Few-shot roadworks detection

DCAITI

Louis Landelle, Yaron Dibner, Alexandre Rozier

Contents

1 Introduction **1**

2 State-of-the-art Convolutional Neural Net **2**
2.1 TensorFlow . . . 2
2.1.1 Origins . . . 2
2.1.2 Functioning . . . 2
2.2 TensorFlow Object Detection API . . . 3
2.3 What's inside the TensorFlow repositories ? . . . 5
2.4 Installation instructions . . . 5
2.4.1 Pet detector . . . 6
2.4.2 Our implementation . . . 7
2.5 Using the user interface . . . 10
2.5.1 Description . . . 10
2.5.2 Usage . . . 10
2.5.3 Outputs . . . 12
2.6 Discussion . . . 13

3 Shaban & Al. **15**
3.1 What can be found in the Shaban & Al. directory? . . . 16
3.1.1 Hand-written utilities . . . 16
3.1.2 Shaban & Al. Source Code - `shaban_al/` . . . 17
3.1.3 voc2012/ . . . 19
3.1.4 snapshots_shaban/ . . . 19
3.1.5 ellipse_generator/ . . . 19
3.1.6 LABELLATOR/ . . . 19
3.1.7 DeepMask/ . . . 19
3.2 Installation instructions . . . 19
3.2.1 Base installation . . . 19
3.2.2 Braxton-centric Installation quirks . . . 20
3.3 Results . . . 20
3.3.1 Intersection over Union results on the Warnbake class . . . 20
3.3.2 Deeper model analysis with Ellipses . . . 20
3.4 Discussion . . . 22

4 Discussion about the two alternatives **22**

References **23**

1 Introduction

The first electronic digital programmable Turing-complete computer was built in the USA in 1945. It weighed 30 tons and could add, subtract, multiply and divide 5000 times a second, which for the time was a lot [1]. Since then, computers have shrunken to fit our pockets and are able to execute millions of operations per second. This progress in the domain of computer science has enabled modern machines to perform a wide range of tasks, which has flood our homes, offices and laboratories and automatic computing machines. Computers help us with our everyday tasks, such as communicating with relatives, checking the weather, or taking a picture. They are also useful for complicated matters, such as research, the broadcasting of information to millions of people, or the tracking of an endangered species. However, for the most part of the computer era, humans have stayed superior in terms of intellectual abilities, for example with games requiring intellect or pattern detection.

This is however changing. The first computer won against a human at Chess in 1997 [2]. This event shocked the world, as people were claiming that computers couldn't master a such complicated game. Since then, computers have surpassed people in numerous other games like Go in 2016, or more recently Dota 2, a very complex game containing much more parameters and possibilities that board games [3].

These achievements were done by what's called an artificial intelligence. An artificial intelligence, according to Wikipedia, is "any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals." [2] The field of AI has exploded in the last decade, with one of its most popular sub-fields : machine learning. In machine learning, statistical theory is used to enable the computer to learn with data, i.e. enhance its performance on a certain task [4]. The important innovation is that the computer isn't explicitly programmed to perform well on the task, it "simply" adjusts a lot of parameters (three millions in some modern machine learning algorithms [5]) in order to improve its performance.

Many of the tasks we want to achieve today are related to images. Indeed, one of the abilities that is qualified as intelligence is the detection and classification of objects in images. Today, some programs can detect objects with similar, or even superior precision than humans. This, however, requires thousands, or even tens of thousands of training images, each labeled by hand. The time necessary to prepare the data, and then train the network, can sometimes extend for years, as gathering the data is a hand-made process. Also, to that is added the cost of the processing units, such as GPUs, on which the training will run.

Humans, on the other hand, are very efficient in the amount of images required to learn how to categorize a new object. If a child sees two goats on the side of the street, and its parents say "Look ! That's a goat ! A goat !", the child will be able to repeat the word the next time it sees a goat, even if the goat it sees is a completely new one. Even if the goat is black and the previous one was white. Even if the goat is turned to the right and the previous one was turned to the left. If the child does a mistake and calls a goat a cow, the parents will rectify him, and the kid will make the difference from that moment on.

Hence, even though modern computers can compete with humans for object detection or classification, the learning part is still much more efficient and quick for humans. And even if remarkable techniques were developed to build and label databases, the resources required to run a training are still out of reach for the common Joe, who is stuck with a CPU, and if he's lucky, a quick GPU, but which will still need weeks to process a whole database.

This problem of today's machine learning is what this project is about. Its aim is to train existing neural networks on less than ten images and evaluate their accuracy. The chosen object is the roadwork sign, an everyday object which assimilates to the domain of self-driving cars. The success of an autonomous car doesn't only rely on its ability to detect known objects, but also to adapt to new situations. If the german minister of transport decides to add a new traffic sign, it would be tedious to collect thousands of images of that traffic sign and incorporate it in the car. It's possible, but too slow. However, if the algorithm in the car only needs to see five such traffic signs to be able to have learned this new category, like a human would do, then the car would almost instantly adapt to the new regulation.

This is why this project is being realised, and why the future of machine learning will probably be oriented on few-samples data training.

In this project, two main approaches were selected.

1. The first one is a few-shot implementation by Shaban & Al. [6]. They propose a model able to perform object segmentation on any class given few examples (called the support set), with a relatively good average Intersection over Union score of 40.8 on one-shot tasks. The goal here is to evaluate how this model performs on a new class, Warnbake, to see if it can be used in the context of autonomous driving.

2. The second one is a few-shot fine-tuning of a state-of-the-art object detection implementation, which isn't supposed to perform well. The chosen SOTA implementation is TensorFlow. It is a neural network developped by Google which allows to quiet easily train the network on any database. The original aim of the SOTA implementation was to prove the better performance of a few-shot implementation, but as will be seen subsequently, we were left with quiet a surprise.

At the end of this report, we will compare the performances of those two models and try to provide more insight on their possible use cases. Both were studied at the same time, and we later focused on the most promising one.

2 State-of-the-art Convolutional Neural Net

The SOTA implementation is done with the TensorFlow Object Detection API. It is built on top of TensorFlow, the open source machine learning framework, and facilitates the training and construction of object detection models [9].

To understand the TensorFlow Object Detection API, it is important to understand how the TensorFlow model is designed.

2.1 TensorFlow

2.1.1 Origins

In 2011, the department of Google specialized in deep learning artificial intelligence, Google Brain [13], started the development of a deep learning neural network system called DistBelief. This system was designed to use big clusters of computers for large models training. Alongside this framework, two algorithms for large-scale training were developed :

1. Downpour SGD, for performing asynchronous stochastic gradient descent

2. Sandblaster, a framework supporting various batch optimization procedures

With this system, Google successfully carried out the ImageNet task, which was trained on 16 million images and 21 thousand categories. [14]

Computer scientists were then assigned the task of simplifying the DistBelief framework into a more robust application, which became the version 1.0.0 of TensorFlow . The framework can run on across many different platforms (GPUs, CPUs, TPUs), servers, personal computers or even mobile devices running Android or IOS thanks to its flexibility [12]. The TensorFlow API was released as an open-source package under the Apache 2.0 license in November 2015. According to Google, TensorFlow is much more flexible and quick than DistBelief, and almost every previous user of DistBelief, being a part of Google or not, has already switched to TensorFlow today. [15]

2.1.2 Functioning

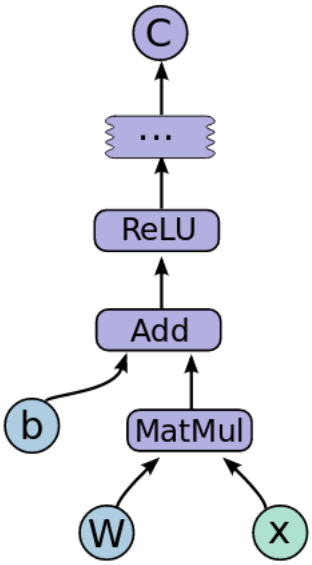
A TensorFlow calculation is described by a directed graph, in which nodes are able to keep states. In that sense, TensorFlow implements a classical neural network, in which some instances have taken new names: values propagating along edges are called *tensors*, for example. With the TensorFlow Python API (documentation), it is for instance possible to create a simple graph using the following Python code :

