Real-time object detection and identification

Using machine learning, training a model from a pre-trained checkpoint

Benjamin Danielsson
2018-01-01
## Contents

1 Introduction .......................................................................................................................... 1

2 Background .......................................................................................................................... 1

2.1 Reason and inspiration ..................................................................................................... 1

2.2 GPU vs CPU ..................................................................................................................... 1

3 Method ................................................................................................................................. 2

3.1 Prerequisite Dependencies and Hardware ....................................................................... 2

3.2 Training model ................................................................................................................ 2

3.2.1 Dataset and conversion ............................................................................................ 3

3.2.2 Pipeline .................................................................................................................... 3

3.2.3 Rocky training montage ......................................................................................... 4

3.3 Real-time detection ......................................................................................................... 5

3.3.1 Code ....................................................................................................................... 5

3.3.2 Object detection function ....................................................................................... 7

4 result ................................................................................................................................... 8

4.1 Training .......................................................................................................................... 8

4.2 Real-time program and code ......................................................................................... 9

5 Discussion .......................................................................................................................... 9

5.1 Optimization ................................................................................................................ 10

6 Bibliography ....................................................................................................................... 11

7 Appendix ........................................................................................................................... 12
1 INTRODUCTION

Earlier in the year of 2017 Google released an API for object detection, through their TensorFlow program. This project utilizes that API to create a real-time object detection and identification program with the use of the built-in webcam from a Mac laptop. First, a model is trained on new data from a pre-trained model’s checkpoint. That model is then used in the program and makes inference on each frame that is received from the webcam. Said frame then has an overlay drawn on top and displayed to the user. The training is done on a GPU with the use of TensorFlow’s version that is specialized for GPU usage. The training is run on Windows with a GeForce GTX 980 Ti.

2 BACKGROUND

2.1 Reason and inspiration
This project was created in order to help get a better understanding for deep machine-learning and get actual practical work with it. Learning from textbooks is fundamental, but to work with the methods in a practical way helps the understanding. However, the inspiration for the project came from not only wanting to get a better understanding but to also get a working program that has some purpose. Creating a model is one thing, but to also write a program that uses said model for inference is another. This project allows me to do both, and with the help from the official documentations, tutorials and an article written by Dat Tran on how to create a real-time application the project was made possible.

2.2 GPU vs CPU
TensorFlow has two primary forms of its software, a version that utilizes the central processing unit (CPU) and one that utilizes the graphical processing unit (GPU). The best version to use depends on the hardware the computer that will do the calculations has access to, depending on which hardware is best the corresponding version should be chosen. However, it is generally said that the GPU is better for machine learning; benchmarks and comparison shows this trend as well (Shi, Wang, Xu, & Chu, 2016).
To understand the reason for this a general comparison between CPUs and GPUs is needed. A CPU can handle many fast calculations, such calculations are the most common on a computer, this stands in contrast to the purpose of a GPU which is instead optimised for complex but slower calculations. Games use complex calculations to render graphics and
calculations for physics, which is why the GPU was created in the first place. These complex calculations are also utilised by deep learning which is why GPUs are favoured for its use. A simple metaphor to better understand this is to think of the CPU as a sports car, and a GPU as a lorry. When simple and easy transportations is wanted a sports car will be favoured since it is much faster than a lorry, but for more complex transportations the lorry is favoured because it can carry much more (Shaikh, 2017).

3 METHOD

3.1 Prerequisite Dependencies and Hardware
This project took use of several software libraries, packages and programs to utilize machine learning. Python was the choice of programing language, and TensorFlow was used for the deep learning computations, which in turn has a list of dependencies. TensorFlow offers a version for CPU usage and another for GPU, this project used the GPU version. Said version requires extra programs from the GPU designer NVIDIA, such as CUDA Toolkit, cuDNN and their GPU drivers.

So far NVIDIA is the leading GPU designer for deep learning (also crypto mining and other similar high complex tasks) since they also write programs that are compatible with their cards that enable much of this capacity. The card used for this project was a GeForce GTX 990 TI.

