Deployment and end-user Documentation for the image processing service

The written service is a rest webservice which runs a deep network on top of the images and returns the probable categories in the input images.

Technical Specifications:

The service has been written in python, and has been tried to be losely coupled to the operating system, and dependencies. However the commands for installation of dependencies given here are limited to the linux os, however it is easy to find their equivalents in other operating systems.

Probably the only two requirements that might not be available on the testing server would be the webpy, which is used to make the rest webservice, and caffe deep learning library.

If a linux is used for testing, this is how webpy can be installed:
easy_install web.py
Webpy is a lightweight web framework for python that is simple to implement and use.

The Caffe deep learning library has been used as the backend. Please make sure your the pycaffe has been installed in addition to the caffe itself:
make pycaffe

Although the service has been written to be flexible and reusable for different networks, I have used the residual network (resnet101) for it. This network has been learned on the imagenet dataset with 1000 different categories. However the configuration file can be easily changed, and it presumable should be able to work with any other network.

I have prepared the directory and everything so that it can be tested without any specific requirement on any server, by only running the service by the command:
python image_processing_service.py

By default, webpy will load a lightweight web service with its root being the current directory, on port 8080. however, the current service can be easily copied to apache or a lighttpd as explained in the documentation with the list below (please note that you don't need to do these steps if you want to test it as it is):
http://webpy.org/install

Service specifications

the written rest web service has been implemented to reply to GET and POST requests. There are three parameters for the service:

- json this is the input json text. Note that the service takes this input exactly in the form of suggested input json. A sample input json is shown below:

```
{
  "images": [
    "http://localhost:8080/static/imgs/now.jpg",
    "http://localhost:8080/static/imgs/lion.jpg",
    "http://localhost:8080/static/imgs/shahin_lama.jpg",
    "http://localhost:8080/static/imgs/setia_my_niece.jpg",
    "http://localhost:8080/static/imgs/my_pet.jpg",
    "http://localhost:8080/static/imgs/my_nieces_cat.jpg",
    "http://localhost:8080/static/imgs/icip_img.jpg",
    "http://localhost:8080/static/imgs/persian_breakfast.jpg",
    "http://localhost:8080/static/imgs/mexico.jpg",
    "http://localhost:8080/static/imgs/basketball.jpg"

  ]
}
```


Illustration 1: Input json sample

- confMinInt this value shows the minimum confidence interval for the class to be shown in output (as suggested by the test). If the value is not provided, no constraint on the value of confidences for class is imposed
- confMaxNoCls this value imposes on the number of the categories that are output for each image. For instance if this value is given as 2, only the first two categories with highest interval are returned for each image.

Please note that if both confMinInt and confMaxNoCls are provided, both these values are satisfied (**and** condition), for example if confMinInt is 0.2 and confMaxNoCls is 3, only the 3 most confidence categories are first found, and only the ones that have a confidence value higher than 0.3 shown. Consequently it might return 0 to 3 categories for each input image by these two filterings.

Example outputs of the service

We note that you can easily copy any text into the url, and it will convert to the correct sending format by default by the browser. In the following some example request/responses are shown

sample1:

request:
http://localhost:8080/?confMaxNoCls=3&json={%22images%22:%20[%20%22http://localhost:8080/static/imgs/my_pet.jpg%22,%20%22http://localhost:8080/static/imgs/my_nieces_cat.jpg%22]}

images:

Illustration 2: my_pet.jpg image

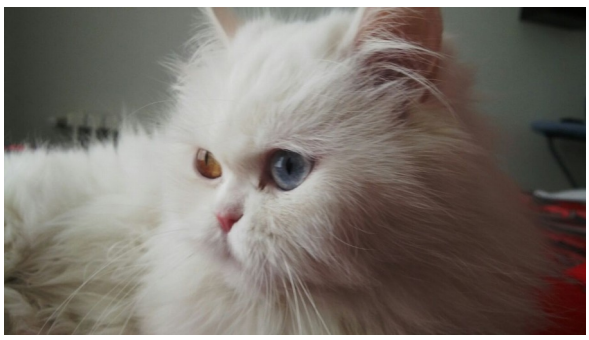


Illustration 3: my_nieces_cat.jpg

response:

