Data Intensive Computing

Assignment 3 - Edge Computing

Group 02

Alexander Jell
Csaba Róbert Csécsy
Luis Kolb

https://github.com/LuisKolb/DataIntensiveComputingSS22

1. Information on how application has been developed, explaining the most important design choices and any modification to the Dockerfile or requirements.txt needed to make your implementation work

We started on the detection_loop first. After spending a lot of time trying to get tensorflow to recognize my (Luis Kolb) local GPU, development was continued on Google Colab, since it would otherwise have been impossible to work on the project.

After getting the object detection "tutorial" from the tf documentation to work (using the Inception Resnet v2 from tfhub), we adapted it to work as a function that takes as parameters:

- a file name (of the image to run detection on)
- a path to a locally accessible directory (where the image is stored)
- a boolean (whether to save the annotated image, detection results and metrics are always saved)

Reworking the Flask endpoint to call the function correctly was also done.

At a high level, the detection loop runs through these steps:

1. load and resize the image file at the provided path
2. run object detection on the image, which additionally records the inference time
3. optionally save the annotated image to <provided path>/output/
4. record the total detection loop time
5. write a dict as json with found objects, scores, the input file and path, and time metrics to the output path described in 3.
6. return the dict described above

So, the endpoint is up and running. Next, we wrote a script called sender.py which is called like in this example:

```
python sender.py /app/images/ --output --nodes 127.0.0.1:5000
127.0.0.1:5001 127.0.0.1:5003
```

This script will crawl the directory passed to it, and for each image file in it, run a curl command. The curls are distributed evenly among all the "nodes" provided to the script. Distributing the load among multiple containers running an endpoint was the idea behind this script, but we ended up running only one container due to limitations on the cluster. The endpoint can, of course, still be called manually, but the sender.py script is still very convenient, especially to run multiple images through detection.

More info on the script can be obtained using

```
python sender.py --help
```

Next, trying to execute the script inside the container threw an error. Apparently, some image files were corrupted such that PIL.Image.open() couldn't open them. So, we wrote yet another script that crawls a directory of image files, and reports the names of the images where PIL.Image.open() throws that error. 2 files ended up being corrupt, removing them resulted in a successful first run.

On the final run, we ran detection on 827 images from the dataset. Results below and in the github repository linked on the title page.

The Dockerfile was modified such that the container could access a local directory as a volume, so we wouldn't have to copy the image files into the container, as well as preserving /output/ across container instances and runs. We also use the base image "tensorflow/tensorflow:latest".

requirements.txt was frozen from a working installation using pip.

2. Commands needed to deploy your container

All necessary commands can be found in README.md, here are a few important ones:

Transfer the repository to the cluster

```
git clone https://github.com/LuisKolb/DataIntensiveComputingSS22.git

scp -r ./DataIntensiveComputingSS22
group02@s25.lbd.hpc.tuwien.ac.at:/home/group02
```

Building the container

```
docker build -t dic_assignmentg02 .

docker run --name dic_assignmentg02 -v ~/dic02-images:/app/images -d -p
5002:5000 dic_assignmentg02:latest
```

Curling the cluster externally

```
curl http://s25.lbd.hpc.tuwien.ac.at:5002/api/detect -d
"input=/app/images/000000000019.jpg&output=1
```

Getting a shell inside the container (necessary for sender.py, since the image directory needs to be locally accessible)

```
docker exec -it dic_assignmentg02 /bin/bash
```

In this example, we run the sender.py script inside the docker container, hence localhost url

```
python sender.py /app/images/ --output --nodes 127.0.0.1:5000
```

3. Explanation on how you calculate inference and transfer time

Inference time is calculated using time.time() inside the function run_detector(). The other time we record is the total time detection loop is taking, using time.time() as well.

Since images are locally accessible, and not transferred across the internet as part of the request, we didn't record this per image. However, uploading the 827 images at 1.1MB/s took around 3 minutes.

