# Week 5 assignment

Name: Cloud and API deployment
Batch code: LISUM05
Submission date: 01/22/2022
Submitted to: Data Glacier

## Connection with Cloud service Heroku

Beginning from the week 4 Github repository, next we connected this with Heroku, once registered, doing click on new > create new app.

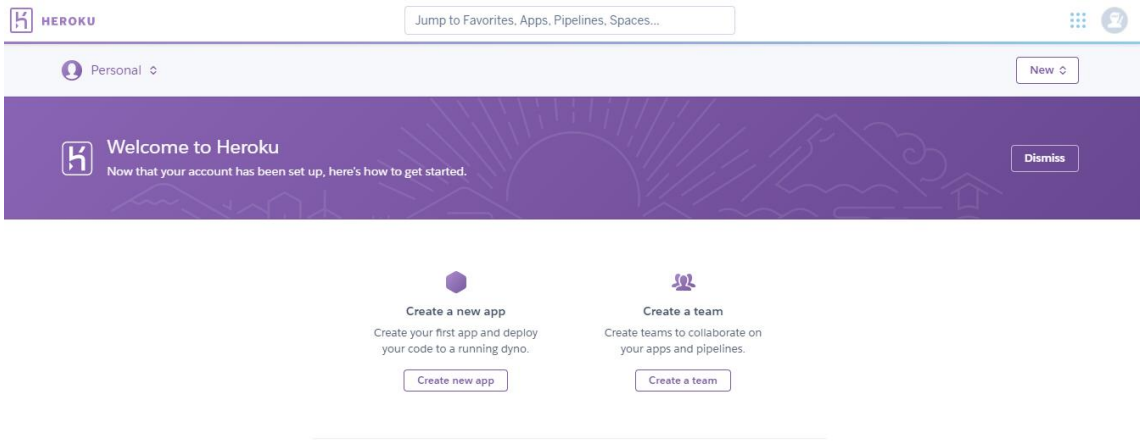



Fig. 1

After chose a name and select region, we click on deployment method and authorize connect with Github.

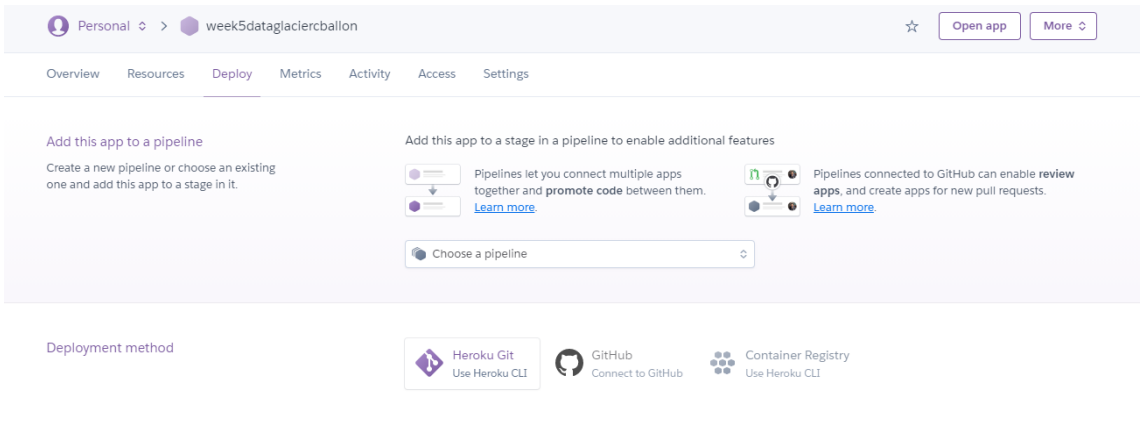



Fig. 2

After select repository we click “Deploy branch”

