Data Intensive Computing SS2023
Project 3: Computation offloading for object detection service - Group 43

GÄCKLE CHRISTIAN, 12010963, e12010963@student.tuwien.ac.at
ROETHLIN BETTINA E., 11719747, e11719747@student.tuwien.ac.at
VIEHAUSER MAXIMILIAN, 11945353, e11945353@student.tuwien.ac.at

1 INTRODUCTION

This assignment is aimed to get accustomed to working with Docker, TensorFlow as well as the infrastructure of Amazon Web Services (AWS) and is split into three parts:

- Object detection: Implementation of an detection algorithm via TensorFlow that takes images as `.jpg` files and returns the information on bounding boxes of a detected object as a JSON response.
- Dockerization of application: making the previous application fit into a Docker container
- Local and remote execution: run the application locally as well as remote via AWS

The images used for the object detection are provided in three datasets, namely 'object-detection-SMALL', 'object-detection-MEDIUM' and 'object-detection-BIG', with the largest containing a very diverse selection of a little under 5000 pictures. The pictures show a wide variety of people, objects and also landscapes, e.g.:

- athletes performing in some sport such as skating, tennis, skiing, surfing, baseball
- people engaging in social interactions such as eating, cooking, playing on the beach
- vehicles like planes, trucks, busses, bikes, motor bikes and cars
- animals like cats, ice bears, giraffes, dogs and guinea pigs
- food as e.g. BBQ, pizza, sandwiches and cakes
- furniture like toilets or kitchens

It is also interesting to note that the images also include graphically edited pictures such as black and white pictures with some selected elements in color, pictures with some digitally added writing or watermarks, pictures of other (older printed) pictures and collages of a collection of pictures. A few examples of that you will find below:

2 PROBLEM OVERVIEW

An already mentioned problem certainly is the variety of images included in the data set. Although we use a pre-trained model from TensorFlow Hub, the variety of the data set and the computational complexity along with it still will take up a lot of computational power which is probably why we will turn to AWS to use their resources.

Authors' addresses: Gäckle Christian, 12010963, e12010963@student.tuwien.ac.at; Roethlin Bettina E., 11719747, e11719747@student.tuwien.ac.at; Viehauser Maximilian, 11945353, e11945353@student.tuwien.ac.at.

Another challenge will be the dockerization of the application. We are aware that Docker is a widely used tool in DevOps, but as this lecture is part of our Data Science program, actual software development and deployment was not yet part of our curriculum and thus our knowledge of Docker is rather basic.

We are also new to the AWS infrastructure, which also has not been part in any course of our curriculum as of now. Other than Docker, this is actually a platform that is a lot more involved with Data Science and widely used in popular domains such as sports (German Bundesliga or Formula 1), but until now there was not yet the need to scale up our computational processes to that capacity.

3 METHODOLOGY AND APPROACH

3.1 Image uploader

To detect objects in our images, we obviously have to deploy an object detection algorithm (see `app.py` method further down in subsection 3.2), but also actually feed this algorithm with images which is easier said than done. We decided to go with an API which sends the provided images to our `app.py` method.

We start by setting up a list of all the images in our image directory and also set three counters to measure the total time passed detecting objects in the images, the time needed for the upload (time between setting up the API and receiving the result) as well as counting the images themselves to later calculate the average inference time as requested in the assignment.

We start looping over all images, open each one, start the clock for the upload time measurement, encode the binary file to a base64 image and then "send" it by defining the URL of the API and then making a "POST" request to the API which includes that image. The image is now sent to the application for the object detection (see subsection 3.2 further down).

After the detection we print out the detected objects given as bounding boxes for each picture, extract our inference time for the processed image and add it to the overall inference time as well as add one to our total number of processed images. We also stop the clock for the upload time and add it to the overall upload time. When we are done with all the images, we print out the average inference and upload time.

3.2 Application for object detection

For the start we "open" the above mentioned API post to our method, of which we can then extract our image for object detection. We subsequently can open this image and turn it into an numpy array such that we can normalise the pixel data from a range of $[0, 255] \in \mathbb{N}$ to a normalised range of $[0, 1] \in \mathbb{N}$. Having preprocessed our data in a proper manner, we run the `object_detection` method as follows:

As mentioned before, the object detection application (`app.py` in our code) will be implemented using mostly TensorFlow and involve its pre-trained model, which we load at the start of our script. After turning our numpy image into a tensor, we start the clock to measure our inference time as requested in the assignment. We then perform the inference using our previously loaded detector model and save the result.