4. Information on your experimental setup. i.e., CPU power, network bandwidth and any additional hardware (GPU) that you use to run your experiments

The hardware we used for our experiments are the resources afforded to the container running on the LBD cluster, since running it locally was not possible on my (Luis Kolb) desktop.

5. Data about local/remote inference time and transfer time

Again, we will ignore local time metrics and transfer time, since they don't apply to this task. Looking at execution times per image on the cluster, we get the following results:

images analyzed:	827	
mean inference time:	3.022669746913241	seconds
mean total loop time:	3.0845179407887766	seconds

More analysis can be found in the jupyter notebook located in **output.zip** (see repository), which is great for poking around our results, or in the **dic-ex3-analysis.html** file. The .zip file also contains all the annotated images (~50MB all in all).

6. Comments on your execution: is it worth to offload execution on the remote cluster? If not, why? What would be needed to improve performance of remote and

local execution? Can you think of a scenario where offloading improves performance?

Given the limitation that tensorflow is incredibly complicated to install and to get up and running, we even “offloaded” our initial development and experimentation to preconfigured Google Colab workspace. Offloading the container onto the cluster, instead of running it locally, was pretty much the only option we had open to us. Moreover, not everyone even has access to a GPU supported by CUDA, which is why I (Luis Kolb) appreciate Google Colab a lot.

Considering that the Inception Resnet v2 is supposedly a larger and more complex and powerful model than some alternatives, detecting objects in an image and drawing boxes around them took (only?) around 3 seconds. You could definitely do better using more resources and a lot of optimization, but these times seem acceptable for a project of this scope.

Cutting execution times would be most easily achieved by either using a smaller model or dedicating more resources to the existing container.

Regarding data “offloading”: transferring the images to the cluster once took around 3 minutes, and we then accessed them again and again, which seems well worth the while upfront, especially during development. Sending every file per http request would have been a huge overhead, though some of the logic around loading/saving the files would have been vastly simplified (on the server side, at least).

Again, all the code can be found in the github repository at
https://github.com/LuisKolb/DataIntensiveComputingSS22
(will be made publically available after the deadline)

Summary of the theoretical background

Convolutional Neural Networks

A neural network is a directed graph, where the nodes can be grouped by the step they are used in the algorithm - such a group called neural layer, and each node is called a neuron, as these algorithms were inspired by the functioning of the human brain. The input of the first layer is the original data or the data resulting from featurization - in case of image processing, the latter. The output of each layer is the input of the following layer (except for the last layer). A convolutional layer transforms the features step by step with kernels that are smaller matrices as the given input matrix, and the output of each such a kernel will be one element of the output matrix. The output of a kernel is the weighted sum of the elements of the corresponding matrix. The weights are trained with the application of backpropagation, a supervised learning using gradient descent (which is an optimization algorithm for finding local minimum of differentiable functions).

Faster R-CNN

Region Based Convolutional Neural Networks (R-CNN) are machine learning models for computer vision and object detection. They produce a set of bounded boxes with categories for the given input image where each box is supposed to contain an object that is relevant to the given detected category.

Faster R-CNN applies Selective Search Algorithm, which has two parts: sub-segmentation of the input image and their recursive combination to gain bigger segments. A graph based image segmentation is applied described by P. F. Felzenszwalb: the nodes of the graph are the pixels, and the edges are weighted based on dissimilarity between the corresponding two pixels, where the dissimilarity is described by different features such as intensity, color, motion and location. The following recursive combination is a Greedy Algorithm, meaning the most similar regions are combined. For the classification, Support Vector Machine models are applied.

Inception-ResNet v2

Inception-ResNet v2 refers to the behavior of the convolutional neural network: residual connections are applied that provide alternative connections to neurons from further layers besides neurons from the following layer. The aim of these extra connections is to decrease the possibility of vanishing or exploding gradients during backpropagation, and thus to make it easier to build neural networks with huge amounts of neural layers.