Listing 1: TensorFlow example

```
import tensorflow as tf
b = tf.Variable(tf.zeros([100]))              # 100-d vector, init to zeroes
W = tf.Variable(tf.random_uniform([784,100],-1,1)) # 784x100 matrix w/rnd vals
x = tf.placeholder(name="x")                  # Placeholder for input
```

```
relu = tf.nn.relu(tf.matmul(W, x) + b)         # Relu(Wx+b)
C = [...]                                      # Cost computed as a function
# of Relu
s = tf.Session()
for step in xrange(0, 10):
    input = ...construct 100-D input array ...  # Create 100-d vector for input
    result = s.run(C, feed_dict={x: input})     # Fetch cost, feeding x=input
    print step, result
```

The graph obtained with the code above is :



Source: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45166.pdf

Hence we obtain a graph for which it is possible to give an input and obtain an output. It is subsequently possible to, for instance, perform cost calculations in the case of training. TensorFlow also lets the user chose on which devices the computations are going to be done: on a single process, on multiple processes, with or without cross-device communication, with or without clustering of devices, and so on. [15]

TensorFlow also comes with TensorBoard, a tool for visualizing graph states, network performances and loss graphes. C.f. Fig. 1.

This project however wasn't about creating our own model. This is a tedious process on which a lot of top-level researchers spent hours and hours, and we don't pretend to have the capacity nor the time. A lot of models already exist and are available to the public, such as Faster R-CNN, SSD, R-FCN, and many more. For our SOTA implentation we rely on Faster R-CNN with Resnet-101.

2.2 TensorFlow Object Detection API

The TensorFLow Object Detection API is built on top of TensorFlow and provides a framework for object detection. Google designed it for two purposes : support state-of-the-art models, as well as research and exploration. The SOTA model we use, Faster R-CNN with Resnet-101, is constructed the following way :

1. A region proposal network (RPN). Features are extracted from the images and used to select bounding boxes for high interest points. Usually 300 such boxes are selected. C.f. Fig. 2b
2. Box classifier. The extracted features are used to predict the class of each box.

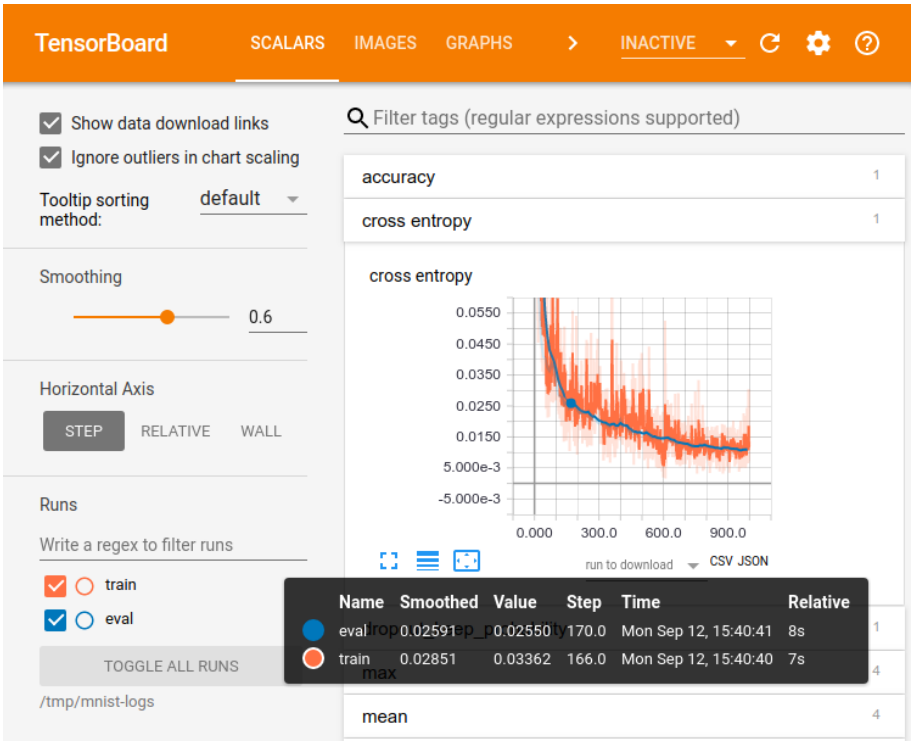
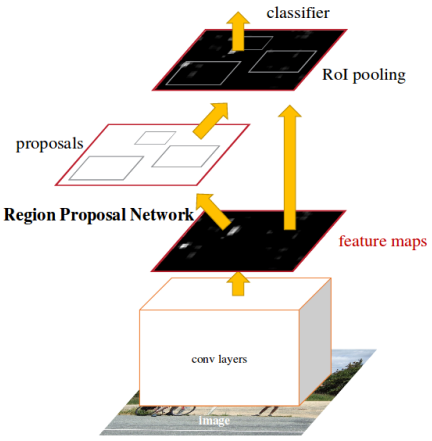
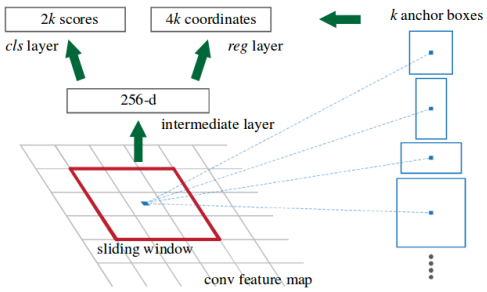


Figure 1: TensorBoard interface

Source: https://www.tensorflow.org/guide/summaries_and_tensorboard



(a) Faster R-CNN network



(b) Region proposal network

Figure 2: Faster R-CNN architecture

2.3 What's inside the TensorFlow repositories ?

The first repository needed for this project is the TensorFlow repository. It can be found here. This repository contains all the necessary files to execute and build a TensorFlow graph [17]. The API is available for many languages, but we are using it in Python, because this version of the library is the most complete, and Python is a language we all have some experience in. The installation instructions will be detailed in the following subsection.

The homonym sub-folder which is of interest for us is /tensorflow/tensorflow. It contains the code for the various languages API, such as Java, C or Python, but most importantly it is the folder that will contain the second repo that we will use.

The second one is the TensorFlow Models repository. It can be found here. It contains different models implemented with TensorFlow broken up into four folders:

- Official models. These are up-to-date regarding the latest stable TensorFlow API and well-maintained. They use the high-level APIs and must keep a balance between fast performance and easy understanding of the implementation. We didn't use these models.
- Research models. These are models implemented by researchers, and which purpose is to be tested and messed around with. These are the models that interest us, as the Object Detection API belongs to them.
- Sample models. These contain snippets and code examples that demonstrate the capabilities of the TensorFlow API.
- Tutorial models. These are the models described in the TensorFlow tutorial.