Before turning our result into a dictionary with numpy arrays, we hold the clock and save the time used for inference. The results together with the inference time is then returned. In case no object has been detected, only an empty `None` object is returned together with the inference time.

We then go on and save the returned results (if there are any) into a list, repack this again into a dictionary together with the inference time and then "jsonify" it.

3.3 Dockerization

We dockerize the application by using the provided files (`dockerfile`, `requirements` and `app.py`) to create a local docker image. In order to do so we open a terminal window (subsequently called app-terminal), move into the directory in which the dockerfile is saved and run the following command is to create a docker image:

```
docker build -t object-detection-app .
```

In order to run the application dockerized, we need to set up the container i.e. the isolated environment in which we want our application to run. To do so we use the following command:

`docker run -p 5001:5001 object-detection-app` (Note: ports might need to be adjusted)

With these two steps we have dockerized our application and can continue to locally execute the object detection app.

3.4 Local Execution

With the container running, we can execute the application on our local machine in an isolated environment by making use of two terminals running simultaneously: the first terminal serves as a client interface and the second one being the already open app-terminal specified above which actually performs the object detection in the container. We "upload" our images to the application via the first terminal as if we were uploading images to a website by running `python3 uploader.py`. Then the second app-terminal takes over, performs the object detection within the docker container and then returns our JSON file. The actual JSON file output is sent to the first terminal i.e. the client interface terminal.

3.5 Remote Execution

Regarding the remote execution on AWS, we visualised the steps to be taken in Fig. 2. As in the local execution, we first create a docker image of our application with the help of the provided `dockerfile`. We can then upload this image to dockerHUB, a platform where developers can privately or publicly share their created images. In our case, this was done with the following commands:

- renaming the docker image to the repository name, in our case bettinaroe/dic_ex3 with
  `docker tag object-detection-app bettinaroe/dic_ex3`
- pushing the docker image to the repository with name bettinaroe/dic_ex3 with
  `docker push bettinaroe/dic_ex3`.

Next up, we create our EC2 (Amazon Elastic Compute Cloud) instance on the AWS platform. An EC2 "allows users to rent virtual computers on which to run their own computer applications [and] encourages scalable deployment of applications by providing a web service through which a user can [...] configure a virtual machine" (https://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud). Thanks to a virtual 100$ provided by AWS we can configure our virtual machine along our needs and budget. With the instance created, we can connect to it from our local machine via SSH:

```
ssh -i /.ssh/id_AWS ec2-user@ip_address
```

where `id_AWS` specifies the key pair used when setting up the AWS instance and the `ip_address` being specific to that AWS instance.

We then install Docker on the instance with:

`sudo yum install -y docker`,

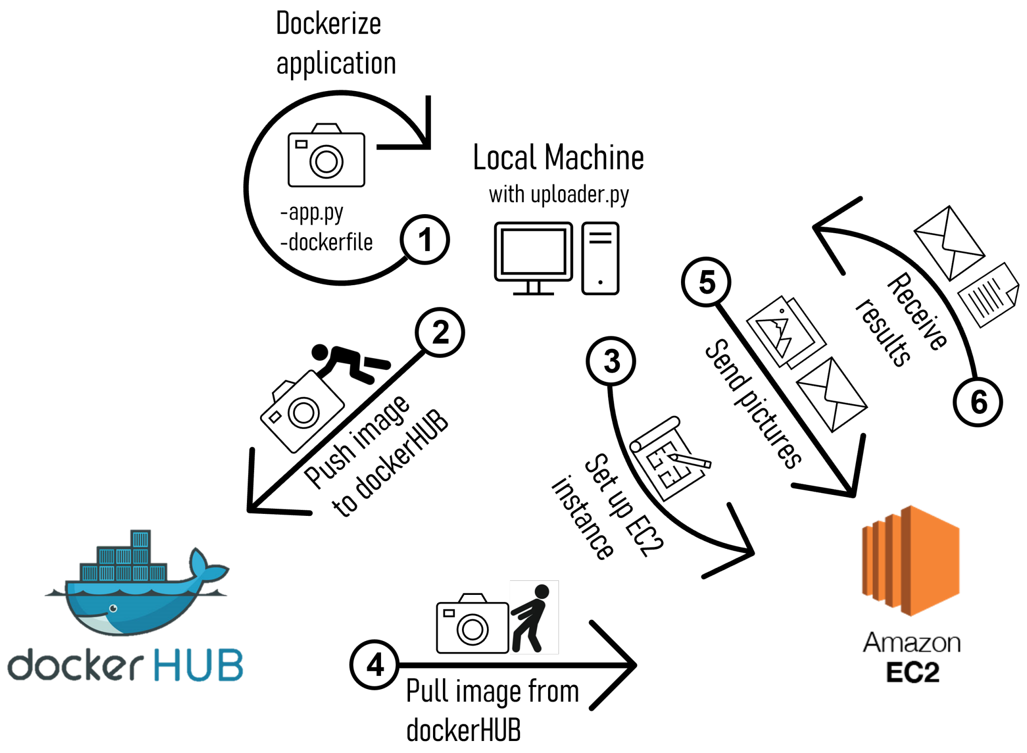


Fig. 2. Offloading Chart to AWS

start the service:

```
sudo service docker start,
```

and add the ec2-docker user to the group:

```
sudo usermod -a -G docker ec2-user.
```

After logging out and reconnecting, we can proceed to pulling the docker image of our application from dockerHUB, where we had pushed it onto earlier and setting it up on our instance:

```
docker pull bettinaroe/dic_ex3:latest
docker run -p 80:5001 -name dic_ex3 bettinaroe/dic_ex3:latest
```

and run the container:

```
docker run -p 80:5001 -name dic_ex3 bettinaroe/dic_ex3:latest
```

Now we proceed as with the local machine: we ensure that in the `uploader.py` the addresses for the API are set correctly and start sending our images to our EC2 instance which then in turn provides us with the desired results.

