API tutorial for waste image classification

This tutorial is a documentation of a keras API, we will be presenting a method developed by Harold Waste staff in order to deploy a REST API for image classification.

The examples covered in this post will serve as a starting point for building your own deep learning APIs, further extension of the code is possible based on your needs and on how scalable your API endpoint needs to be.

Specifically, we will be discovering : - how to load a keras model into memory so it can be efficiently used for inference

- How to use the Flask web framework to create an endpoint for our API
- How to make predictions using our model, JSON-ify them, and return the results to the client
- How to call our Keras REST API using both terminal and postman

By the end of this tutorial you'll have a good understanding of the components that go into a creating Keras REST API.

Feel free to use the code presented in this guide as a starting point for your own deep learning REST API.

Note: The method covered here is intended to be instructional. It is not meant to be production-level and capable of scaling under heavy load. It was developed on top of intern models we made.

Configuration of the development environment

We'll suppose that keras is alread configured and installed on your machine. If not, please make sure you install keras using the official instructions.

We'll be making the assumption that Keras is already configured and installed on your machine. If not, please ensure you install Keras using the official install instructions.

From there, we'll need to install Flask (and its associated dependencies), a Python web framework, so we can build our API endpoint. We'll also need requests so we can consume our API as well.

From there, we can easily install Flask (and its associated dependecies), a web framework designed for building API endpoint. We'll also need requests so we can consume our API as well.

We'll be using the pip command for installing the needed packages.

```
$ pip install -r requirements.txt
```

I strongly recommend that you work within a virtual environment, it helps keep your computer away of incompatible versions of packages

Here is a great tutorial that makes a brief and concise introduction to virtual environnement alternative.

Essentially these instruction will be helpful :

To install (make sure virtualenv is already installed)

```
$ pip install virtualenvwrapper
$ export WORKON_HOME=~/Envs
$ source /usr/local/bin/virtualenvwrapper.sh
```

1- Create a virtual environment:

```
$ mkvirtualenv my_project
```

This creates the my_project folder inside ~/Envs.

2- Activate the virtual environement

```
$ workon my_project
```

3-Deactivating is also possible:

```
$ deactivate
```

4- delete is also allowed

```
$ rmvirtualenv venv
```

Building Harold Keras REST API

Our Keras REST API is self-contained in a single file named app.py. We kept the installation in a single file as a manner of simplicity.

Inside app.py you'll find functions necessary for our classification task.

Our three main function are namely:

- upload-model-weights: Used to load our trained Keras model (architecture & weights), prepare it for inference.
- process_data: This function preprocesses an input image prior to passing it through our network for prediction
- predict: Our API endopoint which will classify the incoming data from the request and return the results to the clients as a json file.

The full code to this tutorial can be found here.

Our API structure has access to a certain number of pre-trained deep learning models on our internal data sets.

Our first code handles importing our required packages, initializing the Flask application and displaying available deep learning models for testing.