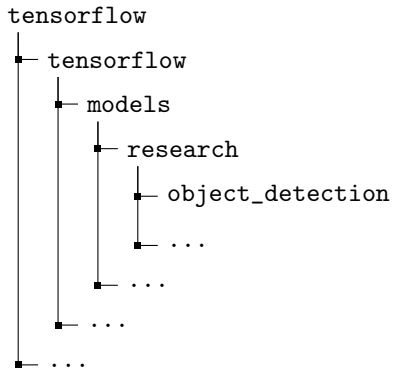
Inside the models/research sub-folder, the only sub-folder which interests us is object_detection. It contains the whole TensorFlow Object Detection API. The sub-directories that we used are the following :

- data. It contains pet_label_map.pbtxt file that we used with the pet detector. More will be said about this in the pet detector installation tutorial.
- dataset_tools. This folder contains helper functions to generate .record files. We used create_pet_tf_record.py for the pet detector, and later modified it to execute it on our dataset. This allows to transform a database containing .jpg and .xml files into a serialization of the data readable by the model. More will be explained about this in the pet detector installation tutorial.
- samples/configs. This folder contains .config file examples, and especially the faster_rcnn_resnet101_pets.config file. This file is passed to the network so that it knows where to get the .pbtxt file, the dataset images and annotations, the model checkpoint, the .record files, and more execution related parameters. It was used for the pet detector, and was later remodeled to be used with our dataset.
- models. Initially empty, this folder contain all the checkpoints and metadata for TensorFlow once train.py or eval.py are run for the first time. This is configured in the .config file.
- utils. It contains helper functions. dataset_util.py was used for the .record file generator.

2.4 Installation instructions

To install the TensorFlow Object Detection API, first TensorFlow must be installed. It was already installed on the Fraunhofer Institut's server machines that we used. The steps required to install TensorFlow with GPU support on Ubuntu using a virtualenv are described in the installation tutorial.

To use the models that are in the TensorFlow Object Detection API, two github must be cloned in a specific manner. First, the TensorFlow repo can be cloned anywhere in the system. Then, the TensorFlow Models repo **must** be cloned in tensorflow/tensorflow/. Here is how the repo should eventually look :



Now that the repositories have been cloned correctly and TensorFlow has been installed, it is time to install the TensorFlow Object Detection API. The install tutorial explains how to install the dependencies, the COCO API if you want to use the COCO evaluation metrics and how to compile the Object Detection API with Protobuf. Almost all the commands must be executed from the **tensorflow/models/research/** directory. It is important to update the PYTHONPATH as follows :

```
$ export PYTHONPATH=$PYTHONPATH:‘pwd‘:‘pwd‘/slim # from tensorflow/models/research
```

This must be executed every time a new terminal is opened, or to add this command to the .bashrc file. The installation can then be tested with :

```
$ python object_detection/builders/model_builder_test.py
```

If the installation was done correctly, the output will contain "OK".

2.4.1 Pet detector

To test that everything was good to run, we tried our installation on a pet detector that is provided by TensorFlow. The tutorial explains how to train the pet detector using Google Cloud, however this isn't necessary, and the training can be done locally, which was we did. Here are the steps to make the training run on a local machine :

1. Install the dataset to the research directory :

   ```
   # From tensorflow/models/research/
   wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
   wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
   tar -xvf images.tar.gz
   tar -xvf annotations.tar.gz
   ```

   An other directory can also be chosen.

   If tensorflow/models/research was chosen, here is how the directory should look now :

   ```
   - images.tar.gz
   - annotations.tar.gz
   + images/
   + annotations/
   + object_detection/
   ```

2. Generate the .record files :

   ```
   # From tensorflow/models/research/
   python object_detection/dataset_tools/create_pet_tf_record.py \
   --label_map_path=object_detection/data/pet_label_map.pbtxt \
   --data_dir=‘pwd‘ \
   --output_dir=‘pwd‘
   ```

   Careful, if another directory was chosen, the paths must be changed accordingly, and the ‘pwd‘ must be changed to the path of the directory where the data is.

3. Download the COCO-pretrained model :

```
wget http://storage.googleapis.com/download.tensorflow.org/models/object_dete
    ction/faster_rcnn_resnet101_coco_11_06_2017.tar.gz
tar -xvf faster_rcnn_resnet101_coco_11_06_2017.tar.gz
```

4. Configure the object detection pipeline. For that, modify then faster_rcnn_resnet101_pets.config file in object_detection/samples/config by replacing every instance of PATH_TO_BE_CONFIGURED with the right value.

5. Start the training by executing the following python script. The question marks "???" should be replaced with the correct paths :

```
import os
import sys

TRAIN_PY_PATH = "???" # absolute path to object_detection/train.py python script
PIPELINE_CONFIG_PATH = "???" # absolute path to
    object_detection/samples/config/faster_rcnn_resnet101_pets.config
TRAIN_DIR_PATH = "???" # absolute path to the directory where the training data will
    be stored, typically research/models/model/train

os.system("python "+TRAIN_PY_PATH+" \
    --logtostderr \
    --pipeline_config_path="+PIPELINE_CONFIG_PATH+" \
    --train_dir="+TRAIN_DIR_PATH+"n/")
```

Training checkpoints are automatically saved. The execution can be stopped at any time by pressing Ctrl+C. To see the loss graphs and the general evolution of the training, TensorBoard can be launched with the following command:

```
$ tensorboard --logdir=. # from models/model/train/
```

The graphical interface of Tensorboard can be typically accessed from localhost:6006. However, if you are connected to a server, you need to bind ports from your local computer to the server in order to access the interface. In our case, we are connected to braxton.fraunhofer.fokus.de. To bind the ports, the following command should be used when connecting :

```
$ ssh -L 16006:127.0.0.1:6006 username@braxton.fokus.fraunhofer.de -p 30022
```

This means that the port 6006 of the server will be bound to the port 16006 of the local computer. Hence, after launching the TensorBoard command from a terminal on the server, the interface can be accessed through the local machine from localhost:16006.

2.4.2 Our implementation

2.4.2.1 First step : generating .record files containing the dataset The .record files serialize the dataset in multiple parts used by tensorflow to input datapoints into the input tensors of the graph. We need to split our dataset into two records, train.record and eval.record. Train.record will be the datapoints used for the training iterations of the neural net, the eval.record will be used punctually to evaluate the performance of the neural net, and to make a decision on whether or not the neural network has a sufficient performance to be used as-is and to stop the training iterations.

Using other record-generating programs provided by tensorflow tutorials, a custom record-generating script was written to serialize our datasets (one provided by Johannes, one compiled from CC-by licensed images found on multiple internet sources) into the VOC standardized format.

Another script was written to read a record and insure the image metadata was correctly serialized, and to insure the image bytes serialized in the record were able to be restored as the original image.

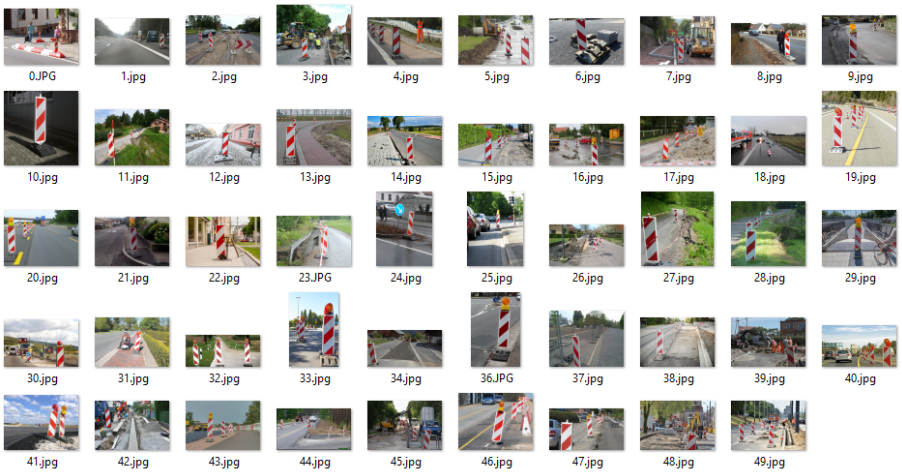


Figure 3: The first internet-compiled dataset

Figure 4: The second dataset containing images from a real single car drive.

To write the record of images and metadata of labeled classes in some folder, the script write.py is used:

```
python write.py /my_dataset
```

This script assumes a root folder structure of : root/images containing the images and root/labels containing XML VOC formatted metadata, named exactly like the images are in root/images. It generates output.record in the location of the script.

