLP and compare outputs . . . 28
6.4 Demo 3: HKR classifier on MNIST dataset . . . 30
6.4.1 data preparation . . . 31
6.4.2 Build lipschitz Model . . . 32
6.4.3 Learn classification on MNIST . . . 33

Controlling the Lipschitz constant of a layer or a whole neural network has many applications ranging from adversarial robustness to Wasserstein distance estimation.

This library provides an efficient implementation of **k-Lispchitz layers for keras**.

CHAPTER

ONE

THE LIBRARY CONTAINS:

- k-Lipschitz variant of keras layers such as `Dense`, `Conv2D` and `Pooling`,
- activation functions compatible with `keras`,
- kernel initializers and kernel constraints for `keras`,
- loss functions that make use of Lipschitz constrained networks (see our paper for more information),
- tools to monitor the singular values of kernels during training,
- tools to convert k-Lipschitz network to regular network for faster inference.

CHAPTER

TWO

EXAMPLE AND USAGE

In order to make things simple the following rules have been followed during development: - `deel-lip` follows the `keras` package structure. - All elements (layers, activations, initializers, ...) are compatible with standard the `keras` elements. - When a k-Lipschitz layer overrides a standard keras layer, it uses the same interface and the same parameters. The only difference is a new parameter to control the Lipschitz constant of a layer.

Here is an example showing how to build and train a 1-Lipschitz network:

```
from deel.lip.layers import SpectralDense, SpectralConv2D, ScaledL2NormPooling2D
from deel.lip.model import Sequential
from deel.lip.activations import GroupSort
from deel.lip.losses import HKR_multiclass_loss
from tensorflow.keras.layers import Input, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np

# Sequential (resp Model) from deel.model has the same properties as any lipschitz␣
↪model.
# It act only as a container, with features specific to lipschitz
# functions (condensation, vanilla_exportation...)
model = Sequential(
    [
        Input(shape=(28, 28, 1)),
        # Lipschitz layers preserve the API of their superclass ( here Conv2D )
        # an optional param is available: k_coef_lip which control the lipschitz
        # constant of the layer
        SpectralConv2D(
            filters=16,
            kernel_size=(3, 3),
            activation=GroupSort(2),
            use_bias=False,
            kernel_initializer="orthogonal",
        ),
        # usual pooling layer are implemented (avg, max...), but new layers are also␣
↪available
        ScaledL2NormPooling2D(pool_size=(2, 2), data_format="channels_last"),
        SpectralConv2D(
            filters=32,
            kernel_size=(3, 3),
            activation=GroupSort(2),
            use_bias=False,
            kernel_initializer="orthogonal",
        ),
```

(continues on next page)

(continued from previous page)

```
        ScaledL2NormPooling2D(pool_size=(2, 2), data_format="channels_last"),
        # our layers are fully interoperable with existing keras layers
        Flatten(),
        SpectralDense(
            100,
            activation=GroupSort(2),
            use_bias=False,
            kernel_initializer="orthogonal",
        ),
        SpectralDense(
            10, activation=None, use_bias=False, kernel_initializer="orthogonal"
        ),
    ],
    # similary model has a parameter to set the lipschitz constant
    # to set automatically the constant of each layer
    k_coef_lip=1.0,
    name="hkr_model",
)

# HKR (Hinge-Krantorovich-Rubinstein) optimize robustness along with accuracy
model.compile(
    loss=HKR_multiclass_loss(alpha=5.0, min_margin=0.5),
    optimizer=Adam(lr=0.01),
    metrics=["accuracy"],
)

model.summary()

# load data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# standardize and reshape the data
x_train = np.expand_dims(x_train, -1)
mean = x_train.mean()
std = x_train.std()
x_train = (x_train - mean) / std
x_test = np.expand_dims(x_test, -1)
x_test = (x_test - mean) / std
# one hot encode the labels
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# fit the model
model.fit(
    x_train,
    y_train,
    batch_size=256,
    epochs=15,
    validation_data=(x_test, y_test),
    shuffle=True,
)

# once training is finished you can convert
# SpectralDense layers into Dense layers and SpectralConv2D into Conv2D
# which optimize performance for inference
vanilla_model = model.vanilla_export()
```

See the full documentation for a complete API description.

CHAPTER

THREE

INSTALLATION

You can install `deel-lip` directly from pypi:

```
pip install deel-lip
```

In order to use `deel-lip`, you also need a valid tensorflow installation. `deel-lip` supports tensorflow versions 2.x

CHAPTER

FOUR

CITE THIS WORK

This library has been built to support the work presented in the paper Achieving robustness in classification using optimaltransport with Hinge regularization which aim provable and efficient robustness by design.

This work can be cited as:

```
@misc{2006.06520,
Author = {Mathieu Serrurier and Franck Mamalet and Alberto González-Sanz and Thibaut␣
↪Boissin and Jean-Michel Loubes and Eustasio del Barrio},
Title = {Achieving robustness in classification using optimal transport with hinge␣
↪regularization},
Year = {2020},
Eprint = {arXiv:2006.06520},
}
```

CHAPTER

FIVE

CONTRIBUTING

To contribute, you can open an issue, or fork this repository and then submit changes through a pull-request. We use black to format the code and follow PEP-8 convention. To check that your code will pass the lint-checks, you can run:

```
tox -e py36-lint
```

You need tox in order to run this. You can install it via `pip`:

```
pip install tox
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

6.1 Example and usage

In order to make things simple the following rules have been followed during development:

- `deel-lip` follows the `keras` package structure.
- All elements (layers, activations, initializers, ...) are compatible with standard the `keras` elements.
- When a k-Lipschitz layer overrides a standard keras layer, it uses the same interface and the same parameters. The only difference is a new parameter to control the Lipschitz constant of a layer.

6.1.1 Which layers are safe to use?

The following table indicates which layers are safe to use in a Lipshitz network, and which are not.

layer	1-lip?	deel-lip equivalent	comments
`Dense`	no	`SpectralDense` `FrobeniusDense`	`SpectralDense` and `FrobeniusDense` are similar when there is a single output.
`Conv2D`	no	`SpectralConv2D` `FrobeniusConv2D`	`SpectralConv2D` also implements Björck normalization.
`MaxPoolingGlobalMaxPooling`	yes	n/a	
`AveragePooling2DGlobalAverageScaledAveragePooling2DScaledGlobalAveragePooling2D`	no	`ScaledAveragePooling2D`	The lipschitz constant is bounded by `sqrt(pool_h * pool_h)`.
`Flatten`	yes	n/a	
`Dropout`	no	None	The lipschitz constant is bounded by the dropout factor.
`BatchNorm`	no	None	We suspect that layer normalization already limits internal covariate shift.

6.1.2 How to use it?

Here is a simple example showing how to build a 1-Lipschitz network:

```
from deel.lip.initializers import BjorckInitializer
from deel.lip.layers import SpectralDense, SpectralConv2D
from deel.lip.model import Sequential
from deel.lip.activations import PReLUlip
from tensorflow.keras.layers import Input, Lambda, Flatten, MaxPool2D
from tensorflow.keras import backend as K
from tensorflow.keras.optimizers import Adam