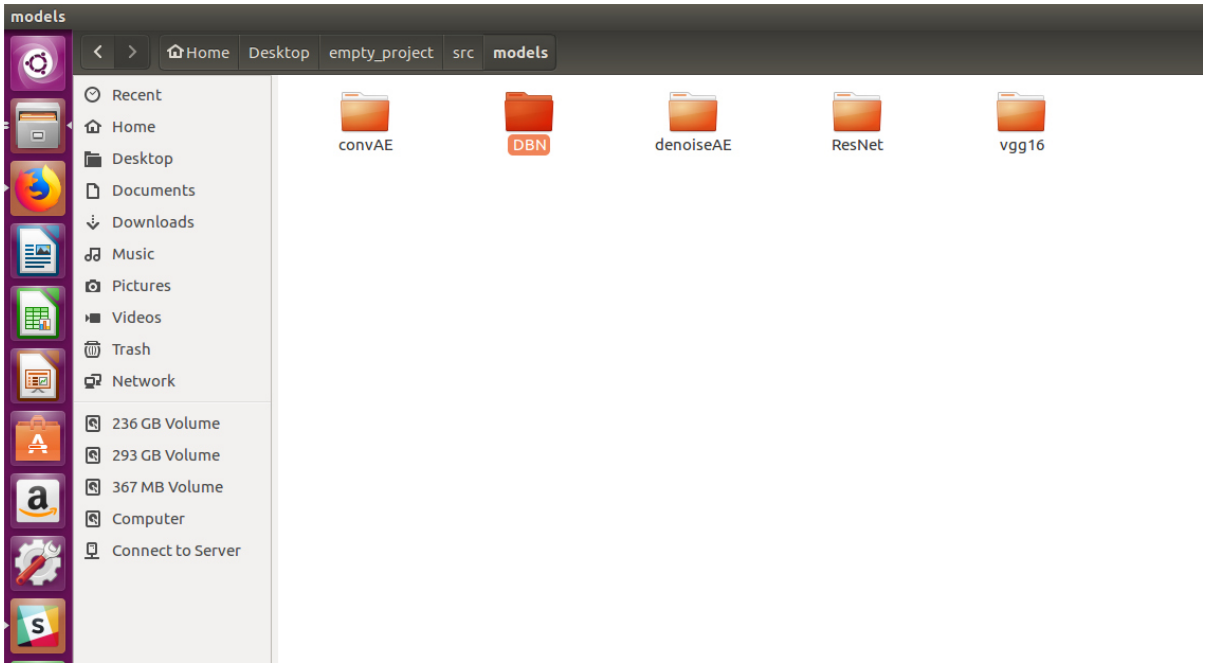
```
import os
from flask import Flask, render_template, request, jsonify
import numpy as np
from PIL import Image
import codecs, json
import tensorflow as tf
from keras.models import model_from_json
```

```
app = Flask(__name__)
```

Thus our models are structered in folders and path to models is given as input to our API for constructing its own dictionnary of models.

```
Path_to_models = '/home/Desktop/name_project/src/models/'
dictio_models_creator(Path_to_models)
```

The function dictio-models-creator creates a dictionnary containing names of all models availables along with paths to their weights and architecture,



Output:

```
{
'convAE': {'classifier_model_ConvAE.h5','classifier_model_ConvAE.json'},

'denoiseAE': {'classifier_model_DenoiseAE.h5','classifier_model_DenoiseAE.json'},

'vgg16': {'model_vgg16_transfer_learning.h5','model_vgg16_transfer_learning.json'}
 }
```

After Manually choising the model to adopt, we can to move to uploading the model architecture and weights.

From there we define the upload-model-weights function:

```
def upload_model_weights(path_to_model, path_to_weights):
    global model

    """ This function takes as input the path to the model architecture      and weights"""
```

```
# load json and create model
    my_json_file = open(path_to_model, 'r')
    loaded_model_json = my_json_file.read()
    my_json_file.close()
    model = model_from_json(loaded_model_json)
# load weights into new model
    model.load_weights(path_to_weights)

    print("Loaded  model and weights from disk")

    return model
```

As the name suggests, this method is responsible for instantiating our architecture and loading our weights from disk.

Before we can perfrom classfication over data coming from our client, we firts need to prepare and preprocess the data:

Processing data goes through 2 steps : - converting images to numpy array - scaling and normalizing data

```
def data_to_images(APP_ROOT):
    """ This function performs a request for reading data from a repository and then transf

    x_data = []
    target = os.path.join(APP_ROOT, 'images')
    print("our target = ", target)

    if not os.path.isdir(target):
        os.mkdir(target)
    print(len(request.files.getlist("my_file")))

    for file in request.files.getlist("my_file"):
        print(" file object  = ", file)
        filename = file.filename
        destination = "/".join([target, filename])
        print("path to image =", destination)

        file.save(destination)
        # taking inputs and saving 'em into tensor form
        img = Image.open(destination)

        # shape should be adjusted to model requieries  default (128,128)
        img = img.resize((128, 128), Image.ANTIALIAS)
        img = np.asarray(img)
        x_data.append(img)
        os.remove(destination)
    # my data tensor
    x_data = np.array(x_data)
    return x_data



def process_data(APP_ROOT):
    """ this function processes data and make them centered,

    it first uses the pre-defined function data_to_images to fulfill a request for reading i

    x_data = data_to_images(APP_ROOT)
    x_data = (x_data - np.min(x_data, 0)) / (np.max(x_data, 0) + 0.0001)
    return x_data
```