To read the record generated by this file, one can use the read.py script which outputs to the terminal its contents:

```
python read.py my_record.record
```

Those scripts were used to generate train.record and eval.record for the two different datasets used.

2.4.2.2 Second step : Training the neural net on train.record The training script is provided as-is by the object detection API. It reads in the configuration file the input record file location and the output path for training checkpoint generation, metadata writing, etc. The runs on the datasets were done on the Fraunhofer Institute GPU at the beginning, and for the internet-sourced dataset, locally on a 2700x AMD CPU for 10k iterations in about 24h. Both training methods yielded well performing graphs.

Using tensorboard we were able to monitor the evolution of the total loss function which was used, along with evaluation runs, to determine when stopping the training was possible.

Figure 5: Evolution of losses functions.

2.4.2.3 Third step : Evaluating the neural net performance with eval.record The evaluating script is provided as-is by the object detection API. Similarly to the training script, it reads in the configuration file the input record file location and the output path for evaluation results.

Figure 6: Example of evaluation run result.

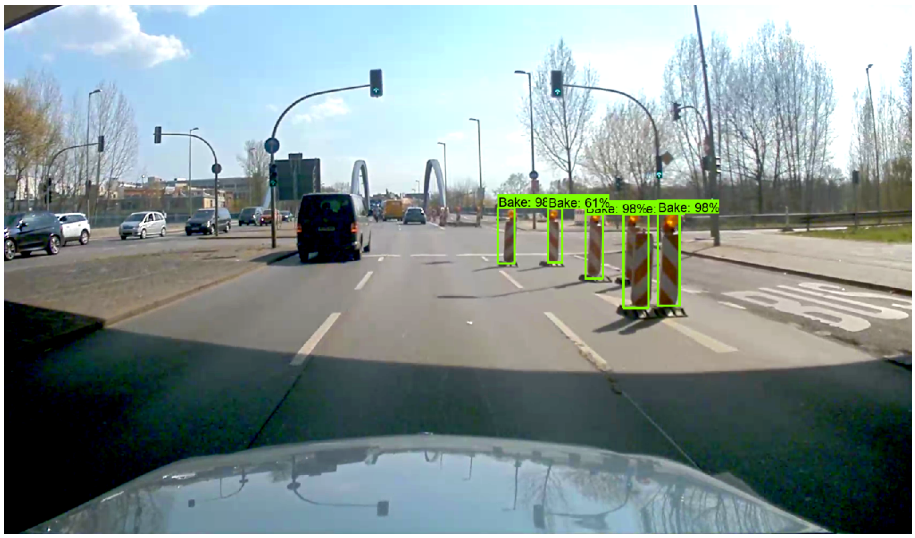


Figure 7: Example of evaluation run result.

Evaluation was performing on a record batch the work we wanted, but to deploy the neural network in a custom application another step is needed : freezing the inference graph and using a frozen inference graph inside anapplication.

2.4.2.4 Fourth step : Freezing and using the inference graph The freezing script is provided as-is by the object detection API. It uses the latest training checkpoint and serializes the graph and its current weights into a single file ready to be loaded into the memory and used by the tensorflow library. Its output is the frozen_inference_graph.

Now, the frozen inference graph is ready to be utilized and deployed within a program. The following is a succinct, summarized script showing how to use a frozen graph file within a program to infer on a single image, a complete version of the code is available in infer2.py, the back-end of the user interface program.

```
def run_inference_for_single_image(image, graph, score_thres=0.5):
    # Graph is a frozen inference graph loaded into memory. Open a tensorflow session.
    with graph.as_default():
        with tf.Session() as sess:
            # Get the image input tensor
            image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0')
            # Run the session feeding the image to the image tensor
            output_dict = sess.run(tensor_dict,feed_dict={image_tensor:
                np.expand_dims(image, 0)})
            # Omitted: formatting output_dict into detections, a list of triplets (class,
                rect, score)
    # Returned : filter results to keep those whose confidence score is above user chosen
        score threshold
    return [d for d in detections if d[2] >= score_thres]
```

2.5 Using the user interface

2.5.1 Description

The showcase program made to use this frozen inference graph will be a simple user interface front-end on a simple back-end of running inference on a single image. The program would be able to load an image, run inference with a user inputed confidence threshold on the loaded image, and output a new image with detected warnbaken and serialize both this new image and a .json file with the more precise triplets (rectangle, label, confidence) for the image.

The front-end was written in python with tkinter, the default library for user interface. The front-end was split off from the back-end through the two files frontend.py and infer2.py (the back-end).

2.5.2 Usage

Before using the program, one needs to find 3 paths on one's machine:

1. The path to the $tensorflow/tensorflow/models/research$ folder.
2. The path to the frozen graph file ($frozen\_inference\_graph.pb$).
3. The path to the label map file ($bake\_label\_map.pbtxt$).

In $frontend.py$, the second line adds the models/research root to the sys.path variable. This should be changed to your specific path (1).

In $infer2.py$, $PATH\_TO\_CKPT$ should be changed to your path to your frozen graph file (2) and $PATH\_TO\_LABELS$ should be changed to your path to your label map file (3).

Now, the program can be launched through :

```
python3 frontend.py
```

Nb.: For tkinter compatibility, Python 3 needs to be used instead of Python 2. The program is stable on Python 3.6.5. Compatibility with Python 2 will require a little bit of retrofitting.

The program can load an image with File > Load image. The user can tweak the threshold for confidence scores, and use Run to start inference on the loaded image. The program saves the result image and json at the frontend script location.

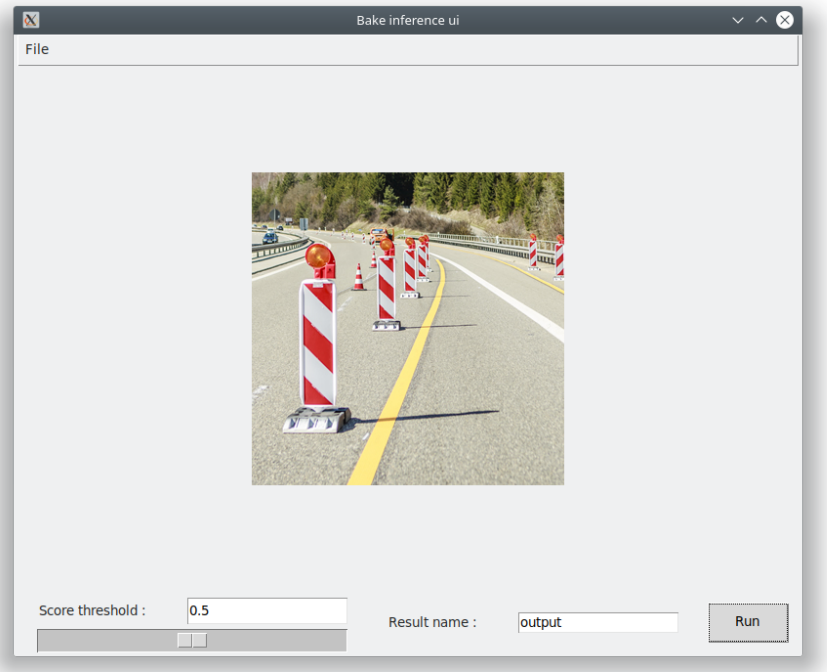


Figure 8: Loaded image, ready for inference.