# Sequential (resp Model) from deel.model has the same properties as any lipschitz
# layer ( condense, setting of the lipschitz factor etc...). It act only as a␣
↪container.
model = Sequential(
    [
        Input(shape=(28, 28)),
        Lambda(lambda x: K.reshape(x, (-1, 28, 28, 1))),

        # Lipschitz layer preserve the API of their superclass ( here Conv2D )
        # an optional param is available: k_coef_lip which control the lipschitz
        # constant of the layer
        SpectralConv2D(
            filters=32, kernel_size=(3, 3), padding='same',
            activation=PReLUlip(), data_format='channels_last',
            kernel_initializer=BjorckInitializer(15, 50)),
        SpectralConv2D(
            filters=32, kernel_size=(3, 3), padding='same',
            activation=PReLUlip(), data_format='channels_last',
            kernel_initializer=BjorckInitializer(15, 50)),
        MaxPool2D(pool_size=(2, 2), data_format='channels_last'),

        SpectralConv2D(
            filters=64, kernel_size=(3, 3), padding='same',
            activation=PReLUlip(), data_format='channels_last',
            kernel_initializer=BjorckInitializer(15, 50)),
        SpectralConv2D(
            filters=64, kernel_size=(3, 3), padding='same',
            activation=PReLUlip(), data_format='channels_last',
            kernel_initializer=BjorckInitializer(15, 50)),
        MaxPool2D(pool_size=(2, 2), data_format='channels_last'),

        Flatten(),
        SpectralDense(256, activation="relu", kernel_initializer=BjorckInitializer(15,
↪ 50)),
        SpectralDense(10, activation="softmax"),
    ],
    k_coef_lip=0.5,
    name='testing'
)

optimizer = Adam(lr=0.001)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

See deel-lip-api for a complete API description.

6.2 Demo 1: Wasserstein distance estimation on toy example

In this notebook we will see how to estimate the wasserstein distance with a Neural net by using the Kantorovich-Rubinestein dual representation.

```
from datetime import datetime
import os
import numpy as np
import math

import matplotlib.pyplot as plt

from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Flatten, ReLU
from tensorflow.keras.optimizers import Adam

from deel.lip.layers import SpectralConv2D, SpectralDense, FrobeniusDense
from deel.lip.activations import MaxMin, GroupSort, FullSort
from deel.lip.utils import load_model
from deel.lip.losses import KR_loss
from deel.lip.model import Model

from model_samples.model_samples import get_lipMLP
```

6.2.1 Parameters input images

The synthetic dataset will be composed image with black or white squares allowing us to check if the computed wasserstein distance is correct.

```
img_size = 64
frac_value = 0.3  # proportion of the center square
```

6.2.2 Generate images

```
def generate_toy_images(shape,frac=0,v=1):
    """
    function that generate a single image.

    Args:
        shape: shape of the output image
        frac: proportion of the center square
        value: value assigned to the center square
    """
    img = np.zeros(shape)
    if frac==0:
        return img
    frac=frac**0.5
    #print(frac)
    l=int(shape[0]*frac)
    ldec=(shape[0]-l)//2
    #print(l)
    w=int(shape[1]*frac)
    wdec=(shape[1]-w)//2
```

(continues on next page)

(continued from previous page)

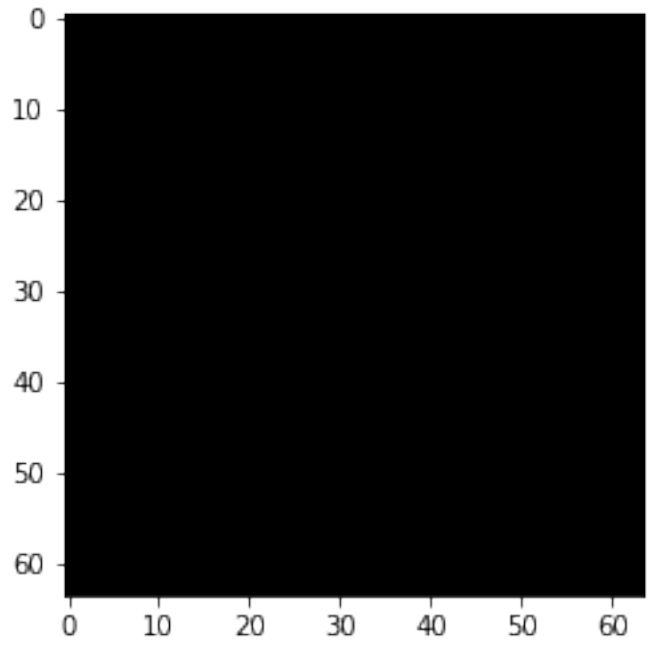
```
    img[ldec:ldec+l,wdec:wdec+w,:]=v
    return img


def binary_generator(batch_size,shape,frac=0):
    """
    generate a batch with half of black images, hald of images with a white square.
    """
    batch_x = np.zeros(((batch_size,)+(shape)), dtype=np.float16)
    batch_y=np.zeros((batch_size,1), dtype=np.float16)
    batch_x[batch_size//2:,]=generate_toy_images(shape,frac=frac,v=1)
    batch_y[batch_size//2:]=1
    while True:
        yield  batch_x, batch_y


def ternary_generator(batch_size,shape,frac=0):
    """
    Same as binary generator, but images can have a white square of value 1, or value␣
↪-1
    """
    batch_x = np.zeros(((batch_size,)+(shape)), dtype=np.float16)
    batch_y=np.zeros((batch_size,1), dtype=np.float16)
    batch_x[3*batch_size//4:,]=generate_toy_images(shape,frac=frac,v=1)
    batch_x[batch_size//2:3*batch_size//4,]=generate_toy_images(shape,frac=frac,v=-1)
    batch_y[batch_size//2:]=1
    #indexes_shuffle = np.arange(batch_size)
    while True:
        #np.random.shuffle(indexes_shuffle)
        #yield  batch_x[indexes_shuffle,], batch_y[indexes_shuffle,]
        yield  batch_x, batch_y
```

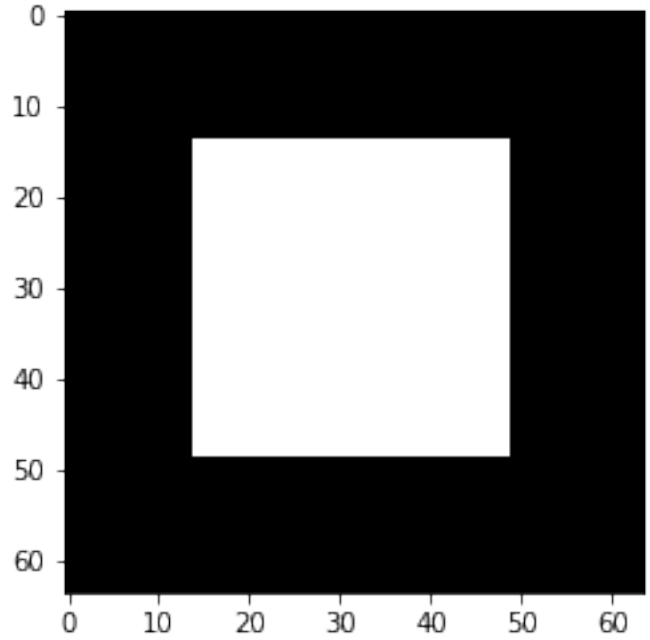
```
def display_img(img):
    """
    Display an image
    """
    if img.shape[-1] == 1:
        img = np.tile(img,(3,))
    fig, ax = plt.subplots()

    imgplot = ax.imshow((img*255).astype(np.uint))
```