4 RESULTS

4.1 Local execution

Due to the limited capacity on our local machine we only ran the 'object-detection-SMALL' dataset:

- Average inference time: 0.20708430296665914 seconds
- Average upload time: 0.2765082902199513 seconds

4.2 Remote execution

The AWS EC2 instance that we used was of type `t2.large` which has below specifications:

- vCPU*: 2
- CPU Credits / hour: 36
- Mem (GiB): 8
- Storage: EBS-Only
- Network Performance: Low to Moderate to 3.0 GHz Intel Scalable Processor

The instance described above was the most powerful instance we could choose due to not being able to use more than 8GB of Memory with our account. Neither could we choose a GPU instance which means the remote machine has about the same capabilities as our local machine.

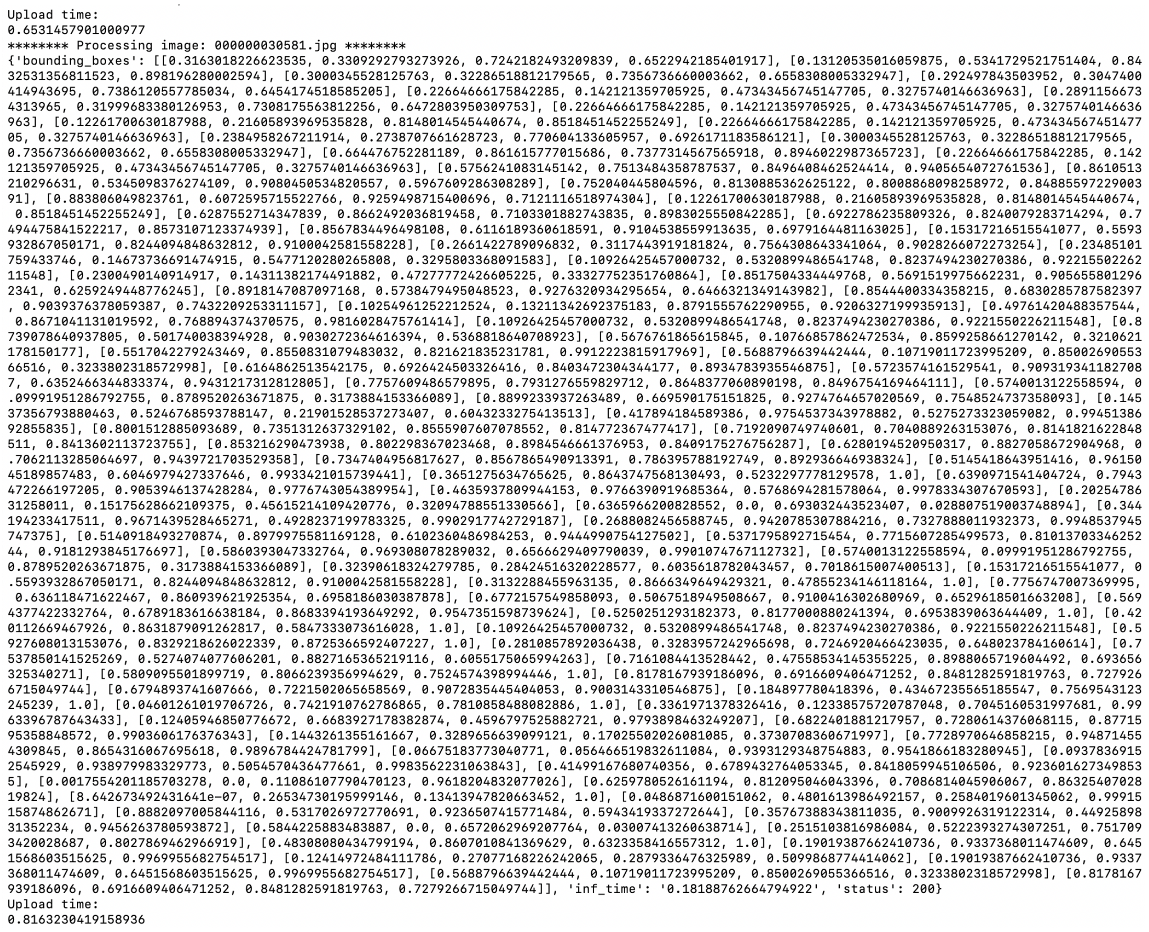
Average Inference Time on AWS EC2 instance:

- 'object-detection-SMALL' dataset:
  Average inference time: 0.18475955885809822
  Average upload time: 0.8059944035233678
- 'object-detection-MEDIUM' dataset:
  Average inference time: 0.18063510950060857
  Average upload time: 0.8011526892150658
- 'object-detection-BIG' dataset:
  Average inference time: 0.18473846533985617
  Average upload time: 0.8023746578228495

As described above, the time required to transfer each image file in the dataset to AWS, this information is output together with the bounding box information for each image. Please see Fig. 3 for a sample output.

5 CONCLUSIONS

When looking at the results, we can see that although the specifications of our EC2 instance are roughly the same as our local machine, the inference time on the instance is only 87 - 89% of the inference time needed on the local machine. On the other hand, the upload time to the EC2 instance is almost 3 times as high as on our local machine, which is not that big of a surprise as we send the pictures to a server connected to our local machine only via internet. On top, there is not yet a AWS server situated in Austria, the closest one so far is Hamburg (https://aws.amazon.com/about-aws/global-infrastructure/localzones/locations/), but our instance might as well be running elsewhere in the world.



```
Upload time:
0.6531457901000977
******** Processing image: 000000030581.jpg ********
{'bounding_boxes': [[0.3163018226623535, 0.3309292793273926, 0.7242182493209839, 0.6522942185401917], [0.13120535016059875, 0.5341729521751404, 0.84
32531356811523, 0.898196280002594], [0.3000345528125763, 0.32286518812179565, 0.7356736660003662, 0.6558308005332947], [0.292497843503952, 0.3047400
414943695, 0.7386120557785034, 0.6454174518585205], [0.22664666175842285, 0.142121359705925, 0.47343456745147705, 0.3275740146636963], [0.2891156673
4313965, 0.31999683380126953, 0.7308175563812256, 0.6472803950309753], [0.22664666175842285, 0.142121359705925, 0.47343456745147705, 0.3275740146636
963], [0.12261700630187988, 0.21605893969535828, 0.8148014545440674, 0.8518451452255249], [0.22664666175842285, 0.142121359705925, 0.473434567451477
05, 0.3275740146636963], [0.2384958267211914, 0.2738707661628723, 0.770604133605957, 0.6926171183586121], [0.3000345528125763, 0.32286518812179565,
0.7356736660003662, 0.6558308005332947], [0.664476752281189, 0.861615777015686, 0.7377314567565918, 0.8946022987365723], [0.22664666175842285, 0.142
121359705925, 0.47343456745147705, 0.3275740146636963], [0.5756241083145142, 0.7513484358787537, 0.8496408462524414, 0.9405654072761536], [0.8610513
210296631, 0.5345098376274109, 0.9080450534820557, 0.5967609286308289], [0.752040445804596, 0.8130885362625122, 0.8008868098258972, 0.8488559722900З
91], [0.883806049823761, 0.6072595715522766, 0.9259498715400696, 0.7121116518974304], [0.12261700630187988, 0.21605893969535828, 0.8148014545440674,
 0.8518451452255249], [0.6287552714347839, 0.8662492036819458, 0.7103301882743835, 0.8983025550842285], [0.6922786235809326, 0.8240079283714294, 0.7
494475841522217, 0.8573107123374939], [0.8567834496498108, 0.6116189360618591, 0.9104538559913635, 0.6979164481163025], [0.15317216515541077, 0.5593
932867050171, 0.8244094848632812, 0.9100042581558228], [0.2661422789096832, 0.3117443919181824, 0.7564308643341064, 0.9028266072273254], [0.23485101
759433746, 0.14673736691474915, 0.5477120280265808, 0.3295803368091583], [0.10926425457000732, 0.5320899486541748, 0.8237494230270386, 0.92215502262
11548], [0.2300490140914917, 0.14311382174491882, 0.47277772426605225, 0.33327752351760864], [0.8517504334449768, 0.5691519975662231, 0.905655801296
2341, 0.6259249448776245], [0.8918147087097168, 0.5738479495048523, 0.9276320934295654, 0.6466321349143982], [0.854440334358215, 0.6830285787582397
, 0.9039376378059387, 0.7432209253311157], [0.10254961252212524, 0.13211342692375183, 0.8791555762290955, 0.9206327199935913], [0.49761420488357544,
 0.8671041131019592, 0.768894374370575, 0.9816028475761414], [0.10926425457000732, 0.5320899486541748, 0.8237494230270386, 0.9221550226211548], [0.8
739078640937805, 0.501740038394928, 0.9030272364616394, 0.5368818640708923], [0.5676761865615845, 0.10766857862472534, 0.8599258661270142, 0.3210621
178150177], [0.5517042279243469, 0.8550831079483032, 0.821621835231781, 0.9912223815917969], [0.5688796639442444, 0.10719011723995209, 0.85002690553
66516, 0.3233802318572998], [0.6164862513542175, 0.6926424503326416, 0.8403472304344177, 0.8934783935546875], [0.572357416152954, 0.909319341182708
7, 0.6352466344833374, 0.9431217312812805], [0.7757609486579895, 0.7931276559829712, 0.8648377060890198, 0.8496754169464111], [0.5740013122558594, 0
.09991951286792755, 0.8789520263671875, 0.3173884153366089], [0.8899233937263489, 0.669590175151825, 0.9274764657020569, 0.7548524737358093], [0.145
37356793880463, 0.5246768593788147, 0.21901528537273407, 0.6043233275413513], [0.417894184589386, 0.9754537343978882, 0.527527332305908, 0.99451386
92855835], [0.8001512885093689, 0.7351312637329102, 0.8555907607078552, 0.814772367477417], [0.7192090749740601, 0.7040889263153076, 0.814182162284