This function perform a request for reading an input, save the input in a local directory, then convert it into a more suitable form for our model. This function - Performs a request for reading an input - Saves the input in a local directory - Convert it into a more suitable form for our model, ideally a numpy array of shape 128x128 pixels for vgg16 for example. - Preprocesses the array via mean substraction and scaling

We are now ready to define the predict function — this method processes any requests to the /predict endpoint:

```
@app.route("/predict", methods = ['POST'])
def predict():

# processing data
    if request.method == "POST":
        x_data= process_data()
        print(" input data shape  ", x_data.shape)
    # classify the input image and then initialize the
    # list of predictions to return to the client

        class_predicted=[]
        for i in range(len(x_data)):
        # reshaping every input in tensor (1,width,height,channels)
            image = x_data[i].reshape(1, x_data[i].shape[0], x_data[i].shape[1], x_data[i].s
            with graph.as_default():
                prediction = predict_function(model, image,dictionnary)
            print('my predictions : ', prediction, '\n')
            class_predicted.append(prediction[0][0])

        data = {"success": False}
        data["predictions"] = []
    # loop over the results and add them to the list of
    # returned predictions

        for (label, prob) in class_predicted:
            r = {"label": label, "probability": float(prob)}
            data["predictions"].append(r)

    # indicate that the request was a success
        data["success"] = True

    return jsonify(data)
```

The data dictionary is used to store any data that we want to return to the client. In our case, it includes a boolean used to indicate if prediction was successful or not add to results of predictions made on the incoming data.

Before accepting the incoming data we check whether : - The request method is POST, this litterarely enables us to send arbitrary data to endpoint such as images, JSON...

We might also wanna check if an image has been passed into the files attribute during the POST

```
if flask.request.files.get("image"):
    {...
    }
```

files code and either parse the raw input data yourself or utilize request.get_json() to automatically parse the input data to a Python dictionary/object.

We then take the incoming data and: - Call out the function process_data which will read input in PIL format - Preprocess it - Pass it through our pre-trained network - Loop predictions over the input data and add them

individually to the data["predictions"] list - Return the response to the client in JSON format

For a general context, when working with non-image data, you might remove the request. If you're working with non-image data you should remove the request.files code and either parse the raw input data yourself or utilize request.get_json() to automatically parse the input data to a Python dictionary/object.

Additionally, Additionally, consider giving following tutorial a read which discusses the fundamentals of Flask's request object.

Our final step now is to launch our service:

```
if __name__ == "__main__":
    upload_model_weights(path_to_model=path_to_model, path_to_weights=path_to_weights)
    app.run(port=4555,debug=True)
```

First we call the function upload-model-weights which loades our keras model from disk and then we call our app to run.

This is a basic representaion of what logically happens but I have omit the part where we give user the possibility to choose the model he wants.

A full code is cited here:

```
if __name__ == "__main__":

    Path_to_models = /Path/to/your/models/

    Dictio_models = dictio_models_creator(Path_to_models)
    print("deep learning models available", list(Dictio_models.keys()))

    model_choice = input('choose a model for the classification task : ')
    paths = list(Dictio_models[model_choice])
    print('list of paths to weights and architecture',paths)
    # we want to ensure that path_to_weights take the data from file.h5

    if (paths[1]).endswith('h5'):

        path_to_weights, path_to_model = Path_to_models + model_choice + '/' + paths[
            1], Path_to_models + model_choice + '/' + paths[0]
    else:
        path_to_weights, path_to_model = Path_to_models + model_choice + '/' + paths[
            0], Path_to_models + model_choice + '/' + paths[1]

    upload_model_weights(path_to_model=path_to_model, path_to_weights=path_to_weights)
    app.run(port=4555,debug=True)
```

Starting your Keras Rest API

Starting the Keras REST API service is easy.