3.2 Training model
To explain the methods used to complete this project it will be divided in to smaller sections. First and foremost, the previously mentioned programs and packages had to be installed, then a model had to be procured and trained. To do this training a dataset in the form of pictures was gathered and subsequently translated to the correct format. When the model was completed its inference graph could be applied to new data to detect and identify said data. This data came in the form of camera input from a laptop, which created a real-time input system where anything in front of the camera was put through the model.

Training a model from scratch can take a very long time, especially when the hardware used is far from optimal. The training for this project was done on a GTX 980 TI, so to save on time a checkpoint from a pre-trained model was used. This model can then be further trained on your own dataset. This will result in a good model with your dataset without taking as much time as training everything yourself.
The model I choose for this project was optimised for speed and not accuracy, this was done because an accurate model will be larger with more data points which means that when new data is used to identify objects each loop will take longer. For the real-time portion of the code to work speed is very important, since it will have to iterate several times a second (see optimization).

### 3.2.1 DATASET AND CONVERSION

The article written by Dat Tran provided his set of pictures for free use, it was a total of 200 jpg and png files with raccoons. However, the model cannot be trained on these file formats but instead have to be converted into the TFRecord file format (using_your_own_dataset, 2017). Said format is good to use when you need to stream large portions of data (Data IO (Python Functions), 2017), it is binary files with sequential content which means it is predictable (Oracle, 2017). In the documentation provided by Google there are instructions how to write a script for this conversion (using_your_own_dataset, 2017). The script is not fully automatic but requires some manual input from the user for each picture. The picture’s height, width, filename and file type has to be added, this can be done automatically through python libraries such as PIL. A bounding box then has to be created around the object that is depicted on the picture, this requires two X coordinates and two Y coordinates. In the dataset used for this project the box would contain the raccoon and everything outside the box is background, such as greenery. The article from Dat Tran suggested using a program called LabelImg which allows the user to draw boxes around a picture and export the coordinates to a XML format. This information could then be imported to the script. The last variables the user has to add is the label for what is depicted, e.g. “Raccoon” and the integer for that label in the label map, e.g. 1. If the dataset contains several different animals or object this could also take some time but since only raccoons is used this could be hard-coded. Once all these variables are collected it is as simple as using a TensorFlow method to create the record (using_your_own_dataset, 2017).

### 3.2.2 PIPELINE

With the dataset in hand the final thing to configure is a training pipeline, which is a textfile with config as file extension. The documentation provides instructions on how to create this config file. At a high-level the file is divided into 5 sub-categories, model, train_config, eval_config, train_input_config, eval_input_config (Configuring_jobs, 2017). These parts can be heavily modified to fit your project to achieve optimal performance. The documentation
provided from Google also give several sample files that can be used as a rough basis which is then slightly modified to get a pipeline that while not perfect will be adequate for this project (Configuring_jobs, 2017). I used a sample config that was heavy on speed rather than accuracy, and that was created for pet recognition as that fit the goal of the project the most. Some variables were changed, such as how many labels and classes was to be used.

3.2.3 ROCKY TRAINING MONTAGE

*TensorFlow* training can be done three different ways, locally on either a CPU or a GPU or on Googles Cloud servers. For this project, the training was done locally on the graphics processor. With the dataset complete, a pipeline provided and a labelmap created the training can be started. It is started through a terminal with the command:

```
python object_detection/train.py
   --logtostderr
   --pipeline_config_path={PATH}
   --train_dir={PATH}
```

`--logtostderr` is a flag to activate logging, which is useful to debug errors. The other flags are paths to the different directories used, path to the pipeline config file, and path to where training checkpoints and events will be saved (Running Locally, 2017).

This step seems quick but will most likely require debugging since this is the step where most code is run for the first time. When it starts training properly *TensorFlow* allows the user to run the evaluation job simultaneously. This is highly recommended (Tran, 2017) because the training will run indefinitely until the user stops it. Stop the training to early and the model will be underfit which means it will generalize too much, train too long and the model will be overfit which means the model is too specific. Both of these will result in inaccurate results, underfitting is easy to avoid with proper evaluation techniques while overfitting is significantly more difficult (Tran, 2017).