511, 0.8413602113723755], [0.853216290473938, 0.802298367023468, 0.8984546661376953, 0.8409175276756287], [0.6280194520950317, 0.8827058672904968, 0
.7062113285064697, 0.9439721703529358], [0.7347404956817627, 0.8567865490913391, 0.786395788192749, 0.892936646938324], [0.5145418643951416, 0.96150
45189857483, 0.6046979427337646, 0.9933421015739441], [0.3651275634765625, 0.8643747568130493, 0.5232297778129578, 1.0], [0.6390971541404724, 0.7943
472266197205, 0.9053946137428284, 0.9776743054389954], [0.4635937809944153, 0.9766390919685364, 0.5768694281578064, 0.9978334307670593], [0.20254786
31258011, 0.1517562866210937, 0.45615214109420776, 0.32094788551330566], [0.6365966200828552, 0.0, 0.693032443523407, 0.02880751900374894], [0.344
194233417511, 0.9671439528465271, 0.4928237199783325, 0.9902917742729187], [0.2688082456588745, 0.9420785307884216, 0.7327888011932373, 0.9948537945
747375], [0.5140918493270874, 0.8979975581169128, 0.6102360486984253, 0.9444990754127502], [0.5371795892715454, 0.771560728549957, 0.81013703346252
44, 0.9181293845176697], [0.5860393047332764, 0.969308078289032, 0.6566629409790039, 0.9901074767112732], [0.5740013122558594, 0.09991951286792755,
0.8789520263671875, 0.3173884153366089], [0.32390618324279785, 0.2842451632022857, 0.6035618782043457, 0.7018615007400513], [0.15317216515541077, 0
.5593932867050171, 0.824409484863281, 0.9100042581558228], [0.3132288455963135, 0.8666349649429321, 0.47855234146118164, 1.0], [0.7756747007369995,
 0.636118471622467, 0.86093962192534, 0.6958186030387878], [0.6772157549858093, 0.5067518949508667, 0.910041630268096, 0.6529618501663208], [0.569
4377422332764, 0.6789183616638184, 0.8683394193649292, 0.9547351598739624], [0.5250251293182373, 0.8177000880241394, 0.6953839063644409, 1.0], [0.42
0112669467926, 0.863187909126281, 0.5847333073616028, 1.0], [0.10926425457000732, 0.5320899486541748, 0.8237494230270386, 0.9221550226211548], [0.5
927608013153076, 0.832921862602233, 0.8725366592407227, 1.0], [0.2810857892036438, 0.3283957242965698, 0.7246920466423035, 0.648023784160614], [0.7
537850141525269, 0.5274074077606201, 0.8827165365219116, 0.6055175065994263], [0.7161084413528442, 0.47558534145355225, 0.8988065719604492, 0.693656
325340271], [0.5809095501899719, 0.8066239356994629, 0.7524574398994446, 1.0], [0.8178167939186096, 0.6916609406471252, 0.8481282591819763, 0.727926
6715049744], [0.6794893741607666, 0.7221502065658569, 0.9072835445404053, 0.9003143310546875], [0.184897780418396, 0.4346723556518554, 0.7569543123
245239, 1.0], [0.04601261019706726, 0.7421910762786865, 0.7810858488082886, 1.0], [0.3361971378326416, 0.12338575720787048, 0.7045160531997681, 0.99
63396787643433], [0.12405946850776672, 0.6683927178382874, 0.4596797525882721, 0.9793898463249207], [0.6822401881217957, 0.7280614376068115, 0.87715
95358848572, 0.9903606176376343], [0.1443261355161667, 0.3289656639099121, 0.17025502026081085, 0.3730708360671997], [0.7728970646858215, 0.94871455
4309845, 0.8654316067695618, 0.9896784424781799], [0.06675183773040771, 0.056466519832611084, 0.9393129348754883, 0.9541866183280945], [0.0937836915
2545929, 0.938979983329773, 0.5054570436477661, 0.9983562231063843], [0.41499167680740356, 0.6789432764053345, 0.8418059945106506, 0.923601627349853
5], [0.0017554201185703278, 0.0, 0.11086107790470123, 0.9618204832077026], [0.6259780526161194, 0.812095046043396, 0.7086814045906067, 0.86325407028
19824], [8.642673492431641e-07, 0.26534730195999146, 0.13413947820663452, 1.0], [0.0486871600151062, 0.4801613986492157, 0.2584019601345062, 0.99915
15874862671], [0.8882097005844116, 0.5317026972770691, 0.9236507415771484, 0.5943419337272644], [0.35767388343811035, 0.9009926319122314, 0.44925898
31352234, 0.945626378059387], [0.5844225883483887, 0.0, 0.6572062969207764, 0.03007413260638714], [0.2515103816986084, 0.5222393274307251, 0.751709
3420028687, 0.802786946296691], [0.48308080434799194, 0.8607010841369629, 0.6323358416557312, 1.0], [0.19019387662410736, 0.9337368011474609, 0.645
1568603515625, 0.9969955682754517], [0.12414972484111786, 0.27077168226242065, 0.2879336476325989, 0.5099868774414062], [0.19019387662410736, 0.9337
368011474609, 0.6451568603515625, 0.9969955682754517], [0.5688796639442444, 0.10719011723995209, 0.8500269055366516, 0.3233802318572998], [0.8178167
939186096, 0.6916609406471252, 0.8481282591819763, 0.7279266715049744]], 'inf_time': '0.18188762664794922', 'status': 200}
Upload time:
0.8163230419158936
```

Fig. 3. Output for Image 000000030581.jpg with Upload time

Considering the major disadvantage regarding the upload time, running this application on a remote server location does not seem worthwhile. Even if we were able to use a much more capable instance to bring the inference time down and close to 0 while not changing something about the server connection and our upload setup, we still would be needing more than double the time compared to our local machine.

If however we were able to improve the upload setup and bring the upload time down to what our local machine needs while also increasing our inference capabilities, using AWS seems like a reasonable option, also when we are about to use even bigger data sets than ours.