Open up a terminal, go to the directory including your python file app.py and execute:

```
$ python3 app.py
Using TensorFlow backend.
deep learning models available ['denoiseAE', 'vgg16', 'convAE']
```

```
Choose a model for the classification task : %%

...

Loaded  model and weights from disk
 * Serving Flask app "app" (lazy loading)
 * Debug mode: on
 * Running on http://127.0.0.1:4555/ (Press CTRL+C to quit)
```

Possible issue when working on terminal and PyCharm at the same time

You might notice that there is a bug saying

```
ImportError: No module named 'src'
```

You can solve this issue by logging into the app code and removing the word src from the import path. check for app.py and prediction.py

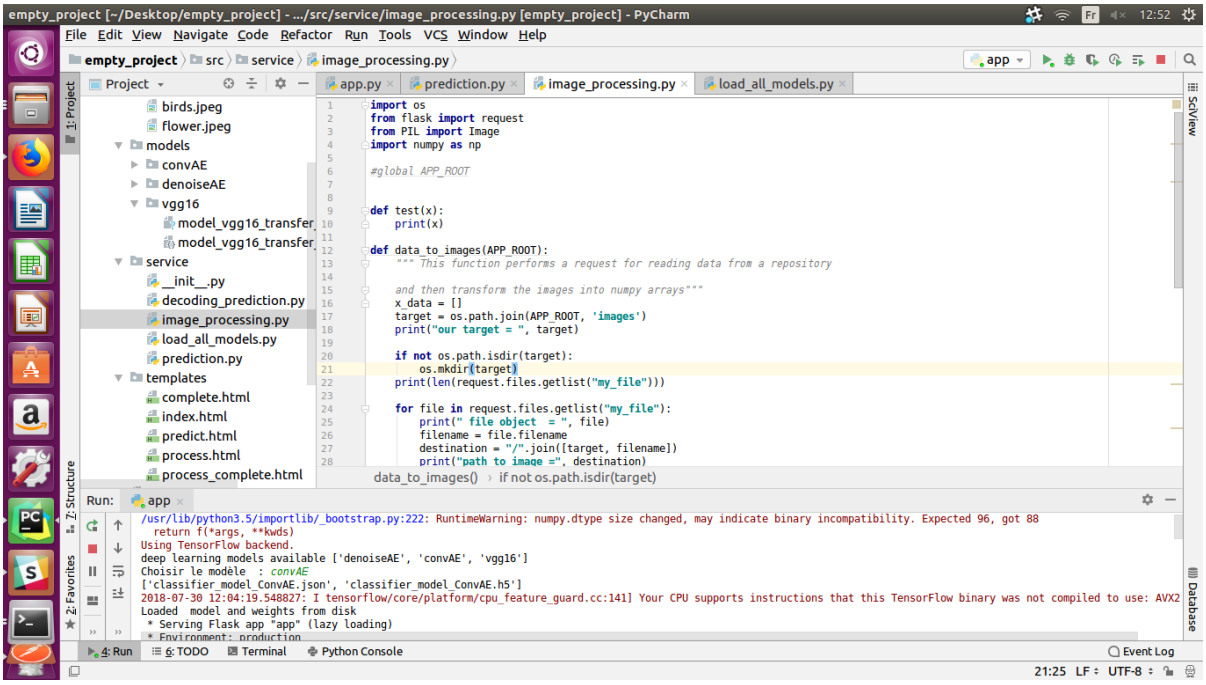
```
from src.service.load_all_models import load_all_models, upload_model_weights
from src.service.prediction import predict_function
```

to

```
from service.load_all_models import load_all_models, upload_model_weights
from service.prediction import predict_function
```

As you can see from the output, our model is loaded first — after which we can start our Flask server. You can now access the server via http://127.0.0.1:4555/