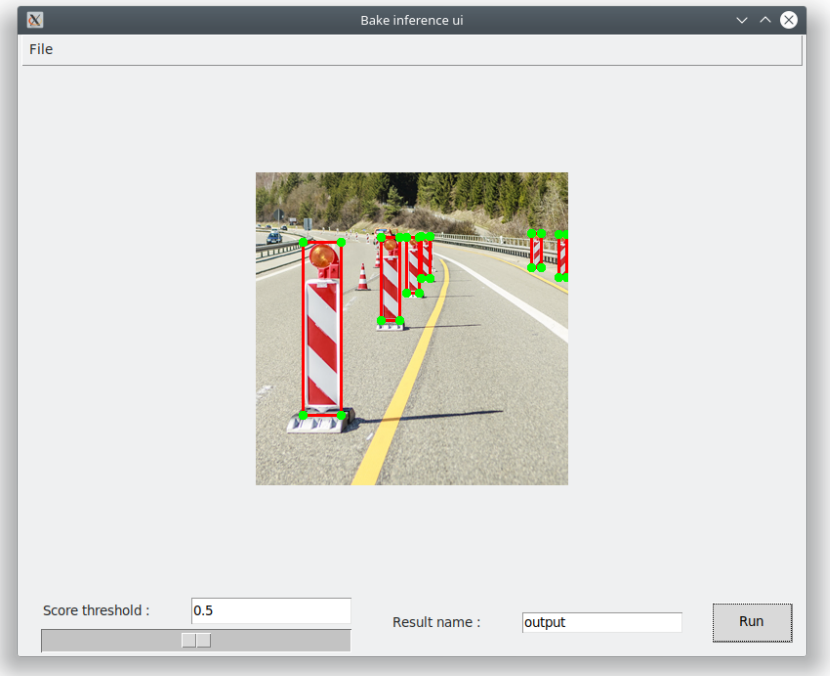


Figure 9: After the inference : showing the results representation.

2.5.3 Outputs

Figure 10: The image output of the program, visible on the UI too.

All rectangles are generated by the neural network, as the detection of class 1 (Warnbake) with a confidence score above the threshold. The more precise information are serialized in the .json file.

Figure 11: The second output, a .json file containing all the available info from the neural net

- The class value is the index of the label, 1 being here mapped to a Warnbake, and the only available class.
- The rect is a quadruplet on $[0; 1]^4$ representing the two-corners of the rectangle relative to the width and height of the image (thus need to be multipled by the (w, h) to find pixel values of the rectangle)
- The score is the confidence value of the detection. All scoring above the user-inputed threshold will be serialized.

2.6 Discussion

One using the demo software or using the frozen neural net on one's own dataset will notice how good the performance of state-of-the-art network is on many images in many lighting/quality/resolution conditions, even with "small" datasets of 11 and 25 training images for respectively the given dataset of the single car run and the internet compiled images.

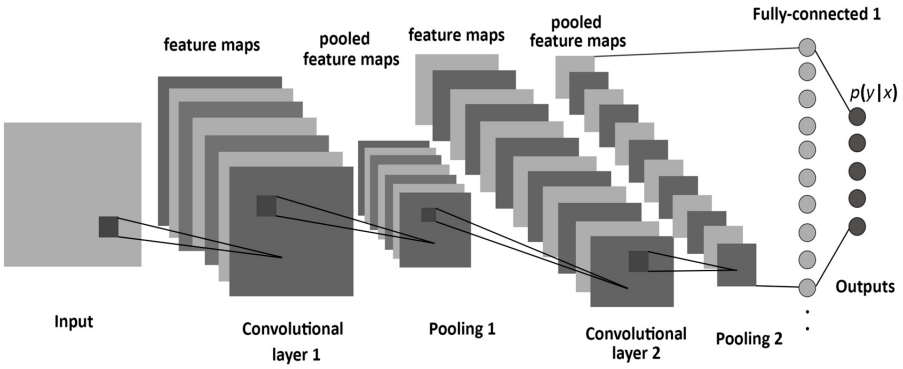


Figure 12: A CNN standard architecture : the last layer is the one fully connected to the output tensors.

Source: http://www.mdpi.com/1099-4300/19/6/242

To understand why this is the case, we need to go back to the mechanism of retraining the last layer of a neural network : - the first part, sometimes called "feature extraction", transforms all the pixels values into a feature vector via successive partially connected layers. - The last part, that we will call "inference layer", fully connects the feature vector to whatever the output data structure is.

The feature extraction is not the part making a difference when training two different CNNs to detect, say, cats or apples, instead the part with different weights for the two goals is the inference layer. Therefore the work to train this large feature extraction part can be skipped by using a general feature extractor trained on a large dataset for a large amount of classes, this part is generically good at extracting features, and can be used in any CNN.

Thus today the consensus state-of-the-art CNN for object detection, image classification, etc. is built on a retraining basis, where the feature extractor is never trained, only the inference layer.

The feature extractor that we used for the object detection CNN is trained on MS COCO, which contains more than 200k images containing objects of 80 labeled classes. One of those classes being the STOP road-sign, the feature extraction is already highly trained to extract features relevant for STOP road-sign detection.

Figure 13: The 80 classes with the STOP road-sign class.

Source: http://cocodataset.org/#explore

The warnbake is very similar to a STOP road-sign, thus the neural net does not need to learn "from scratch" how to detect a warnbake. It is more about some fine-tuning. Most objects in the world (that are targeted by neural nets applications today) are similar to one of the 80 classes of MS COCO, thus most neural nets to detect them can "cheat" the few-shot behavior by using a pretrained feature extractor in this way, and fine-tuning detection via the last layer retraining.

3 Shaban & Al.

This approach is based on the One-Shot Learning for Semantic Segmentation paper [6] (Shaban & Al.), and makes the bet to teach a neural network to assimilate class characteristics only by looking at a few instances of the selected classes.
They propose a neural network composed of three main component:

1. A FCN-32s-based branch, taking as input a query image (the image from which a segmentation mask should be generated). Its role is to extract features of interest F_{query_image} from this query image and output it. One should be aware that this branch has almost fixed weights, which means it will be just slightly changed after the training phase.

2. A VGG-based branch, which is trained to generate good meta-parameters (w, b) that will be used during the generation of the segmentation mask. It uses as inputs a support set (set of images and associated ground truth masks that will be used for training!), and outputs parameters that will be used to predict a mask in the last component

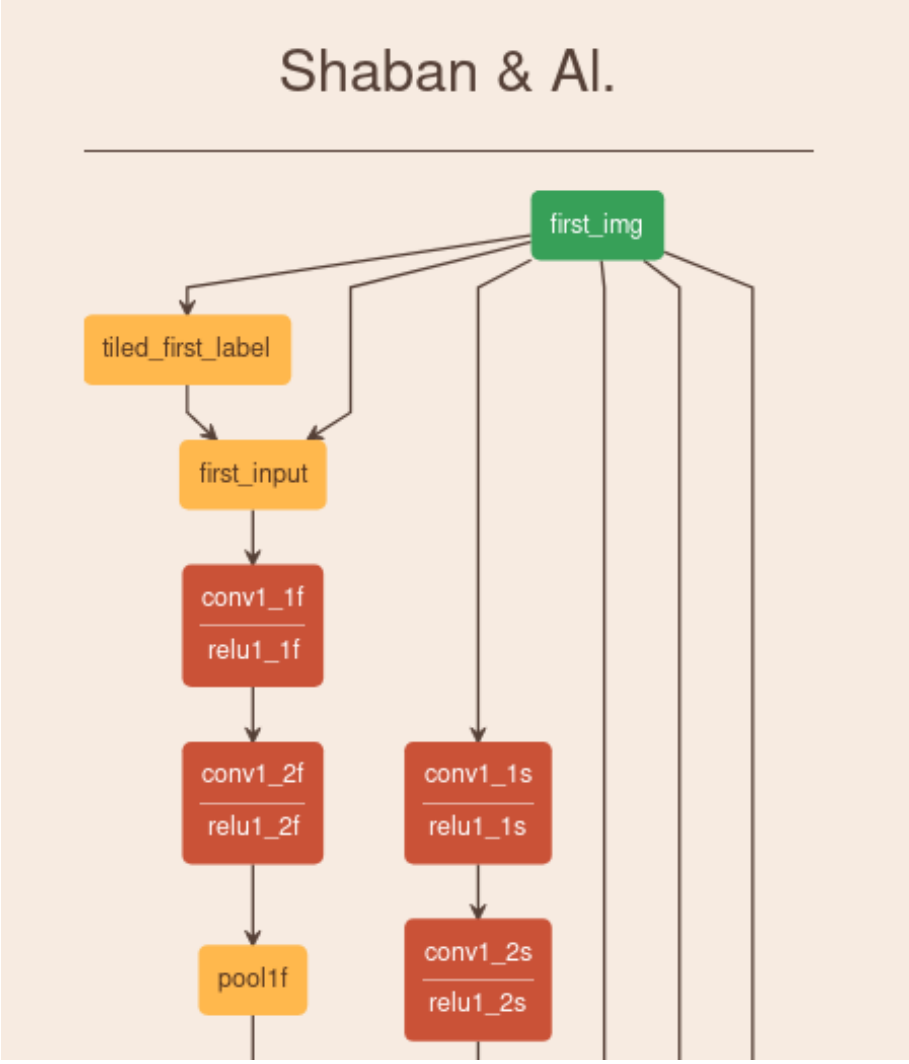


Figure 14: Model Architecture. The leftmost branch is the FCN-32s, and the second one is the VGG branch.