Now let's take a look at the generated batches

```
test=binary_generator(2,(img_size,img_size,1),frac=frac_value)
imgs, y=next(test)

display_img(imgs[0])
display_img(imgs[1])
print("Norm L2 "+str(np.linalg.norm(imgs[1])))
print("Norm L2(count pixels) "+str(math.sqrt(np.size(imgs[1][imgs[1]==1]))))
```

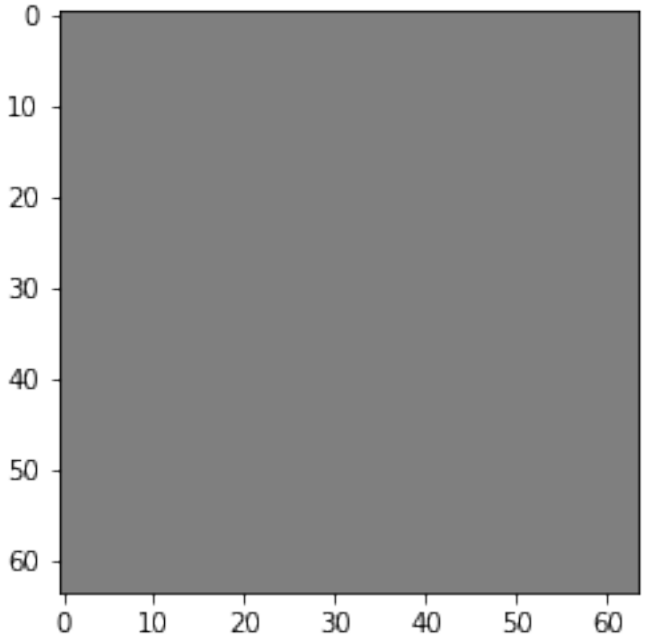
```
Norm L2 35.0
Norm L2(count pixels) 35.0
```

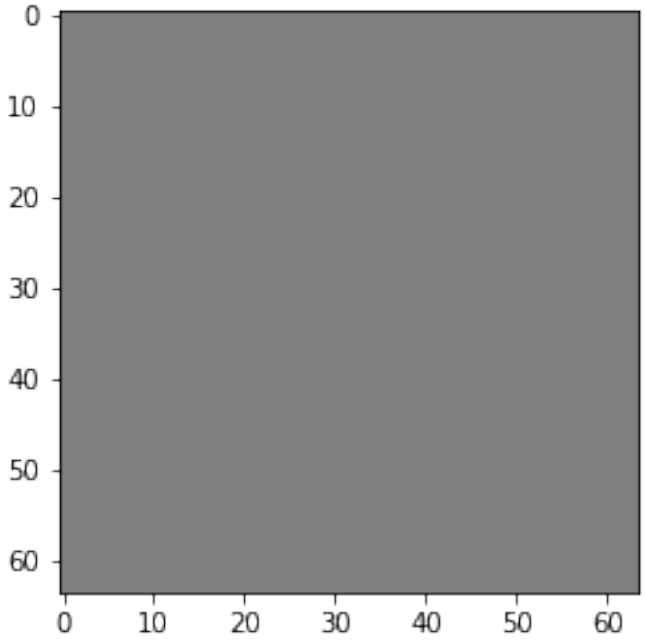


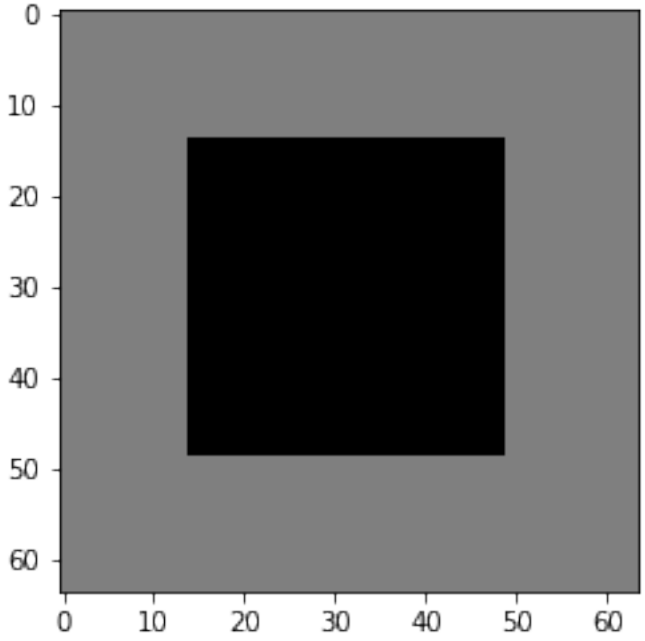


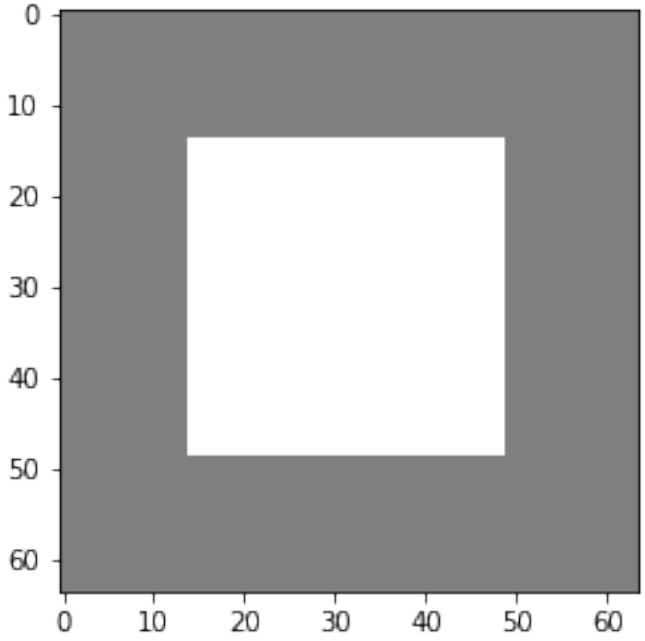

```
test=ternary_generator(4,(img_size,img_size,1),frac=frac_value)
imgs, y=next(test)

for i in range(4):
    display_img(0.5*(imgs[i]+1.0)) # we ensure that there is no negative value wehn␣
↪displaying images

print("Norm L2(imgs[2]-imgs[0])"+str(np.linalg.norm(imgs[2]-imgs[0])))
print("Norm L2(imgs[2]) "+str(np.linalg.norm(imgs[2])))
print("Norm L2(count pixels) "+str(math.sqrt(np.size(imgs[2][imgs[2]==-1]))))
```

```
Norm L2(imgs[2]-imgs[0])35.0
Norm L2(imgs[2]) 35.0
Norm L2(count pixels) 35.0
```






0
10
20
30
40
50
60
0
10
20
30
40
50
60


0
10
20
30
40
50
60
0
10
20
30
40
50
60

6.2.3 Expe parameters

Now we know the wasserstein distance between the black image and the images with a square on it. For both binary generator and ternary generator this distance is 35.