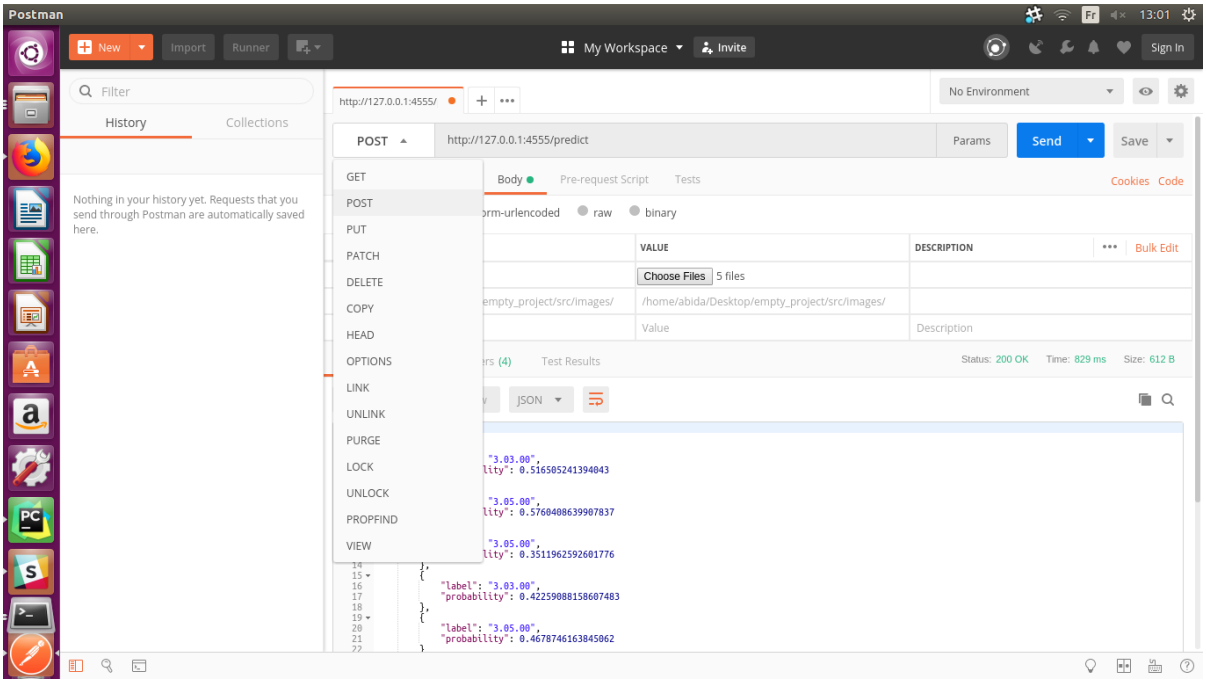
For me, I prefer working with PyCharm, he gives you direct access to your code and provides also the debugging task wich helps detect any kind of problems and performs checkpoints.



For running the API, you can simply press the button Run app.py and then our instance is activated.

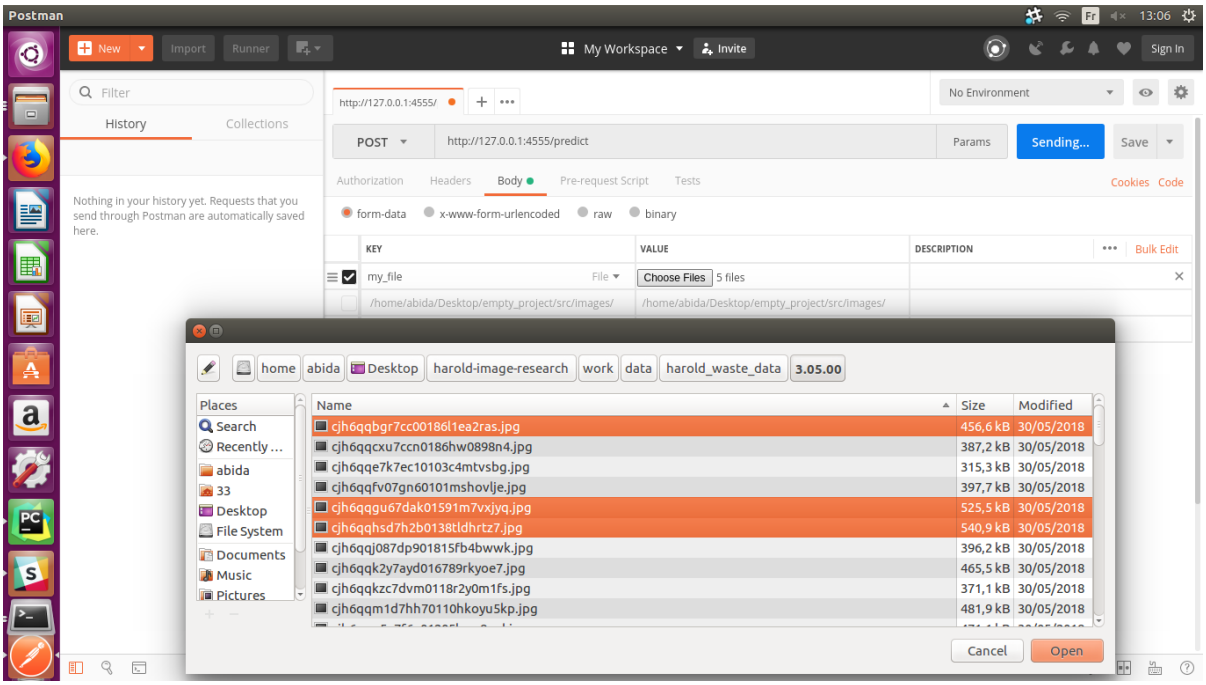
For fulfilling the task of uploading data, we omit the html way which is not very efficent and prefer the application postman which helps us interact with our API through a dynamic interface.