3. An activation function, as simple as

$$\hat{M}^{mn} = \sigma(w^\top F^{mn} + b). \tag{1}$$

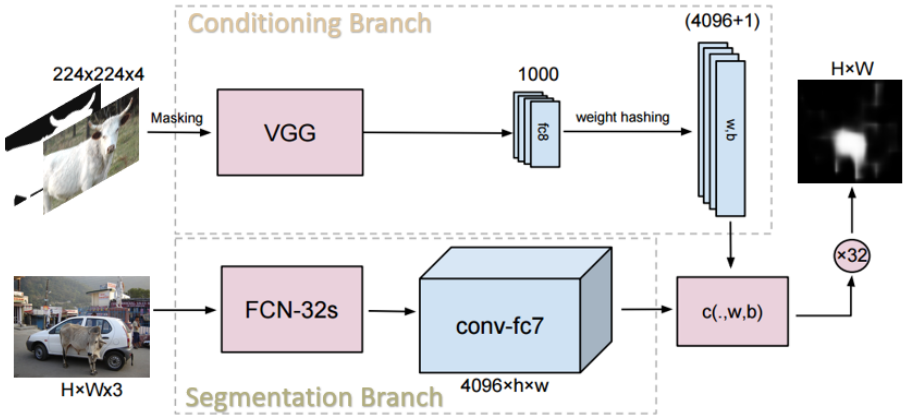


Figure 15: Model Architecture. The conditioning branch receives an image-label pair and produces a set of parameters $\{w, b\}$ for the logistic regression layer $c(\cdot, w, b)$. The segmentation branch is an FCN that receives a query image as input and outputs strided features of conv-fc7. The predicted mask is generated by classifying the pixel-level features through $c(\cdot, w, b)$, which is then upsampled to the original size.

Where $\hat{M}^{mn}$ is the pixel value of the mask at position (m,n) and σ is an activation function. This gives us a matrix of the size of the input query image, with each pixel having a value between 0 and 1. The final mask will be generated from this matrix by using a threshold of $t = 0.5$, which means each pixel having a greater value than t will be part of the final predicted mask.

A more visual explanation can be found in figure 15, which resumes what we've just said.

During this study, we made this model classify an additional class, the Warnbake, whose correct segmentation was at the core of what we were trying to achieve.

3.1 What can be found in the Shaban & Al. directory?

3.1.1 Hand-written utilities

We often needed during this project to automatize some painful tasks, such as creating VOC2012-compliant annotations or generating ground truth masks for the Warnbake that the Shaban & Al. model could process.

The original source code had already a VOC2012 processing unit, that is to say was able to read and be trained on VOC2012 images, so we decided to reuse these existing capabilities to plug (theoretically effortlessly) an additional class: the Warnbake. C.f. Fig. 16.

1. **Resize pictures to a correct size:** When working on image segmentation, people often first reduce the size of the images they are working on, since processing big images is often a lot more computation intensive. We used the *imagemagick* linux package to process images in batches, with the following command :

   ```
   for file in *.jpg; do convert $file -resize 500 $file; done
   ```

   This resizes each image with a fixed width of 500, and preserves image ratio. Great!

2. Let's talk a bit about the VOC2012 dataset. The model knows how to load test and training sets by reading text files called "train.txt" and "val.txt", located in `/path/to/voc/ImageSets/Segmentation/train.txt`. Once this is done, the images can be found in the subfolder `JPEGImages` and masks in both `SegmentationClass` and `SegmentationObject`. The model can determine the class of objects contained in each mask by reading the 8-bits pixel values, which each map to a class. For example, a value of 0 indicates background, whereas a value of 1 indicates an airplane, 21 a Warnbake....
   But how could one possibly generate those masks, which is a very time-consuming task?

Figure 16: An example of the Warnbake images we worked with. Those were provided by the Fraunhofer Fokus' autonomous car, during a test drive.

3. We decided to use the online label tool LabelBox **??**, which is a great utility to label images for segmentation tasks. We just had to import our re-sized images, label them using polygons, and the website would generate xml annotations containing information about the location of the Warnbake. C.f. Fig 17

4. Now that we have annotations ready, we need to generate masks from them. We decided to implement a small python script to parse all annotations at once, generate masks from them and output VOC2012-ready images.
   All the source code can be found under `LABELLATOR/`, which comes alongside with a virtual environment for ease-of-use. Just specify in `main.py` where your annotations are stored, then run `python main.py` and you'll be provided with VOC2012-ready masks. A small additionnal feature that we added is the automatic generation of train.txt and val.txt files, whose content should be added to the eponym files in the VOC2012 repository for the model to be aware of those new masks.

3.1.2 Shaban & Al. Source Code - `shaban_al/`

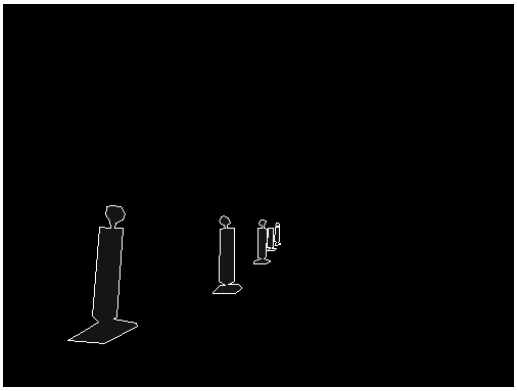
Even if the researchers behind this few-shots model did wrote a great paper, the same cannot be said about the documentation in the affiliated repository [7]. We took time to delve into their code, and add some useful comments to help understand its inner workings. All useful source files are located in the `OSLSM/` subfolder.

1. `code/` holds files specifying:
   - Paths to the datasets (VOC2012, COCO,...) (`db_paths.py`)
   - Utilities to extract images and annotations from the datasets (`ss_datalayer.py`)
   - Miscanellous utilities (`util.py`)
   - How classes splits should be formed and how many shots are we willing to do during testing (`ss_settings.py`)

```
<object>
    <name>Warnbake</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <polygon>
        <x1>376.0</x1>
            <y1>242.0</y1>
        <x2>353.0</x2>
            <y2>245.0</y2>
        <x3>342.0</x3>
            <y3>237.0</y3>
        <x4>345.0</x4>
            <y4>231.0</y4>
        <x5>351.0</x5>
            <y5>231.0</y5>
        <x6>355.0</x6>
            <y6>178.0</y6>
        <x7>369.0</x7>
```

Figure 17: An example of the annotations generated by LabelBox [8]. One can see the Warnbake information represented by polygons.