We will then compute this distance using a neural network.

6.2.3.1 KR dual formulation

In our setup, the KR dual formulation is stated as following:

$$W_1(\mu,\nu) = \sup_{f \in Lip_1(\Omega)} \mathop{\mathbb{E}}_{\mathbf{x}\sim\mu}[f(\mathbf{x})] - \mathop{\mathbb{E}}_{\mathbf{x}\sim\nu}[f(\mathbf{x})]$$

This state the problem as an optimization problem over the 1-lipschitz functions. Therefore k-Lipschitz networks allows us to solve this maximization problem.

[1] C. Anil, J. Lucas, et R. Grosse, « Sorting out Lipschitz function approximation », arXiv:1811.05381 [cs, stat], nov. 2018.

```
batch_size=64
epochs=5
steps_per_epoch=6400
```

```
generator = ternary_generator   #binary_generator, ternary_generator
activation = FullSort #ReLU, MaxMin, GroupSort
```

6.2.3.2 Build lipschitz Model

```
K.clear_session()
wass=get_lipMLP((img_size,img_size,1), hidden_layers_size = [128,64,32] ,
↪activation=activation, nb_classes = 1,kCoefLip=1.0)
## please note that the previous helper function has the same behavior as the␣
↪following code:
# inputs = Input((img_size, img_size, 1))
# x = SpectralDense(128, activation=FullSort())(inputs)
# x = SpectralDense(64, activation=FullSort())(x)
# x = SpectralDense(32, activation=FullSort())(x)
# y = FrobeniusDense(1, activation=None)(x)
# wass = Model(inputs=inputs, outputs=y)
wass.summary()
```

```
128
64
32
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 64, 64, 1)]       0
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
spectral_dense (SpectralDens (None, 128)               524545
_________________________________________________________________
```

(continues on next page)

(continued from previous page)

```
full_sort (FullSort)         (None, 128)               0
_________________________________________________________________
spectral_dense_1 (SpectralDe (None, 64)                8321
_________________________________________________________________
full_sort_1 (FullSort)       (None, 64)                0
_________________________________________________________________
spectral_dense_2 (SpectralDe (None, 32)                2113
_________________________________________________________________
full_sort_2 (FullSort)       (None, 32)                0
_________________________________________________________________
frobenius_dense (FrobeniusDe (None, 1)                 33
=================================================================
Total params: 535,012
Trainable params: 534,785
Non-trainable params: 227
_________________________________________________________________
```

```
optimizer = Adam(lr=0.01)
```

```
wass.compile(loss=KR_loss(), optimizer=optimizer, metrics=[KR_loss()])
```

6.2.3.3 Learn on toy dataset

```
wass.fit_generator( generator(batch_size,(img_size,img_size,1),frac=frac_value),
                steps_per_epoch=steps_per_epoch// batch_size,
                epochs=epochs,verbose=1)
```

```
WARNING:tensorflow:From <ipython-input-12-b25f21272064>:3: Model.fit_generator (from␣
↪tensorflow.python.keras.engine.training) is deprecated and will be removed in a␣
↪future version.
Instructions for updating:
Please use Model.fit, which supports generators.
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
Train for 100 steps
Epoch 1/5
100/100 [==============================] - 17s 166ms/step - loss: -33.9067 - KR_loss_
↪fct: -33.9067
Epoch 2/5
100/100 [==============================] - 17s 172ms/step - loss: -34.9944 - KR_loss_
↪fct: -34.99443s - loss: -34.9944 - KR
Epoch 3/5
100/100 [==============================] - 18s 180ms/step - loss: -34.9941 - KR_loss_
↪fct: -34.9941
Epoch 4/5
100/100 [==============================] - 18s 177ms/step - loss: -34.9942 - KR_loss_
↪fct: -34.9942
Epoch 5/5
100/100 [==============================] - 18s 177ms/step - loss: -34.9942 - KR_loss_
↪fct: -34.9942
```

```
<tensorflow.python.keras.callbacks.History at 0x14adcc6c088>
```

As we can see the loss converge to the value 35 which is the wasserstein distance between the two distributions (square and non-square).

6.3 Demo 2: HKR Classifier on toy dataset

In this demo notebook we will show how to build a robust classifier based on the regularized version of the Kantorovitch-Rubinstein duality. We will perform this on the `two moons` synthetic dataset.

```
from datetime import datetime
import os
import numpy as np
import math

from sklearn.datasets import make_moons, make_circles  # the synthetic dataset

import matplotlib.pyplot as plt
import seaborn as sns

from wasserstein_utils.otp_utils import otp_generator  # keras generator of the␣
↪synthetic dataset

# in order to build our classifier we will use element from tensorflow along with
# layers from deel-lip
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.keras.layers import ReLU, Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.metrics import binary_accuracy

from deel.lip.model import Model  # use of deel.lip is not mandatory but offers the␣
↪vanilla_export feature
from deel.lip.layers import SpectralConv2D, SpectralDense, FrobeniusDense
from deel.lip.activations import MaxMin, GroupSort, FullSort, GroupSort2
from deel.lip.utils import load_model  # wrapper that avoid manually specifying␣
↪custom_objects field
from deel.lip.losses import HKR_loss, KR_loss, hinge_margin_loss  # custom losses for␣
↪HKR robust classif
from deel.lip.normalizers import DEFAULT_NITER_BJORCK, DEFAULT_NITER_SPECTRAL

from model_samples.model_samples import get_lipMLP  # helper to quickly build an HKR␣
↪classifier
```