Postman help achieve simple tasks as POST and GET through two or three clicks



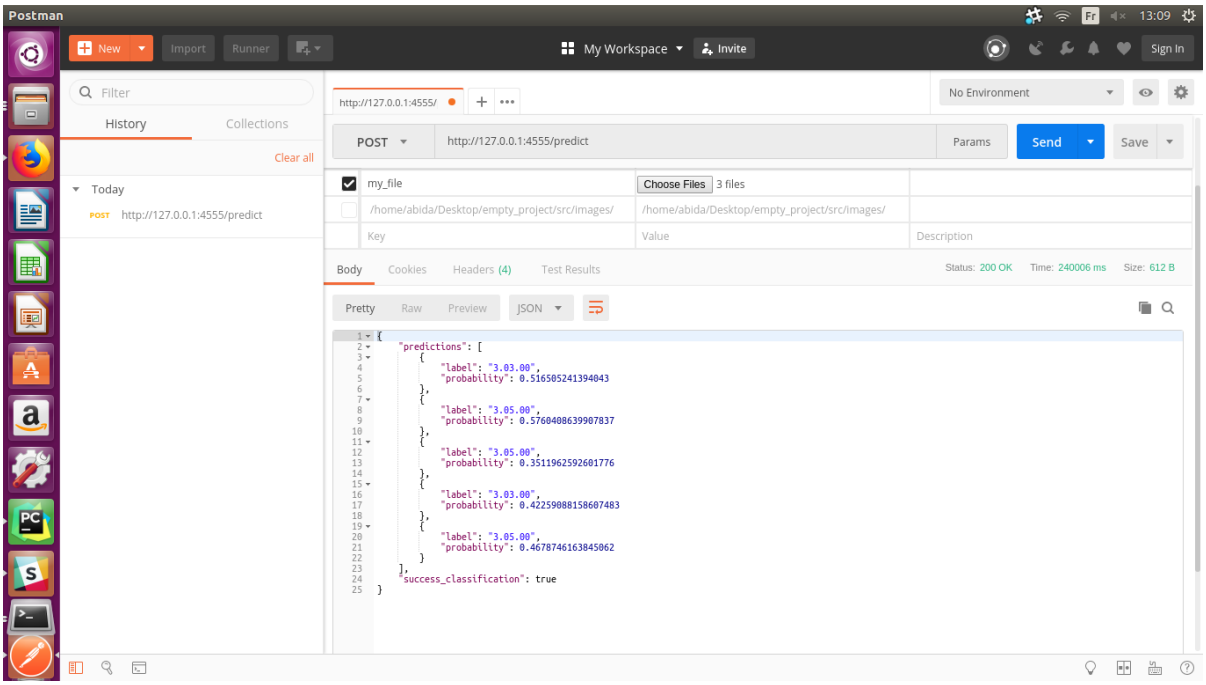
As we can see in the image above, POSTMAN give you a large set of possible action to perform within an API.

For instance, in our API we are intrested in performing a POST request to upload our data images, so we cite here the link to our local server " http://127.0.0.1:4555/predict ", and we press on the label Body, we'll have then access to our data by clicking on "Choose Files"



You simply then choose your images data and press "Send", Output will then be printed within POSTMAN

as a json file



And here we come to our main goal which is visualizing the predictions over a set of input images. Output is in format json file containing data of different types and collected into a dictionnary.

I hope this documentation helps you to understand the mechanism developed to build this classification API.

Don't hesitate to comment or to contact me in case of a problem or a suggestion to make.

You can find here link to my_mail.

The code is been saved as a github project in this url.

You can find a quick and useful tutorial for github project management.