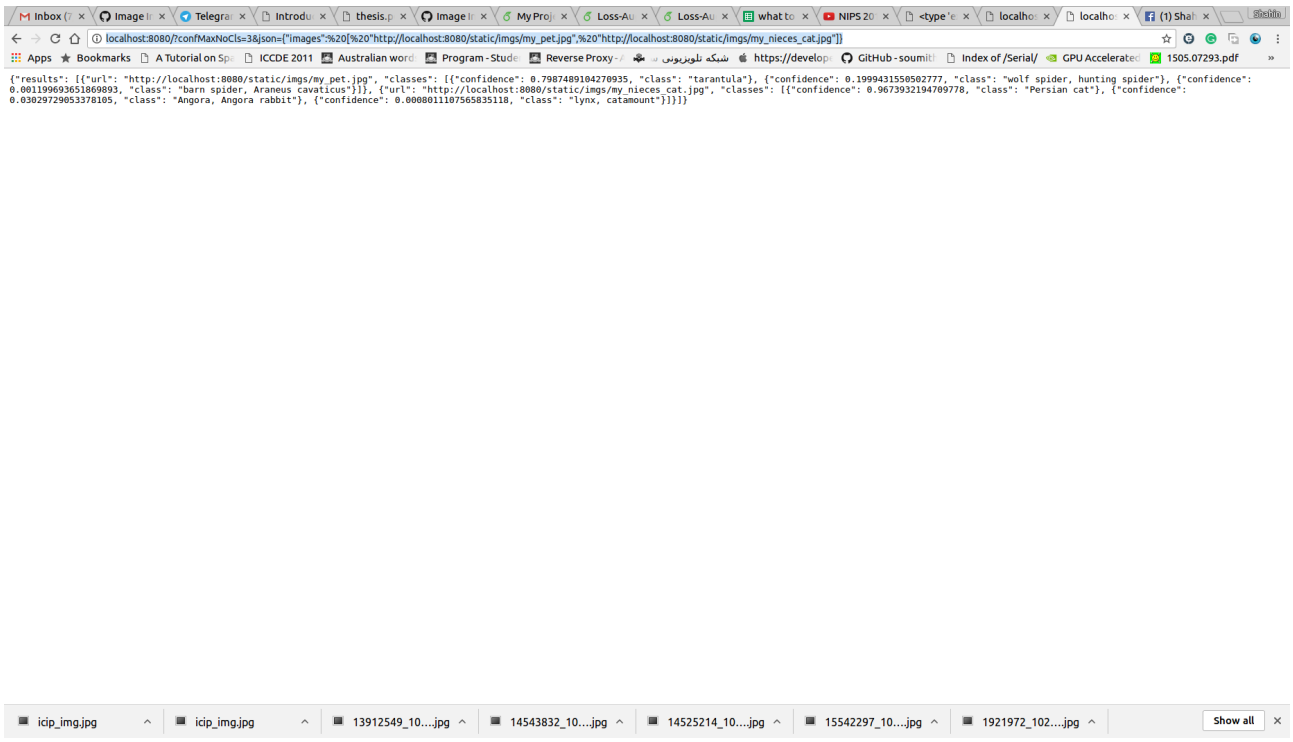


Illustration 4: output for the spider and cat image

as can be seen, the output shows only 3 categories for each image. The format of the json output is exactly as it has been suggested in the test. For the cat image, it has been able to discriminate it with the highest probability as persian cat, and for the spider image, it has been correctly classified as:

```
[{"confidence": 0.7987489104270935, "class": "tarantula"}, {"confidence":
0.1999431550502777, "class": "wolf spider, hunting spider"}, {"confidence":
0.001199693651869893, "class": "barn spider, Araneus cavaticus"}]
```

sample2:
request:
with default parameters, 10 images are considered as input, as suggested in the ./static/json/jsonfile.json file (more on this on the next session). So the default request is:
http://localhost:8080/
and the thumbnail of the 10 images are shown below:

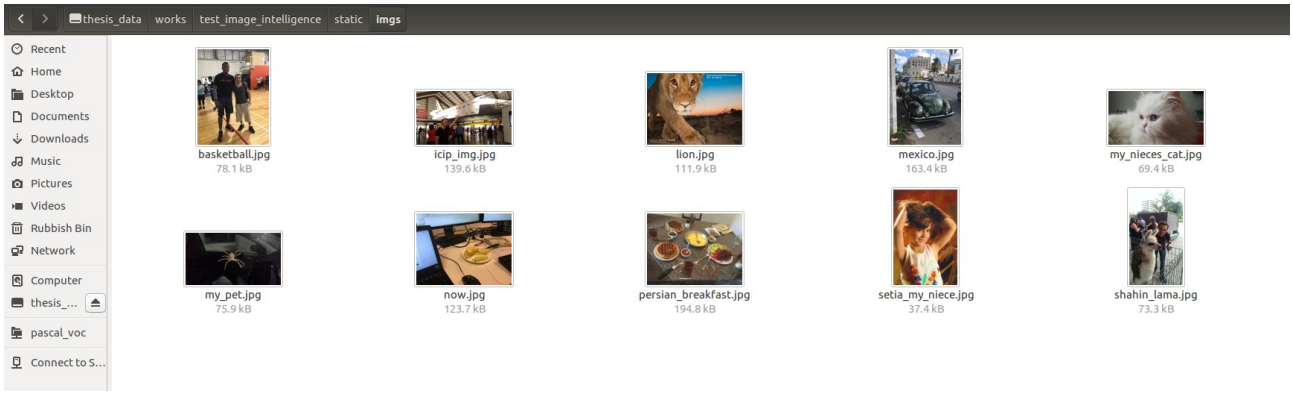


Illustration 5: Thumbnail of the 10 images considered as default

Response:

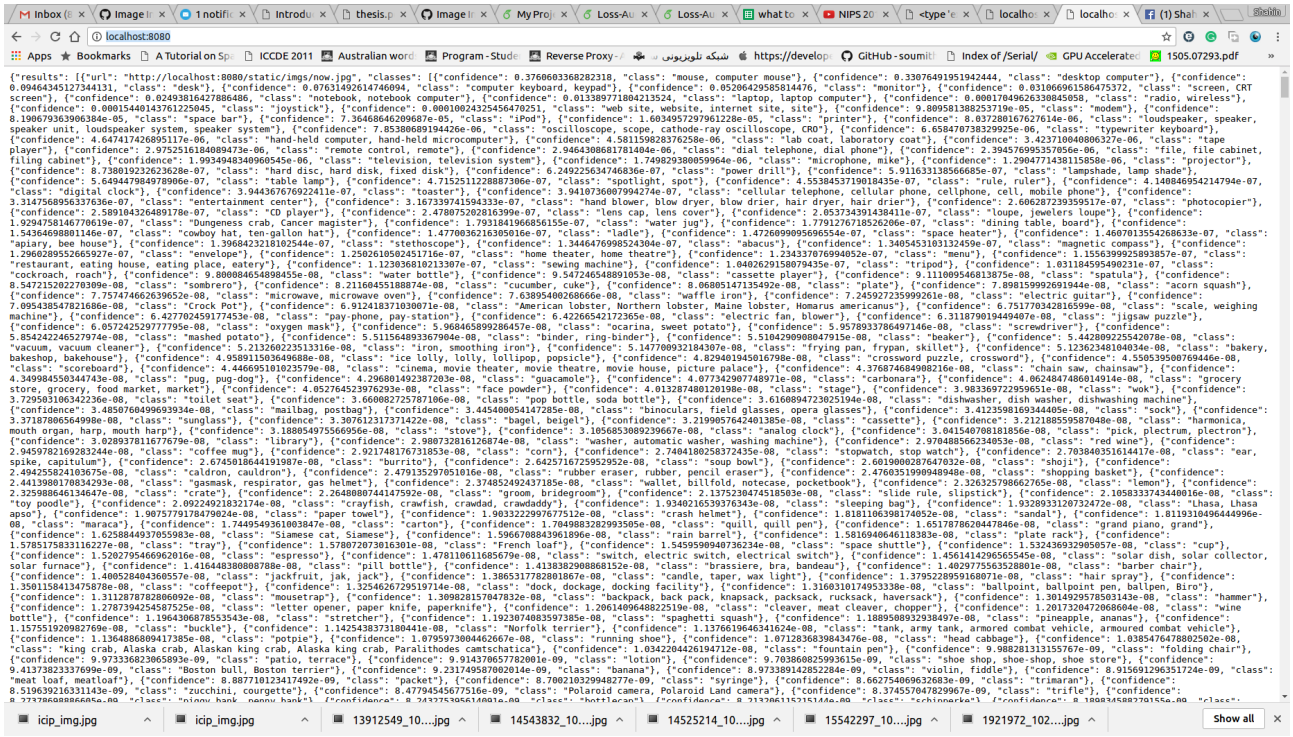


Illustration 6: The response for the 10 images which is given as input by default. Note that for each image all 1000 categories are shown with their intervals in descending order