The evaluation is done through *TensorBoard* which provides graphs over the training and its performance. By looking at the graph over *Total Loss* and *Mean Average Precision* (see *Results* for graphs and further details) a decision was made when to stop training.

When the training was completed the model was exported to a protobuf file to be used in the programing for object detection.
3.3 Real-time detection
With the training completed it is time to use the frozen checkpoint to run testing on a live feed from a camera. The article from Dat Tran had a working version I took help from, however a much simpler version was created since he was using multithreading and similar techniques that was out of the scope for this project.

3.3.1 CODE
The code can be dissected in to smaller parts to easily understand it (see appendix for full code with inline comments). First comes the different libraries and packages that are needed for the code.

```python
1. import time
2. import os
3. import cv2
4. import numpy as np
5. import tensorflow as tf
6. from object_detection.utils import label_map_util
7. from object_detection.utils import visualization_utils as vis_util
```

The next part are variables that will be used in the document, such as filenames and paths for the model and label map. The reason for having paths and names separate like this is because it makes it possible to easily switch models by simply changing the filenames at the start of the file.

```python
8. path_cwd = os.getcwd()
9.
10. name_model = 'ssd_mobilenet_v1_coco_11_06_2017'
11. path_ckpt = os.path.join(path_cwd, 'object_detection', name_model, 'frozen_inference_graph.pb')
12.
13. path_labels = os.path.join(path_cwd, 'object_detection', 'data', 'mscoco_label_map.pbtxt')
14.
15. amount_classes = 90
16.
17. label_mapp = label_map_util.load_labelmap(path_labels)
18. categories = label_map_util.convert_label_map_to_categories(label_mapp, max_num_classes=amount_classes, use_display_name=True)
19. category_index = label_map_util.create_category_index(categories)
```

With the basics covered, next comes the main part of the code that is run. First the model is loaded in to the memory, and a TensorFlow session initiated.

```python
20. graph_for_detection = tf.Graph()
21. with graph_for_detection.as_default():
22.     graph_def = tf.GraphDef()
```
With the use of the library called CV2 a video feed can be opened, with the window size of 480 by 360 pixels (see optimization for reasons for using a small screen). A package with FPS tracking is also started.

The loop that makes the program work continuously and in real-time is written next. A method called read for the livefeed that was created on line 29-31 is called. It returns a Boolean, and the current frame from the camera. The Boolean is True or False depending on weather the VideoCapture is active or not, in this instance it should always be True.

Row 37 calls a method called imshow which shows the image provided to it. For this project that image is what is returned form the detect_object function which returns a frame with an overlay of what was detected on it (see end of chapter for further details on this function).

Row 39 calls the FPS package and tells it that another frame has been created as well as the elapsed time in order to keep track of the current FPS.

Lastly, a stop function is created. If the user presses “q” on their keyboard when to focus is on the video screen the loop finishes.

When the loop is ended the project will cancel with the appropriate methods. The FPS package is stopped and then prints how much FPS the program had with readable formatting.
Then video feed is removed from memory and stops being active, and then the window is removed.

Lastly the TensorFlow session is closed and with that the program ends.

```python
43. fps.stop()
44. print('[INFO] elapsed time: {:.2f}'.format(fps.elapsed()))
45. print('[INFO] approx. FPS: {:.2f}'.format(fps.fps()))
46.
47. videoCapture.release()
48. cv2.destroyAllWindows()
49. sess.close()
```

### 3.3.2 OBJECT DETECTION FUNCTION

On row 37 a function called `detect_objects` was called; with it a frame, a TensorFlow session and the graph is sent. TensorFlow expects all frames to be of the same size so firstly the frame is expanded to be of a certain proportion.