6.3.1 Parameters

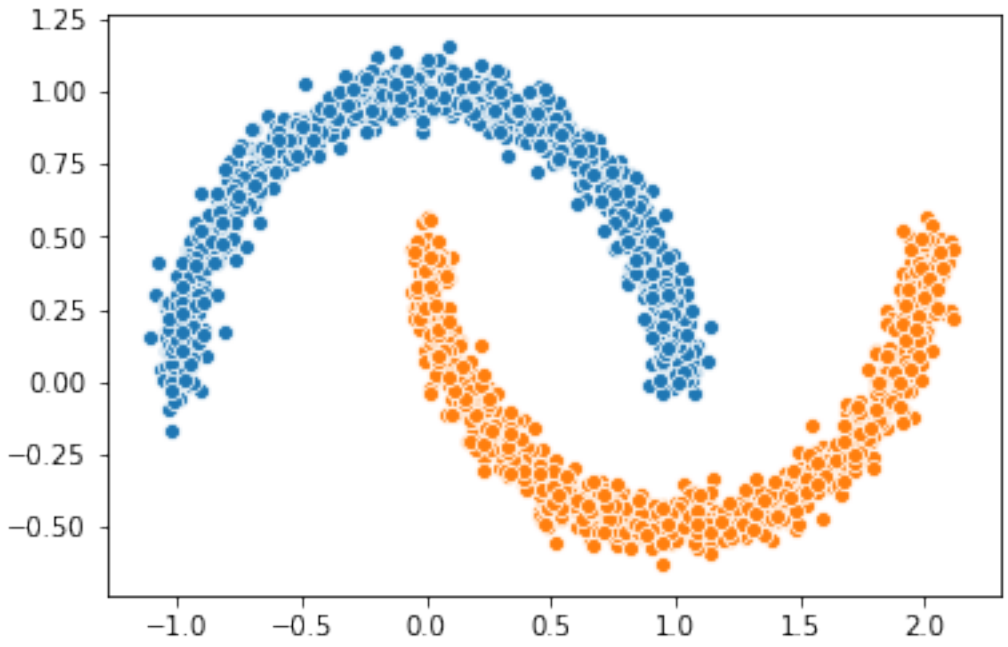
Let's first construct our two moons dataset

```
circle_or_moons = 1  # 0 for circle , 1 for moons
n_samples=5000  # number of sample in the dataset
noise=0.05  # amount of noise to add in the data. Tested with 0.14 for circles 0.05␣
↪for two moons
factor=0.4  # scale factor between the inner and the outer circle
```

```
if circle_or_moons == 0:
    X,Y=make_circles(n_samples=n_samples,noise=noise,factor=factor)
else:
    X,Y=make_moons(n_samples=n_samples,noise=noise)

# When working with the HKR-classifier, using labels {-1, 1} instead of {0, 1} is␣
↪advised.
# This will be explained further on
Y[Y==1]=-1
Y[Y==0]=1
```

```
X1=X[Y==1]
X2=X[Y==-1]
sns.scatterplot(X1[:1000,0],X1[:1000,1])
sns.scatterplot(X2[:1000,0],X2[:1000,1])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2c40c220208>
```



6.3.2 Relation with optimal transport

In this setup we can solve the optimal transport problem between the distribution of `X[Y==1]` and `X[Y==-1]`. This usually require to match each element of the first distribution with an element of the second distribution such that this minimize a global cost. In our setup this cost is the $ l_1 $ distance, which will allow us to make use of the KR dual formulation. The overall cost is then the W_1 distance.

However the W_1 distance is known to be untractable in general.

6.3.3 KR dual formulation

In our setup, the KR dual formulation is stated as following:

$$W_1(\mu, \nu) = \sup_{f \in Lip_1(\Omega)} \mathop{\mathbb{E}}_{\mathbf{x} \sim \mu} [f(\mathbf{x})] - \mathop{\mathbb{E}}_{\mathbf{x} \sim \nu} [f(\mathbf{x})]$$

This state the problem as an optimization problem over the 1-lipschitz functions. Therefore k-Lipschitz networks allows us to solve this maximization problem.

6.3.4 Hinge-KR classification

When dealing with W_1 one may note that many functions maximize the maximization problem described above. Also we want this function to be meaningfull in terms of classification. To do so, we want f to be centered in 0, which can be done without altering the inital problem. By doing so we can use the obtained function for binary classification, by looking at the sign of f.

In order to enforce this, we will add a Hinge term to the loss. It has been shown that this new problem is still a optimal transport problem and that this problem admit a meaningfull optimal solution.

6.3.4.1 HKR-Classifier

Now we will show how to build a binary classifier based on the regularized version of the KR dual problem.

In order to ensure the 1-Lipschitz constraint `deel-lip` uses spectral normalization. These layers also can also use Bjork orthonormalization to ensure that the gradient of the layer is 1 almost everywhere. Experiment shows that the optimal solution lie in this sub-class of functions.

```
batch_size=256
steps_per_epoch=40480
epoch=10
hidden_layers_size = [256,128,64]  # stucture of the network
niter_spectral = DEFAULT_NITER_SPECTRAL
niter_bjorck = DEFAULT_NITER_BJORCK
activation = FullSort  # other lipschitz activation are ReLU, MaxMin, GroupSort2,␣
↪GroupSort
min_margin= 0.29  # minimum margin to enforce between the values of f for each class
```

```
# build data generator
gen=otp_generator(batch_size,X,Y)
```

6.3.4.2 Build lipschitz Model

Let's build our model now.

```
K.clear_session()
wass=get_lipMLP(
    (2,),
    hidden_layers_size = hidden_layers_size,
    activation=activation,
    nb_classes = 1,
    kCoefLip=1.0,
    niter_spectral = niter_spectral,
    niter_bjorck = niter_bjorck
)
## please note that calling the previous helper function has the exact
## same effect as the following code:
# inputs = Input((2,))
# x = SpectralDense(256, activation=FullSort(),
#                   niter_spectral=niter_spectral,
#                   niter_bjorck=niter_bjorck)(inputs)
# x = SpectralDense(128, activation=FullSort(),
#                   niter_spectral=niter_spectral,
#                   niter_bjorck=niter_bjorck)(x)
# x = SpectralDense(64, activation=FullSort(),
#                   niter_spectral=niter_spectral,
#                   niter_bjorck=niter_bjorck)(x)
# y = FrobeniusDense(1, activation=None)(x)
# wass = Model(inputs=inputs, outputs=y)
wass.summary()
```

```
256
128
64
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 2)]               0
_________________________________________________________________
flatten (Flatten)            (None, 2)                 0
_________________________________________________________________
spectral_dense (SpectralDens (None, 256)               1025
_________________________________________________________________
full_sort (FullSort)         (None, 256)               0
_________________________________________________________________
spectral_dense_1 (SpectralDe (None, 128)               33025
_________________________________________________________________
full_sort_1 (FullSort)       (None, 128)               0
_________________________________________________________________
spectral_dense_2 (SpectralDe (None, 64)                8321
_________________________________________________________________
full_sort_2 (FullSort)       (None, 64)                0
_________________________________________________________________
frobenius_dense (FrobeniusDe (None, 1)                 65
=================================================================
Total params: 42,436
Trainable params: 41,985
```

(continues on next page)

(continued from previous page)

```
Non-trainable params: 451
_________________________________________________________________
```

As we can see the network has a gradient equal to 1 almost everywhere as all the layers respect this property.