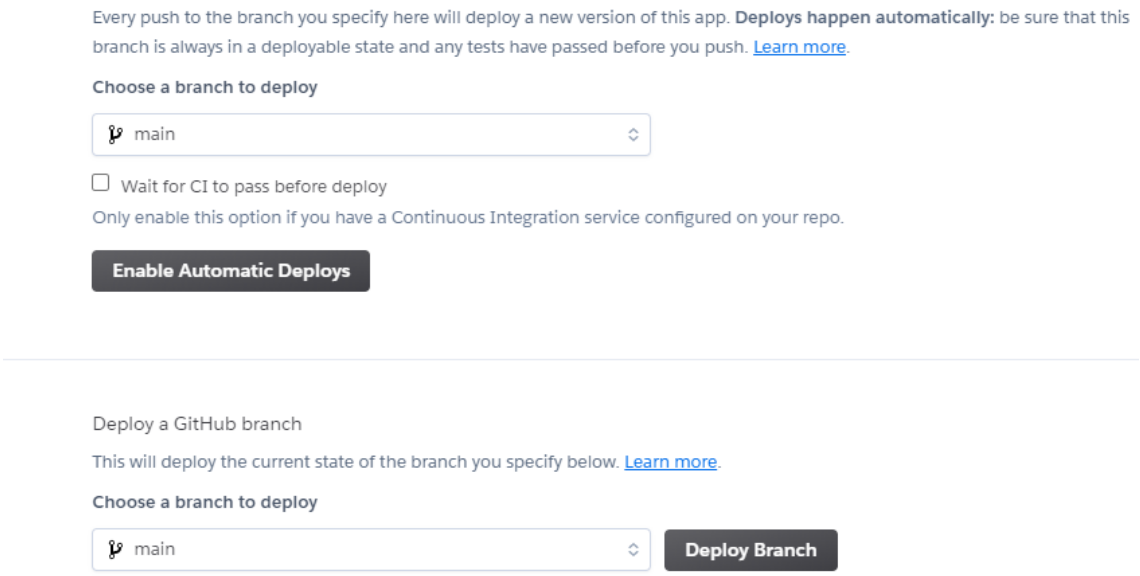



Fig. 3

After load and install all the necessary modules, the application executes. In this case our model needs to be deployed with docker to successfully execute for this purpose we create a new pipeline to Tensorflow serving.

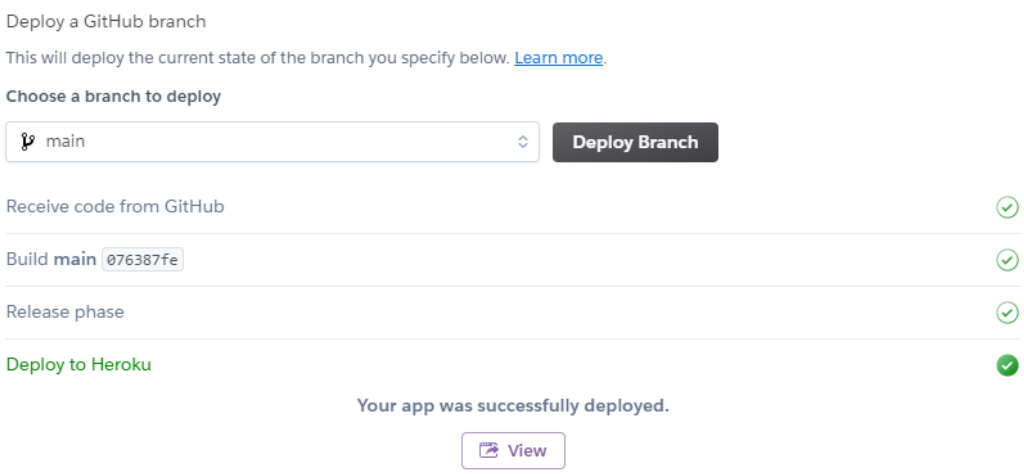



Fig. 4

## Deployment of the model

We downloaded a database with cats and dogs photos from Kaggle, the url:

[Cats -- VS -- Dogs | Kaggle]

After, we utilize a model trained for recognition of dog photos from the code at the same url: (deployed in tensorflow)

The principal code file “app.py” contains the methods ‘GET’ and ‘POST’ from Flask and link the html files “show.html” and “index.html”.

```python
from flask import Flask, render_template, url_for, request, redirect
from flask_bootstrap import Bootstrap

import os
import inference


app = Flask(__name__)
Bootstrap(app)

"""
Routes
"""
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        uploaded_file = request.files['file']
        if uploaded_file.filename != '':
            image_path = os.path.join('static', uploaded_file.filename)
            uploaded_file.save(image_path)
            class_name = inference.get_prediction(image_path)
            print('CLASS NAME=', class_name)
            result = {
                'class_name': class_name,
                'image_path': image_path,
            }
            return render_template('show.html', result=result)
    return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

To implement the web pages using html we utilized flask templates and complement styles with CSS:

```
{% extends "base.html" %}

{% block content %}
 <div>Upload File</div>
 <hr>
 <form class="form-
inline" action="{{ url_for('index') }}" method="post" enctype="multipart/
form-data">
     <div class="form-group">
        <input type="file" name="file" class="btn btn-default">
     </div>
     <div class="form-group">
        <input type="submit" value="Upload" class="btn btn-default">
     </div>
```

```
 </form>
{% endblock %}
```

Index.html

```
{% extends "base.html" %}

{% block content %}
    <div class="container">
        <h1>Predicted Class: {{ result.class_name }}</h1>
        <hr>
        <div>
            <img src="{{ result.image_path }}" class="img-
rounded" width="400" height="auto">
        </div>
        <a ref="{{ url_for('index') }}" class="btn btn-default">Back</a>
    </div>
{% endblock %}
```

Show.html

```
{% extends "bootstrap/base.html" %}

{% block title %}Cats, Dogs detector{% endblock %}

{% block content %}{% endblock %}
```

Base.html

The model calls the trained file of recognition, loads a photo and if the prediction lies upper from 0.5 the result is class Dog otherwise it's a cat. Utilizing json and request libraries we can deserialize the image files to analysis.

```
import tensorflow as tf
import numpy as np
import json
import requests


SIZE=128
MODEL_URI='http://localhost:8501/v1/models/pets:predict'
CLASSES = ['Cat', 'Dog']

def get_prediction(image_path):
    image = tf.keras.preprocessing.image.load_img(
        image_path, target_size=(SIZE, SIZE)
    )
    image = tf.keras.preprocessing.image.img_to_array(image)
    image = tf.keras.applications.mobilenet_v2.preprocess_input(image)
    image = np.expand_dims(image, axis=0)

    data = json.dumps({
        'instances': image.tolist()
```

```
    })
    response = requests.post(MODEL_URI, data=data.encode())
    result = json.loads(response.text)
    prediction = np.squeeze(result['predictions'][0])
    class_name = CLASSES[int(prediction > 0.5)]
    return class_name
```

inference.py

to run the model, we utilize Docker:

```
(base) C:\WINDOWS\system32>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

(base) C:\WINDOWS\system32>
```

Fig. 5

Running the command:

docker run -p 8501:8501 –name=pets -v “local-route” -e MODEL_NAME=pets tensorflow/serving

```
2021-07-04 01:38:01.285538: I external/org_tensorflow/tensorflow/cc/saved_model/reader.cc:132] Reading SavedModel debug
info (if present) from: /models/pets/1
2021-07-04 01:38:01.287884: I external/org_tensorflow/tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow
 binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performan
ce-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2021-07-04 01:38:01.619612: I external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:206] Restoring SavedModel bund
le.
2021-07-04 01:38:01.670490: I external/org_tensorflow/tensorflow/core/platform/profile_utils/cpu_utils.cc:114] CPU Frequ
ency: 3493435000 Hz
2021-07-04 01:38:02.818934: I external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:190] Running initialization op
 on SavedModel bundle at path: /models/pets/1
2021-07-04 01:38:03.059347: I external/org_tensorflow/tensorflow/cc/saved_model/loader.cc:277] SavedModel load for tags
{ serve }; Status: success: OK. Took 1930315 microseconds.
2021-07-04 01:38:03.118415: I tensorflow_serving/servables/tensorflow/saved_model_warmup_util.cc:59] No warmup data file
 found at /models/pets/1/assets.extra/tf_serving_warmup_requests
2021-07-04 01:38:03.270078: I tensorflow_serving/core/loader_harness.cc:87] Successfully loaded servable version {name:
pets version: 1}
2021-07-04 01:38:03.273580: I tensorflow_serving/model_servers/server_core.cc:486] Finished adding/updating models
2021-07-04 01:38:03.273845: I tensorflow_serving/model_servers/server.cc:367] Profiler service is enabled
2021-07-04 01:38:03.278553: I tensorflow_serving/model_servers/server.cc:393] Running gRPC ModelServer at 0.0.0.0:8500 .
..
[warn] getaddrinfo: address family for nodename not supported
2021-07-04 01:38:03.289973: I tensorflow_serving/model_servers/server.cc:414] Exporting HTTP/REST API at:localhost:8501
...
[evhttp_server.cc : 245] NET_LOG: Entering the event loop ...
```

Fig. 6

We verified the correct creation of the environment:

```
(base) C:\WINDOWS\system32>docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED         STATUS         PORTS
                    NAMES
2fb387e42f4f   tensorflow/serving   "/usr/bin/tf_serving…"   14 minutes ago   Up 14 minutes   8500/tcp, 0.0.0.0:8501->85
01/tcp, :::8501->8501/tcp   pets
```

Fig. 7

After, we can run the python file and it begins to run the web server in localhost:

```
(base) C:\Users\crbal\OneDrive\Data Glacier Internship\Week 4\app>python app.py
2021-07-04 08:17:52.104709: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic libra
ry 'cudart64_110.dll'; dlerror: cudart64_110.dll not found
2021-07-04 08:17:52.107555: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do n
ot have a GPU set up on your machine.
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
2021-07-04 08:18:15.382488: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic libra
ry 'cudart64_110.dll'; dlerror: cudart64_110.dll not found
2021-07-04 08:18:15.384568: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do n
ot have a GPU set up on your machine.
 * Debugger is active!
 * Debugger PIN: 336-907-541
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Fig. 8

Introducing this address to web browser we visualize the app running.

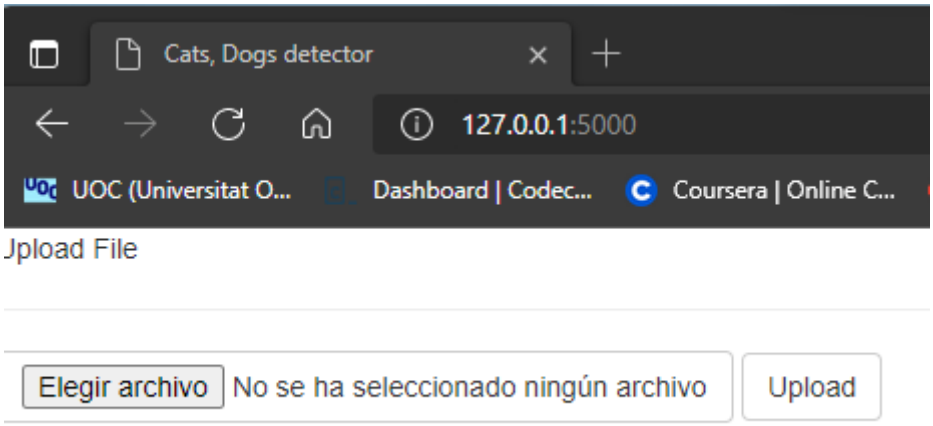
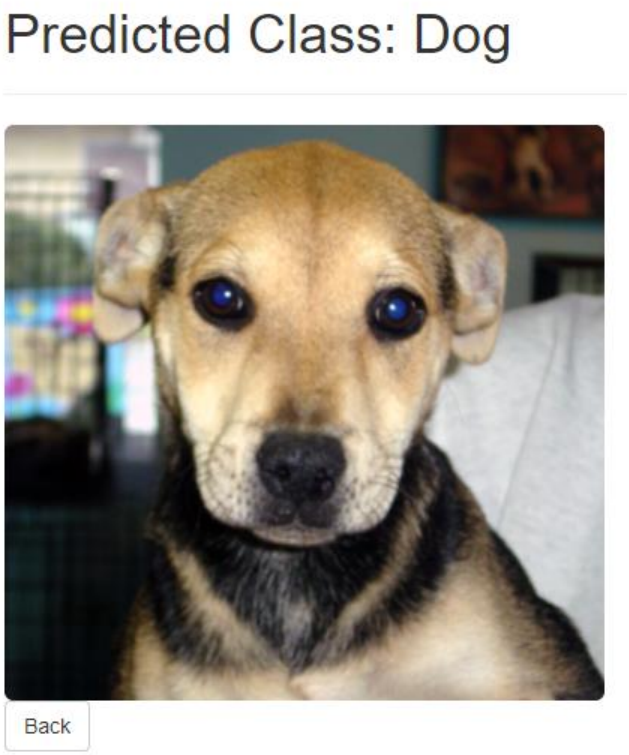



Fig. 9

We select an image from the database and click upload.




(a)


(b)

Fig. 10

After, the result for inference is displayed.