(a) Base image



(b) Generated Mask

Figure 18: Automatic Mask Generation based on xml Annotations

2. `os_semantic_segmentation/` holds files specifying:
   - Various results that we obtained during the project (`results/`)
   - The test script used to obtain the IoU results (`deploy_1_shot.protoxt`). Use:

     ```
     CUDA_VISIBLE_DEVICES=2 python test.py deploy_1shot.prototxt
     ./training/snapshots/pretrained/fold3/ss_os_fc_iter_60000.caffemodel
     res_1000it_pretrainedOnFold3outof4_foldnr6outof7 1000 fold6_1shot_test
     ```

     to get IoU results on 1000 iterations on the last fold (that is, classes 18 to 21) when classes are split amongst 7 folds of 3 elements.
   - Scripts used to train the model on VOC2012-like datasets (script `train_ss_os_fc.sh`)

3.1.3 voc2012/

This folder contains the VOC2012 dataset, ready to be used by the model. One can add extra classes by following the procedure covered in the 3.1.1 Section!

3.1.4 snapshots_shaban/

1. Be aware that a branch of the neural network (FCN-32s) is extracted from a pre-trained existing network, thus should be initialized with the corresponding weights. In this folder, the authors of this model provide us with the base weights (`os_pretrained.caffemodel`) in order to initialize the FCN-32s branch with its correct weights.
2. Then, the authors added the final weights of the whole model trained on different splits of the data, so that we can check the IoU results claimed in their paper.
   For example, `fold0/ss_os_fc_iter_60000.caffemodel` contains the weights of the model after 60000 iterations trained on classes 6 to 20 (all of them except the first split)

3.1.5 ellipse_generator/

This folder contains the code necessary to generate VOC2012-compliant ellipses (images, masks, train.txt and val.txt)

3.1.6 LABELLATOR/

This folder contains our handwritten script used to generate segmentation masks from xml annotations. C.f. 4 for more details about how using this utility!

3.1.7 DeepMask/

DeepMask is a lass-agnostic segmentation tool, that we first used to generate segmentation masks on the baken dataset. We wanted to check whether it was able to generate correct segmentation on instances of bakens, in order to reduce the effort needed to make VOC2012-compliant masks for the training of our model.
Unfortunately, this performed poorly and was quickly discarded!

3.2 Installation instructions

3.2.1 Base installation

This source code comes from the official repository of [6], which can be found here.They implemented the model in caffe [**?**], an open-source machine learning framework specialized in neural networks, being able to exploit the computing power of GPUs clusters.

- **Warning: the model takes more than 6GB of dedicated GPU memory, so make sure you have access to a powerful board!**.
- A working version of the model can be found at **alexandrerozier@braxton.fokus.fraunhofer.de:~/sebastiantries-cuda-working/OSLSM**.

- Instructions about installing the model can be found here [7].

3.2.2 Braxton-centric Installation quirks

We built from source using the Make program, which was the advised way to go. The process with CMake is kinda broken and should be avoided! Some useful points when installing on the Fraunhofer Fokus' GPUs cluster:

- We need CUDNN capabilities (USE_CUDNN := 1)
- OpenCV version: < 3
- Python version: 2.7
- Cuda version: 8.0 (not higher!) (CUDA_DIR := /usr/local/cuda-8.0)
- BLAS library: atlas (BLAS := atlas)
- Make sure to compile with python support (WITH_PYTHON_LAYER := 1)
- Finally, also watch carefully the PYTHON_INCLUDE, INCLUDE_DIRS and LIBRARY_DIRS, which are correctly set up on braxton but gave us huge headaches.

Re-using the Makefile.config present in the current installation should prevent most of the hurdles, the only difficulty being installing all the correct versions of dependencies.

3.3 Results

3.3.1 Intersection over Union results on the Warnbake class

- Results are presented in the paper [6] (see Fig. 1), where one can compare the efficiency of the model in terms of Intersection over Union (IoU) averaged on all the test classes. The 20 classes of the VOC2012 dataset being split in 4 sets of 5 classes, each column corresponds to the result of a model tested on the 5 provided classes, and trained on the 15 others. For example, `PASCAL-`5^0 corresponds to a model tested on `[aeroplane, bicycle, bird, boat, bottle]`.
- One can see that **the average IoU on 1-shot is 40.8**, so we should use this value as baseline to compare the performance of our model updated to handle the additional Warnbake class.
- Knowing that we now have 21 classes (including the additional Warnbake class), we split the data into 7 splits of 3 categories each. The last split is the one that'll focus on, since it contains the classes `[train, tv, Warnbake]`. We test the model provided by Shaban & Al. trained on the former `[potted plant, sheep, sofa, train, tv]` split, to give us the IoU results for `[train, tv, Warnbake]`.
  Note: we should obtain unchanged IoU results on the train & tv classes, since this model wasn't retrained at all. The results are on Fig. 19
- One can see that the model performs quite poorly on the Warnbake class, which was quite disappointing. However, those poor results were already seen on for example the tv class, were IoU hardly reaches the 8%. As a consequence, we decided to dive further in the model and try to explain why those results were so poor.

3.3.2 Deeper model analysis with Ellipses

To refine our analysis of the model's behavior, we decided to generate programmatically a new class, composed of ellipses of different sizes and shapes. We thought that this simple form, having always the same color, should yield far better IoU results.

The *imggen.py* script generates size 500x300px images of ellipses between 40 and 200 pixels width and heights (randomly selected). It then places the ellipse randomly where it can fit. It also generates along with this image the mask with a 3px wide border. This script is used to generate 200 such images of 100 ellipses.

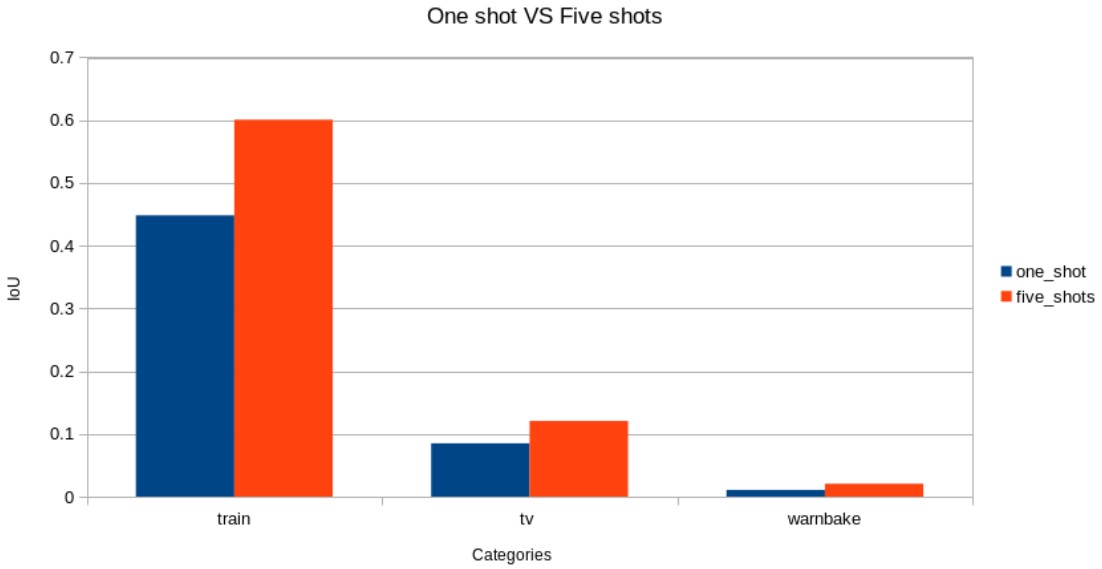


Figure 19: IoU results on the last split (3 classes), with a model trained on the 18 first classes

$i=0$	$i=1$	$i=2$	$i=3$
aeroplane, bicycle, bird, boat, bottle	bus, car, cat, chair, cow	diningtable, dog, horse, motorbike, person	potted plant, sheep, sofa, train, tv/monitor

Methods (1-shot)	PASCAL-5^0	PASCAL-5^1	PASCAL-5^2	PASCAL-5^3	Mean
1-NN	25.3	44.9	**41.7**	18.4	32.6
LogReg	26.9	42.9	37.1	18.4	31.4
Finetuning	24.9	38.8	36.5	30.1	32.6
Siamese	28.1	39.9	31.8	25.8	31.4
Ours	**33.6**	**55.3**	40.9	**33.5**	**40.8**

Methods (5-shot)	PASCAL-5^0	PASCAL-5^1	PASCAL-5^2	PASCAL-5^3	Mean
Co-segmentation	25.1	28.9	27.7	26.3	27.1
1-NN	34.5	53.0	**46.9**	25.6	40.0
LogReg	**35.9**	51.6	44.5	25.6	39.3
Ours	**35.9**	**58.1**	42.7	**39.1**	**43.9**

Table 1: Mean IoU results on PASCAL-5^i. **Top:** test classes for each fold of PASCAL-5^i. The **middle** and **bottom** tables contain the semantic segmentation meanIoU on all folds for the 1-shot and 5-shot tasks respectively.