Afterwards 5 tensors are called from the graph with the TensorFlow package (tf.Tensor, 2017). This is the information that will be used for the detection.

```python
1. def detect_objects(frame, sess, graph_for_detection):
2.     expanded_frame = np.expand_dims(frame, axis=0)
3.
4.     image_tensor = graph_for_detection.get_tensor_by_name('image_tensor:0')
5.     boxes = graph_for_detection.get_tensor_by_name('detection_boxes:0')
6.     scores = graph_for_detection.get_tensor_by_name('detection_scores:0')
7.     classes = graph_for_detection.get_tensor_by_name('detection_classes:0')
8.     num_detections = graph_for_detection.get_tensor_by_name('num_detections:0')
```

Next the object detection API is used to do the object detection and identification, the variables that was created on row 4-8 is used, and will return updated information.

```python
9.     (boxes, scores, classes, num_detections) = sess.run(
10.        [boxes, scores, classes, num_detections],
11.        feed_dict={image_tensor: expanded_frame})
```

With the help of numpy and another TensorFlow utility that creates an overlay on top of the original frame information that was gathered through TensorFlow is shown. It draws a box around the object that was found, writes what type of object it is and the confidence that said identification is correct. There are also some additional settings like the thickness of the border and format of the coordinates. The function does not provide an easy way to change fonts so I opened the file where the `visualize_boxes_and_labels_on_image_array` method was written and manually changed the font with some hardcode. Lastly the newly created frame is returned which is used in the main loop of the code, and subsequently shown to the user.
4 RESULT

4.1 Training
Figure 1 shows the total loss over time (steps) and is one of several graphs that can be seen in TensorBoard, figure 2 shows the mean average precision over time (steps). It is with the help of graphs like these one can decide when the training is sufficient and should be stopped. The TotalLoss stabilizes quickly because of the use of a pre-trained model. The Mean Average Precision is not as steep since it is tested on the 40 raccoon pictures, which is data the pre-trained model has no experience with.

![Figure 1: Training evaluation shown in total loss by steps, from Dat Tran](image1)

![Figure 2: Training evaluation shown through mean average precision, from Dat Tran](image2)
4.2 Real-time program and code
The program updates the video window with a new frame every between 0.25 sec and 0.5 seconds, which means an average of 2 - 4 FPS. The object identification for raccoons is not great, but sufficient. The program can be run with other models. When run against a model with 90 labels and categories it provides the same speed and FPS as the model with raccoons. The other model I tested with was a sample model from TensorFlow and provided better object detection and identification in the sense that it could detect 90 categories instead of only raccoons. Figure 3 shows a screenshot from the program as it is running on that model. It identifies me as a person with 95% confidence, and the flask with water as a bottle, also with 95% confidence.

![Figure 3: A screenshot from the program while it is running](image)

5 DISCUSSION
The program runs well enough for me to call the project successful, the FPS is in my opinion quite low and is the biggest bottleneck. However, for a video it is sufficient to see movements and whatever is placed in front of the camera. Although the model was trained on a windows machine with mid to high end hardware the program is run on a Mac with significantly

---

1 In terms of personal computers, significantly better hardware is traditionally used for machine learning. Googles Cloud servers that can be used for your own machine learning is substantially better.
worse hardware performance. The reason for this is that the Mac has a built-in webcam while the Windows computer does not.

5.1 Optimization
The main loop for the program is called as quickly as possible, it calls the object detection code and then displays a frame to the user. The faster that loop is the higher the videos FPS will be, and the more stable the video will appear. It was due to this limitation I had to use a small window for capturing the frames. I used 480 by 360 as frame size which is small for modern computers, but higher than that and the performance suffered and smaller and it became increasingly difficult to actually see what the frames depicted. There are ways this could be improved, however, I considered them out of the scope of this project. The article from Dat Tran writes that threading of multiprocessing could be used. This would fundamentally change the code to run the loop several times parallel to itself, this would have to be done with queuing so that correct frames are taken and displayed in correct order. Another way to improve performance is to change the hardware. As stated earlier the program was run on an older Mac laptop, this was because it had easy access to a webcam. One way to solve this could be to get an external webcam to the Windows computer, this would allow the program to run on better hardware.