It is good to note that the last layer is a `FrobeniusDense` this is because, when we have a single output, it become equivalent to normalize the frobenius norm and the spectral norm (as we only have a single singular value)

```
optimizer = Adam(lr=0.01)
```

```
# as the output of our classifier is in the real range [-1, 1], binary accuracy must␣
↪be redefined
def HKR_binary_accuracy(y_true, y_pred):
    S_true= tf.dtypes.cast(tf.greater_equal(y_true[:,0], 0),dtype=tf.float32)
    S_pred= tf.dtypes.cast(tf.greater_equal(y_pred[:,0], 0),dtype=tf.float32)
    return binary_accuracy(S_true,S_pred)
```

```
wass.compile(
    loss=HKR_loss(alpha=10,min_margin=min_margin),  # HKR stands for the hinge␣
↪regularized KR loss
    metrics=[
        KR_loss((-1,1)),  # shows the KR term of the loss
        hinge_margin_loss(min_margin=min_margin),  # shows the hinge term of the loss
        HKR_binary_accuracy  # shows the classification accuracy
    ],
    optimizer=optimizer
)
```

6.3.4.3 Learn classification on toy dataset

Now we are ready to learn the classification task on the two moons dataset.

```
wass.fit_generator(
    gen,
    steps_per_epoch=steps_per_epoch // batch_size,
    epochs=epoch,
    verbose=1
)
```

```
WARNING:tensorflow:From <ipython-input-11-258ce98fe6fe>:5: Model.fit_generator (from␣
↪tensorflow.python.keras.engine.training) is deprecated and will be removed in a␣
↪future version.
Instructions for updating:
Please use Model.fit, which supports generators.
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
Train for 158 steps
Epoch 1/10
158/158 [==============================] - 5s 30ms/step - loss: -0.3610 - KR_loss_
↪fct: -0.9315 - hinge_margin_fct: 0.0571 - HKR_binary_accuracy: 0.9176 4s - loss: 0.
↪1094 - KR_loss_fct: -0.8685 - hinge_marg
Epoch 2/10
```

(continues on next page)

(continued from previous page)

```
158/158 [==============================] - 2s 15ms/step - loss: -0.8084 - KR_loss_
↪fct: -0.9796 - hinge_margin_fct: 0.0171 - HKR_binary_accuracy: 0.9890
Epoch 3/10
158/158 [==============================] - 2s 15ms/step - loss: -0.8202 - KR_loss_
↪fct: -0.9858 - hinge_margin_fct: 0.0166 - HKR_binary_accuracy: 0.9895
Epoch 4/10
158/158 [==============================] - 2s 15ms/step - loss: -0.8313 - KR_loss_
↪fct: -0.9949 - hinge_margin_fct: 0.0164 - HKR_binary_accuracy: 0.9894
Epoch 5/10
158/158 [==============================] - 3s 17ms/step - loss: -0.8239 - KR_loss_
↪fct: -0.9818 - hinge_margin_fct: 0.0158 - HKR_binary_accuracy: 0.9903
Epoch 6/10
158/158 [==============================] - 3s 18ms/step - loss: -0.8229 - KR_loss_
↪fct: -0.9896 - hinge_margin_fct: 0.0167 - HKR_binary_accuracy: 0.9891
Epoch 7/10
158/158 [==============================] - 3s 19ms/step - loss: -0.8361 - KR_loss_
↪fct: -0.9911 - hinge_margin_fct: 0.0155 - HKR_binary_accuracy: 0.9908
Epoch 8/10
158/158 [==============================] - 3s 19ms/step - loss: -0.8333 - KR_loss_
↪fct: -0.9941 - hinge_margin_fct: 0.0161 - HKR_binary_accuracy: 0.9899
Epoch 9/10
158/158 [==============================] - 3s 19ms/step - loss: -0.8315 - KR_loss_
↪fct: -0.9945 - hinge_margin_fct: 0.0163 - HKR_binary_accuracy: 0.9895
Epoch 10/10
158/158 [==============================] - 3s 20ms/step - loss: -0.8438 - KR_loss_
↪fct: -0.9913 - hinge_margin_fct: 0.0147 - HKR_binary_accuracy: 0.9925
```

```
<tensorflow.python.keras.callbacks.History at 0x2c40d92c388>
```

6.3.4.4 Plot output countour line

As we can see the classifier get a pretty good accuracy. Let's now take a look at the learnt function. As we are in the 2D space, we can draw a countour plot to visualize f.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
batch_size=1024

x = np.linspace(X[:,0].min()-0.2, X[:,0].max()+0.2, 120)
y = np.linspace(X[:,1].min()-0.2, X[:,1].max()+0.2,120)
xx, yy = np.meshgrid(x, y, sparse=False)
X_pred=np.stack((xx.ravel(),yy.ravel()),axis=1)
```

```
# make predictions of f
pred=wass.predict(X_pred)

Y_pred=pred
Y_pred=Y_pred.reshape(x.shape[0],y.shape[0])
```

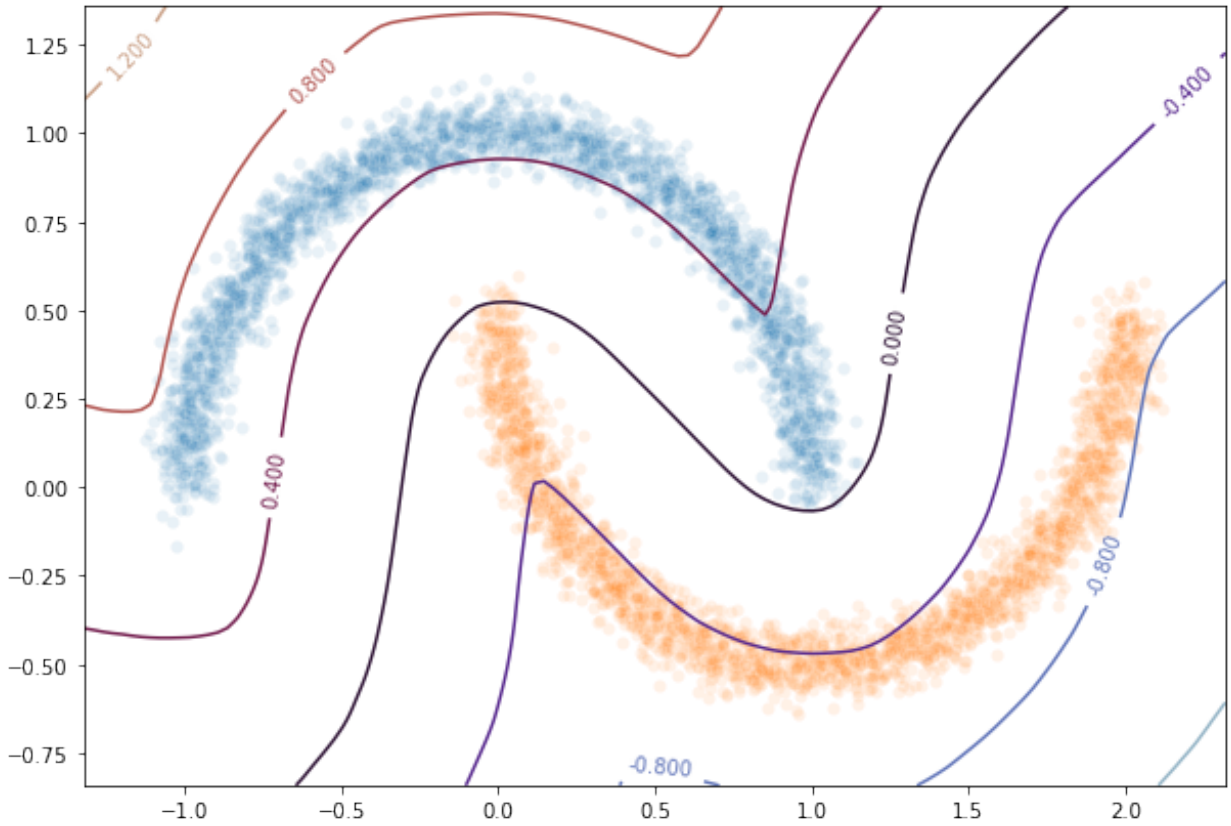
```
#plot the results
fig = plt.figure(figsize=(10,7))
ax1 = fig.add_subplot(111)
```

(continues on next page)

(continued from previous page)