1. After testing the model on the new Ellipse class, we obtained better IoU results ($\tilde{4}\%$), but still not enough compared to the other classes! We then tried to find some explanations.

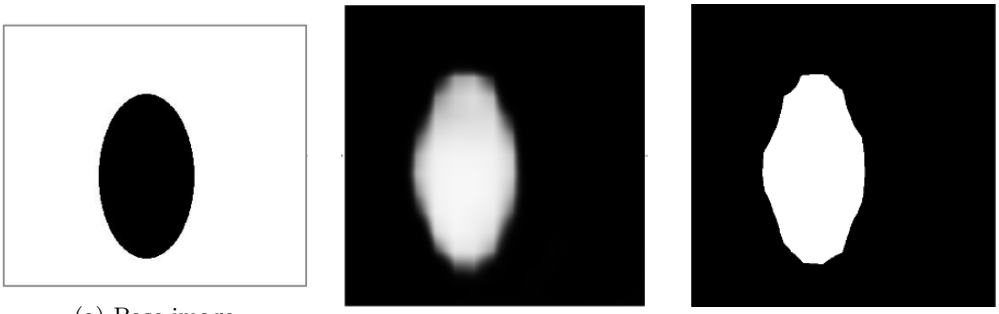
2. One of the most intuitive thought we had was that those example are horribly simple, and lack the noise and colors one would find in a real picture. We formulated the hypothesis that the model wasn't trained enough on "flat" surfaces, since it never had the example in the training dataset!

3. But this wasn't satisfying enough, so we decided to take a visual look at the generated masks, and compare them to the inputs. In Fig. 21 we can see the output of our model, which is very close to the ground truth. However, the threshold seems to be too high as the hard mask looks a little bit too cropped on areas of uncertainty. Decreasing the threshold from 0.5 to a lower value should yield far better IoU results!

4. However, since we were simultaneously developing the Shaban & Al. model alongside with the SOTA Convolutional Neural Net, **we abandoned further analysis since the latter yielded far better results on this image segmentation task.**

(a) Base image (b) Generated Mask

Figure 20: Generating a new test class: Ellipses



(a) Base image (b) Generated "Soft" Mask (c) Generated "Hard" Mask

Figure 21: Soft & Hard mask generation. Hard mask is obtained by keeping all pixels from the soft mask bigger the a defined threshold (here 0.5)

3.4 Discussion

- The Shaban & Al. model relies on a huge, complicated neural network to perform few-shots image segmentation.
- Its training takes at least 10 hours on a Titan X GPU, mostly due to the size of the model and the processed images.
- It performed quite poorly on the two additional class that we added
- But has the incredible ability to segment any arbitrary class given we have a decent support set, and this **without retraining**!

4 Discussion about the two alternatives

The few-shot approach was from the start about the trade-off between performance and size of dataset. The low dataset size required for few-shot inference came at a heavy performance cost.

It was increasingly obvious that small datasets can yield very strong performance through state-of-the-art object detection conventional means, which gradually voided our need for more specific few-shot NN architectures.

Our general conclusion is that any object detection of objects close to any of the 80 classes of the MS COCO dataset is so performant through the state-of-the-art training that choosing another architecture is an unnecessary penalty.

Such a few-shot approach is only valid when a large number of classes need to be detected, which increases the size of the dataset for the SOTA approach above a reasonable few-shot-like size. There, the few-shot neural net will be able to infer on any number of new classes without retraining. However, this was not the goal of the project.

Since the project is axed towards driverless steering, performance is synonymous with passenger safety. The case for strong performance is therefore not only easy but probably necessary, legally speaking.

References

[1] Wikipedia contributors. (2018, July 21). Computer. In Wikipedia, The Free Encyclopedia. Retrieved 15:43, July 30, 2018, from https://en.wikipedia.org/w/index.php?title=Computer&oldid=851264210

[2] Wikipedia contributors. (2018, July 30). Artificial intelligence. In Wikipedia, The Free Encyclopedia. Retrieved 16:22, July 30, 2018, from https://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=852581036

[3] OpenAI. (2017, August 11). Dota 2. Retrieved from https://blog.openai.com/dota-2/

[4] Wikipedia contributors. (2018, July 30). Machine learning. In Wikipedia, The Free Encyclopedia. Retrieved 11:01, July 31, 2018, from https://en.wikipedia.org/w/index.php?title=Machine_learning&oldid=852693809

[5] Faster R-CNN: Down the rabbit hole of modern object detection. (2018, January 18). Javier Rey. Retrieved from https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/

[6] One-Shot Learning for Semantic Segmentation. (2017, September 11). Amirezza Shaban, Shray Bansal, Zhen Liu, Irfan Essa, Byron Boots. Retrieved from https://arxiv.org/pdf/1709.03410.pdf

[7] One-Shot Learning for Semantic Segmentation. (2017, September 11). Retrieved from https://github.com/lzzcd001/OSLSM

[8] LabelBox Image-annotation tool. Official url: https://www.labelbox.com/

[9] Tensorflow Object Detection API. (2018, July 25). Retrieved from https://github.com/tensorflow/models/tree/master/research/object_detection

[10] Tensorflow Models. (2018, August 1). Retrieved from https://github.com/tensorflow/models

[11] Speed/accuracy trade-offs for modern convolutional object detectors. (2017, April 25). Huang J, Rathod V, Sun C, Zhu M, Korattikara A, Fathi A, Fischer I, Wojna Z, Song Y, Guadarrama S, Murphy K, CVPR 2017

[12] Wikipedia contributors. (2018, August 1). TensorFlow. In Wikipedia, The Free Encyclopedia. Retrieved 11:24, August 1, 2018, from https://en.wikipedia.org/w/index.php?title=TensorFlow&oldid=852891608

[13] Wikipedia contributors. (2018, July 13). Google Brain. In Wikipedia, The Free Encyclopedia. Retrieved 11:27, August 1, 2018, from https://en.wikipedia.org/w/index.php?title=Google_Brain&oldid=850094954

[14] Large Scale Distributed Deep Networks. (2012). Jeffrey Dean and Greg S. Corrado and Rajat Monga and Kai Chen and Matthieu Devin and Quoc V. Le and Mark Z. Mao and Marc'Aurelio Ranzato and Andrew Senior and Paul Tucker and Ke Yang and Andrew Y. Ng.

[15] TensorFlow: Large-scale machine learning on heterogeneous systems. (2015, November 9). Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya

Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Retrieved from https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45166.pdf

[16] Speed/accuracy trade-offs for modern convolutional object detectors. (2017, April 15). Jonathan Huang, Vivek Rathod , Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama and Kevin Murphy.

[17] API Documentation. (2018, May 25). TensorFlow. Retrieved 18:27, August 2, 2018, from https://www.tensorflow.org/api_docs/

[18] Install TensorFlow on Ubuntu. (2018, July 19). Retrieved 11:54, August 3, 2018, from https://www.tensorflow.org/install/install_linux#InstallingVirtualenv