There is however one aspect with using a laptop that is better which is the simple fact that it is mobile, and allows me to use the identification software where ever I want. Instead of bringing objects to the computer I can bring the computer to the objects. It also makes it easier to present the program to other people.
6 BIBLIOGRAPHY


7 APPENDIX

1. import time
2. import os
3. import cv2
4. import numpy as np
5. import tensorflow as tf
6. import sys
7. from utils.app_utils import FPS
8. from object_detection.utils import label_map_util
9. from object_detection.utils import visualization_utils as vis_util
10. path_cwd = os.getcwd()
11. #The path to the frozen graph for detection. It's the model that will be used for the object detection.
12. name_model = 'ssd_mobilenet_v1_coco_11_06_2017'
13. path_ckpt = os.path.join(path_cwd, 'object_detection', name_model, 'frozen_inference_graph.pb')
14. #This is the list of the strings used to add labels for each boundary.
15. path_labels = os.path.join(path_cwd, 'object_detection', 'data', 'mscoco_label_map.pbtxt')
16. amount_classes = 90
17. label_map = label_map_util.load_labelmap(path_labels)
18. categories = label_map_util.convert_label_map_to_categories(label_map, max_num_classes=amount_classes, use_display_name=True)
19. category_index = label_map_util.create_category_index(categories)
20. def detect_objects(frame, sess, graph_for_detection):
21. #TensorFlow expects images to have the same shape
22. expanded_frame = np.expand_dims(frame, axis=0)
23. image_tensor = graph_for_detection.get_tensor_by_name('image_tensor:0')
24. boxes = graph_for_detection.get_tensor_by_name('detection_boxes:0')
25. scores = graph_for_detection.get_tensor_by_name('detection_scores:0')
26. classes = graph_for_detection.get_tensor_by_name('detection_classes:0')
27. num_detections = graph_for_detection.get_tensor_by_name('num_detections:0')
28. #This uses API to do the actual identification
29. boxes, scores, classes, num_detections = sess.run([boxes, scores, classes, num_detections], feed_dict={image_tensor: expanded_frame})
30. #Uses data from API to create overlay with information on frame
31. vis_util.visualize_boxes_and_labels_on_image_array(frame, np.squeeze(boxes), np.squeeze(classes).astype(np.int32), np.squeeze(scores), category_index, use_normalized_coordinates=True, line_thickness=8)
#Original frame, before expansion, with overlay

```
59. #Original frame, before expansion, with overlay
60. return frame
61.
62. if __name__ == '__main__':
63. #Loads a Tensorflow model into the memory.
64. graph_for_detection = tf.Graph()
65. with graph_for_detection.as_default():
66.     graph_def = tf.GraphDef()
67.     with tf.gfile.GFile(path_ckpt, 'rb') as fid:
68.         graph_serialized = fid.read()
69.         graph_def.ParseFromString(graph_serialized)
70.         tf.import_graph_def(graph_def, name='')
71.     sess = tf.Session(graph=graph_for_detection)
72.
73. #CV2 package. Starts VideoCapture
74. videoCapture = cv2.VideoCapture(0)
75. videoCapture.set(cv2.CAP_PROP_FRAME_WIDTH, 480)
76. videoCapture.set(cv2.CAP_PROP_FRAME_HEIGHT, 360)
77.
78. #initialize fps tracking
79. fps = FPS().start()
80.
81. while True:
82.
83.     #retval is bool, frame is "image" which is a matrix
84.     retval, frame = videoCapture.read()
85.
86.     #Calls detection function, displays the Return with CV2 in a window called V
87.     #video.
88.     cv2.imshow('Video', detect_objects(frame, sess, graph_for_detection))
89.
90.     fps.update()
91.
92.     #Stop function
93.     if cv2.waitKey(1) & 0xFF == ord('q'):
94.         break
95.
96. #Stop the clock, then displays the FPS.
97. fps.stop()
98. print('[INFO] elapsed time: {:.2f}'.format(fps.elapsed()))
99. print('[INFO] approx. FPS: {:.2f}'.format(fps.fps()))
100.
101. #After stop function is executed, peacefully turn everything off
102. videoCapture.release()
103. cv2.destroyAllWindows()
104. sess.close()
```