Sample3

In this sample we pass a non-complete json to the service and we expect to get a caught error. Here is the passed json:

```
{
  "images": [
    "http://localhost:8080/static/imgs/now.jpg"
    "http://localhost:8080/static/imgs/lion.jpg"]}
```

as can be seen, we have missed the comma between the two urls.

The corresponding request is:

http://localhost:8080/?
json={%20%22images22:%20[%20%22http://localhost:8080/static/imgs/now.jpg%22%20%22http://localhost:8080/static/imgs/lion.jpg%22]}

As suggested by the error in the response:

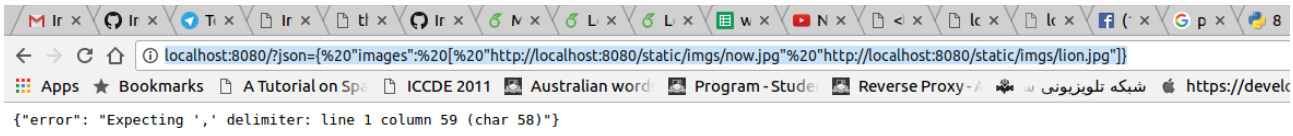
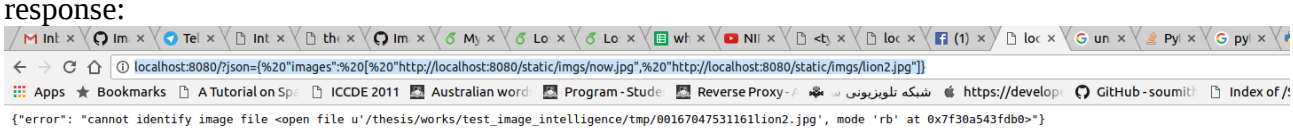


Illustration 7: Response with error. The json url suggests what the problem is in the sent parameters

Sample4
In this example an image url is given which doesn't exist
request:
http://localhost:8080/?json={%20%22images%22:%20[%20%22http://localhost:8080/static/imgs/now.jpg%22,%20%22http://localhost:8080/static/imgs/lion2.jpg%22]}
note that the lion2.jpg image does not exist. We expect to get an informative response accordingly.

response:



As can be seen, the response error suggests that the lion2 image doesn't exist.

Explanation of files in the directory

The structure of the current directory is as show above:

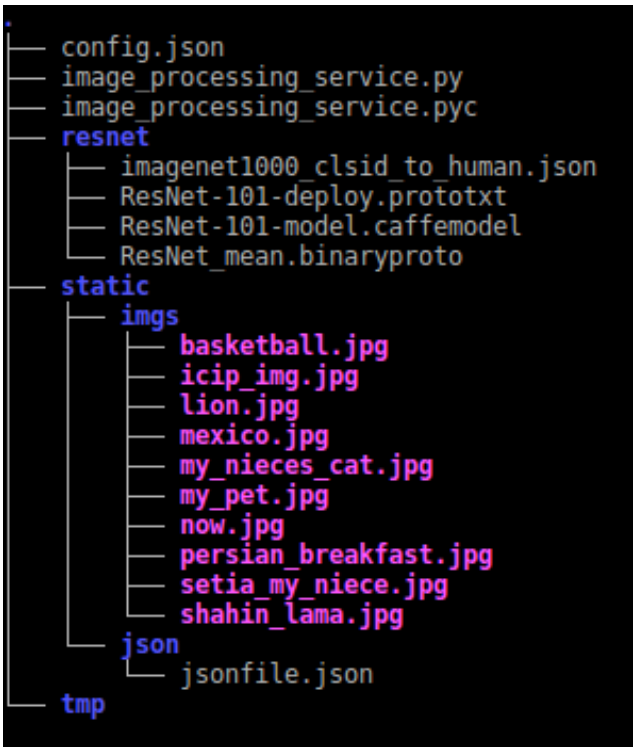


Illustration 8: Illustration of the root directory of the service

config.json:

this file keeps the configuration for the web service in a json format. I have tried to write the description of the parameters in the file, this is the configuration file in my own laptop:

```
{
  "configuration" : {
    "caffe_root" : "/thesis/git/modified_nns/PSPNet-master" ,
    "model_prototxt" : "/thesis/works/test_image_intelligence/resnet/ResNet-101-deploy.prototxt" ,
    "model_weights" : "/thesis/works/test_image_intelligence/resnet/ResNet-101-model.caffemodel" ,
    "output_layer_name" : "prob",
    "model_mean" : "/thesis/works/test_image_intelligence/resnet/ResNet_mean.binaryproto",
    "tmp_dir" : "/thesis/works/test_image_intelligence/tmp/",
    "use_gpu" : true ,
    "batch_max" : 8 ,
    "img_width" : 224,
    "img_height" : 224 ,
    "class_names": "/thesis/works/test_image_intelligence/resnet/imagenet1000_clsid_to_human.json"
  },
  "config_params_desc": {
    "caffe_root": "The address of caffe",
    "model_prototxt": "network architecture definition, Basically Resnet101",
    "model_weights" : "network weights",
    "output_layer_name" : "the name of the network output",
    "model_mean" : "mean values for input (binaryproto file)",
    "tmp_dir" : "the directory that the images are copied into temporarily and are deleted after the respond is successfully sent" ,
    "use_gpu" : "if true, use GPU for caffe, otherwise use CPU" ,
    "batch_max" : "the maximum number of images that can be computed simultaneously. Note that this is the limitation of the GPU memory,if the number of input
    "img_width" : "the network's input image width",
    "img_height" : "the network's input image height",
    "class_names" : "json file with class names in format {'classes': ['class0' , 1:'class1' , ...]}"
  }
```

Illustration 9: the config json file

In this file

- caffe_root should point to the root directory of the caffe

- model_prototxt shows the prototxt file, which is basically the network architecture of the network used.
- model_weights shows the learned weights for the network
- output_layer_name this is the layer that we want to see its output, by default it should have the name of the last softmax layer which is the probability of the categories
- model_mean is the binaryproto file that keeps the mean values of the input images for the network. These values are subtracted from the input image's pixels before being given to the network as input
- tmp_dir the service uses this directory to first download the images from the given urls, and then runs the network on the images. It can point to the tmp directory in the web service directory
- use_gpu If true, the network runs on GPU which is quite faster, but if the nvidia GPUs are not available on the server, it can be turned to false to use CPUs instead
- batch_max this number shows the number of images that their predictions are computed concurrently (test batch size in caffe). The higher the better, but there is an upper bound to this value, enforced by the memory of the GPU. For instance, if this value is 8 and we have 10 images to be classified, one feed forward of the network is carried out with the first 8 photos, and then the remaining 2 photos are computed in the next round. Note that the end user has no sense of this value, but he might get a slower response to his request if this value is small.
- img_width and img_height are the size of the network. Note that if it doesn't accurately fit the size of the image, it is scaled to this size and it then passed to the network (a common practice for deep networks), for resnet the size of the input images are 224*224
- class_names this json file is the mapping from the class number to the class name. Please have a look at the file ./resnet/imagenet1000_clsid_to_human.json to see the structure used

./static directory

This directory can keep the files that the web service can have access to. Note that there is no need to put your images/json etc. here, the image urls can be anything which is accessible to the web service.

./static/imgs directory

I have copied a set of images here and have tried to run the service on top of it. Note that there is no constraint for the images to be here, anywhere which the webservice has access to its url can be used.

./static/json/jsonfile.json

This is a sample json file which keeps the urls for the images in the ./static/imgs/. If no input is given to the service, it will read this file and classifies the images that their urls are available in this file. I have used the input and output json formats as suggested in the test. This file can be seen as one json input sample.

./resnet directory

The files corresponding to the caffe resnet-101 are copied here.