```
sns.scatterplot(X[Y==1,0],X[Y==1,1],alpha=0.1,ax=ax1)
sns.scatterplot(X[Y==-1,0],X[Y==-1,1],alpha=0.1,ax=ax1)
cset =ax1.contour(xx,yy,Y_pred,cmap='twilight')
ax1.clabel(cset, inline=1, fontsize=10)
```

```
<a list of 7 text.Text objects>
```



6.3.4.5 Transfer network to a classical MLP and compare outputs

As we saw, our networks use custom layers in order to constrain training. However during inference layers behave exactly as regular `Dense` or `Conv2d` layers. Deel-lip has a functionnality to export a model to it's vanilla keras equivalent. Making it more convenient for inference.

```
from deel.lip.model import vanillaModel
## this is equivalent to test2 = wass.vanilla_export()
test2 = vanillaModel(wass)
test2.summary()
```

```
tensor input shape (None, 2)
tensor input shape (None, 2)
tensor input shape (None, 2)
tensor input shape (None, 256)
256
tensor input shape (None, 256)
```

(continues on next page)

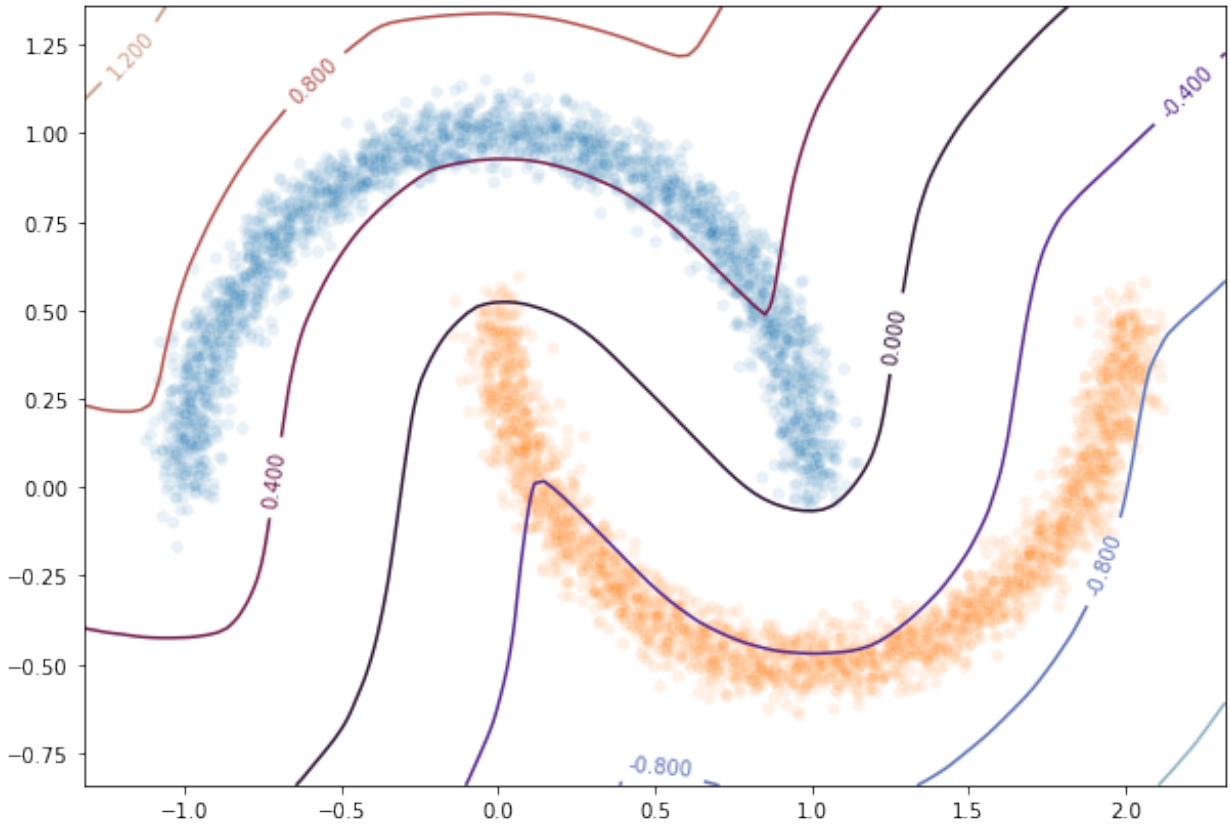
(continued from previous page)

```
tensor input shape (None, 128)
128
tensor input shape (None, 128)
tensor input shape (None, 64)
64
tensor input shape (None, 64)
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 2)]               0
_________________________________________________________________
flatten (Flatten)            (None, 2)                 0
_________________________________________________________________
spectral_dense (Dense)       (None, 256)               768
_________________________________________________________________
full_sort (FullSort)         (None, 256)               0
_________________________________________________________________
spectral_dense_1 (Dense)     (None, 128)               32896
_________________________________________________________________
full_sort_1 (FullSort)       (None, 128)               0
_________________________________________________________________
spectral_dense_2 (Dense)     (None, 64)                8256
_________________________________________________________________
full_sort_2 (FullSort)       (None, 64)                0
_________________________________________________________________
frobenius_dense (Dense)      (None, 1)                 65
=================================================================
Total params: 41,985
Trainable params: 41,985
Non-trainable params: 0
_________________________________________________________________
```

```
pred_test=test2.predict(X_pred)
Y_pred=pred_test
Y_pred=Y_pred.reshape(x.shape[0],y.shape[0])
```

```
fig = plt.figure(figsize=(10,7))
ax1 = fig.add_subplot(111)
#ax2 = fig.add_subplot(312)
#ax3 = fig.add_subplot(313)
sns.scatterplot(X[Y==1,0],X[Y==1,1],alpha=0.1,ax=ax1)
sns.scatterplot(X[Y==-1,0],X[Y==-1,1],alpha=0.1,ax=ax1)
cset =ax1.contour(xx,yy,Y_pred,cmap='twilight')
ax1.clabel(cset, inline=1, fontsize=10)
```

```
<a list of 7 text.Text objects>
```



6.4 Demo 3: HKR classifier on MNIST dataset

This notebook will demonstrate learning a binary task on the MNIST0-8 dataset.

```
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.keras.layers import ReLU
from tensorflow.keras.optimizers import Adam

from deel.lip.layers import SpectralConv2D, SpectralDense, FrobeniusDense
from deel.lip.activations import MaxMin, GroupSort, GroupSort2, FullSort
from deel.lip.utils import load_model
from deel.lip.losses import HKR_loss, KR_loss, hinge_margin_loss

from model_samples.model_samples import get_lipMLP, get_lipVGG_model
```

6.4.1 data preparation

For this task we will select two classes: 0 and 8. Labels are changed to {-1,1}, wich is compatible with the Hinge term used in the loss.

```
from tensorflow.keras.datasets import mnist

# first we select the two classes
selected_classes = [0, 8]  # must be two classes as we perform binary classification

def prepare_data(x, y, class_a=0, class_b=8):
    """
    This function convert the MNIST data to make it suitable for our binary␣
↪classification
    setup.
    """
    # select items from the two selected classes
    mask = (y==class_a)+(y==class_b)  # mask to select only items from class_a or␣
↪class_b
    x=x[mask]
    y=y[mask]
    x=x.astype('float32')
    y=y.astype('float32')
    # convert from range int[0,255] to float32[-1,1]
    x/=255
    x=x.reshape((-1,28,28,1))
    # change label to binary classification {-1,1}
    y[y==class_a] = 1.0
    y[y==class_b] = -1.0
    return x, y

# now we load the dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# prepare the data
x_train, y_train = prepare_data(x_train, y_train, selected_classes[0], selected_
↪classes[1])
x_test, y_test = prepare_data(x_test, y_test, selected_classes[0], selected_
↪classes[1])

# display infos about dataset
print("train set size: %i samples, classes proportions: %.3f percent" %
      (y_train.shape[0], 100*y_train[y_train==1].sum()/y_train.shape[0]))
print("test set size: %i samples, classes proportions: %.3f percent" %
      (y_test.shape[0], 100*y_test[y_test==1].sum()/y_test.shape[0]))
```

```
train set size: 11774 samples, classes proportions: 50.306 percent
test set size: 1954 samples, classes proportions: 50.154 percent
```

6.4.2 Build lipschitz Model

Let's first explicit the paremeters of this experiment

```
# training parameters
epochs=5
batch_size=128

# network parameters
hidden_layers_size = [128,64,32]
activation = GroupSort #ReLU, MaxMin, GroupSort2

# loss parameters
min_margin=1
alpha = 10
```

Now we can build the network. Here the experiment is done with a MLP. But `Deel-lip` also provide state of the art 1-Lipschitz convolutions.

```
K.clear_session()
# helper function to build the 1-lipschitz MLP
wass=get_lipMLP((28,28,1), hidden_layers_size = hidden_layers_size ,
↪activation=activation, nb_classes = 1,kCoefLip=1.0)
# an other helper function exist to build a VGG model
# wass=get_lipVGG_model((28,28,1),layers_conv=[32,64],layers_dense=[128],activation_
↪conv=GroupSort2,activation_dense=FullSort,use_bias=True , nb_classes = 1, last_
↪activ = None)
wass.summary()
```

```
128
64
32
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
spectral_dense (SpectralDens (None, 128)               100609
_________________________________________________________________
group_sort (GroupSort)       (None, 128)               0
_________________________________________________________________
spectral_dense_1 (SpectralDe (None, 64)                8321
_________________________________________________________________
group_sort_1 (GroupSort)     (None, 64)                0
_________________________________________________________________
spectral_dense_2 (SpectralDe (None, 32)                2113
_________________________________________________________________
group_sort_2 (GroupSort)     (None, 32)                0
_________________________________________________________________
frobenius_dense (FrobeniusDe (None, 1)                 33
=================================================================
Total params: 111,076
Trainable params: 110,849
Non-trainable params: 227
_________________________________________________________________
```

```
optimizer = Adam(lr=0.01)
```

```
# as the output of our classifier is in the real range [-1, 1], binary accuracy must␣
↪be redefined
def HKR_binary_accuracy(y_true, y_pred):
    S_true= tf.dtypes.cast(tf.greater_equal(y_true[:,0], 0),dtype=tf.float32)
    S_pred= tf.dtypes.cast(tf.greater_equal(y_pred[:,0], 0),dtype=tf.float32)
    return binary_accuracy(S_true,S_pred)
```

```
wass.compile(
    loss=HKR_loss(alpha=alpha,min_margin=min_margin),  # HKR stands for the hinge␣
↪regularized KR loss
    metrics=[
        KR_loss((-1,1)),  # shows the KR term of the loss
        hinge_margin_loss(min_margin=min_margin),  # shows the hinge term of the loss
        HKR_binary_accuracy  # shows the classification accuracy
    ],
    optimizer=optimizer
)
```

6.4.3 Learn classification on MNIST

Now the model is build, we can learn the task.

```
wass.fit(
    x=x_train, y=y_train,
    validation_data=(x_test, y_test),
    batch_size=batch_size,
    shuffle=True,
    epochs=epochs,
    verbose=1
)
```

```
Train on 11774 samples, validate on 1954 samples
Epoch 1/5
11774/11774 [==============================] - 5s 426us/sample - loss: -3.8264 - KR_
↪loss_fct: -5.2401 - hinge_margin_fct: 0.1413 - HKR_binary_accuracy: 0.9546 - val_
↪loss: -6.3826 - val_KR_loss_fct: -6.6289 - val_hinge_margin_fct: 0.0269 - val_HKR_
↪binary_accuracy: 0.9889
Epoch 2/5
11774/11774 [==============================] - 2s 194us/sample - loss: -6.5813 - KR_
↪loss_fct: -6.8297 - hinge_margin_fct: 0.0248 - HKR_binary_accuracy: 0.9906 - val_
↪loss: -6.8006 - val_KR_loss_fct: -6.9829 - val_hinge_margin_fct: 0.0202 - val_HKR_
↪binary_accuracy: 0.9908
Epoch 3/5
11774/11774 [==============================] - 2s 206us/sample - loss: -6.8227 - KR_
↪loss_fct: -7.0366 - hinge_margin_fct: 0.0214 - HKR_binary_accuracy: 0.9929 - val_
↪loss: -6.8027 - val_KR_loss_fct: -7.0636 - val_hinge_margin_fct: 0.0270 - val_HKR_
↪binary_accuracy: 0.9893
Epoch 4/5
11774/11774 [==============================] - 2s 206us/sample - loss: -6.9042 - KR_
↪loss_fct: -7.1081 - hinge_margin_fct: 0.0204 - HKR_binary_accuracy: 0.9929 - val_
↪loss: -6.9615 - val_KR_loss_fct: -7.1755 - val_hinge_margin_fct: 0.0233 - val_HKR_
↪binary_accuracy: 0.9913
Epoch 5/5
```

(continues on next page)

(continued from previous page)

```
11774/11774 [==============================] - 2s 207us/sample - loss: -6.9774 - KR_
↪loss_fct: -7.1707 - hinge_margin_fct: 0.0193 - HKR_binary_accuracy: 0.9927 - val_
↪loss: -6.9884 - val_KR_loss_fct: -7.1752 - val_hinge_margin_fct: 0.0215 - val_HKR_
↪binary_accuracy: 0.9918
```

```
<tensorflow.python.keras.callbacks.History at 0x1fd64b2a048>
```

As we can see the model reach a very decent accuracy on this task.