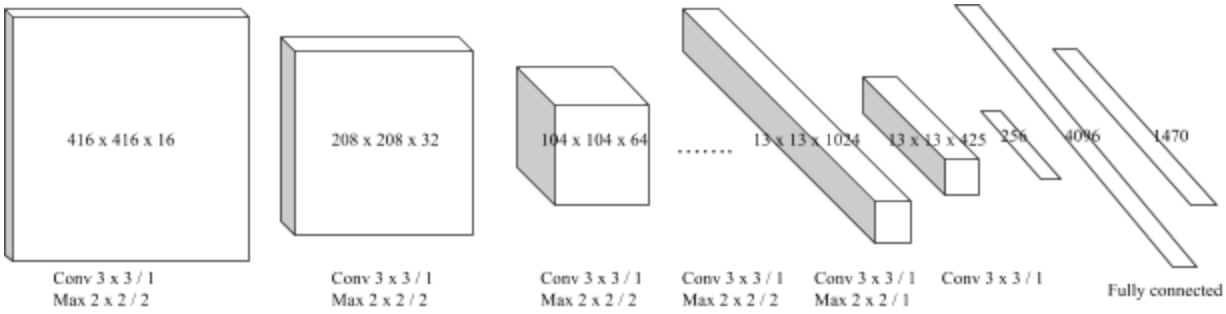
Final Project Report - YOLO on iOS

Deep Learning for Computer Graphics

Yang You(yy7hd), Tianyi Jin(tj2cw), Jingxuan Zhang(jz8bw)

Introduction

You Only Look Once, as known as YOLO, is a state-of-the-art, real-time object detection system. Especially, YOLO focuses on improving the speed and simplicity of detection tasks. Inspired by its imitation of human's instant perception of visual elements, we want to reimplement a YOLO model on devices with limited computing resources, such as mobile phones. We would further like to explore the architecture of YOLO and figure out means to optimize its performances, which may help build practical applications regarding this technique. In this project, we adopted the compact, Tiny YOLO as our initial model as it is much faster than the normal one.

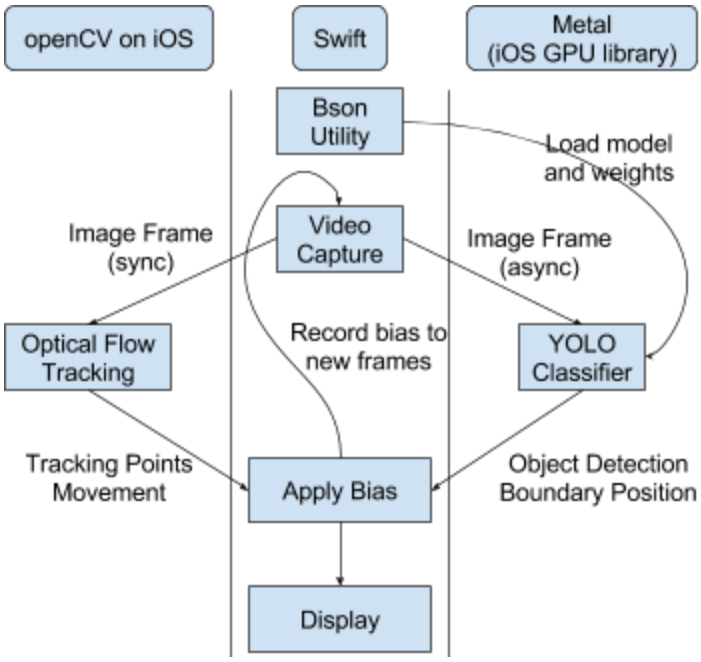


The above figure is the Tiny YOLO architecture we used.

Group members and work division

- Yang You (yy7hd)
  - Fully connected layer
  - Fast convolutional layer
  - Integration
- Tianyi Jin (tj2cw)
  - Darknet model to json model
  - Bson utility (json to bson)
- Jingxuan Zhang (jz8bw)
  - OpenCV optical flow
  - Video capture and integration

Components and Architecture



The above figure shows the general architecture of our project. We partially utilized an iOS based deep learning framework, which is written in swift, together with Metal to manipulate GPU computation resource. We also used OpenCV as an auxiliary method to smooth the detection experience.

However, the original framework was too weak to satisfy our requirements and we added and modified many parts of it to made the neural network work.

At the very first part, we created a new file format called *binary json* to load model configurations and weights. Because the training process is extremely time consuming somehow unrealistic to do so (pretrain using ImageNet and fine train using VOC 2007 + 2012) and the training process itself was somehow tedious, we wanted to use the presented Darknet trained model rather than repeat it from scratch. However, if we converted the original weights into a json format, the storage consumption can be extremely high. Thus, we decided to serialize the weights and architecture information into a binary json file. This saved a big amount of memory on iOS (from 1.5GB to 600MB), and it also reduced the time of loading all the weights from the file at the beginning.

As to the model part, we also added a new type of layer called fully connected layer, which was not included in the original framework. What's more, we replaced the original convolutional layer with a newly written fast convolutional layer, which proved to be about 30% faster than the original layer testing on iPhone. This was realized by excluding redundant data processing procedure like flattening matrix and optimizing the logic flow. In addition, we provided support for more layer configurations such as negative slope. All these improvements were written using Metal shading language.

After adding the above components, we made it possible to run YOLO on an iPhone. Now we can integrate it into our workflow.

By using bson utility, we can load the model efficiently. After that, we captured the video (stream of frames) from iPhone's back camera and get each frame of the video. We dispatched the image to the YOLO classifier and let it to detect the objects and their locations using iPhone's GPU. We could achieve about 2 fps using purely YOLO neural network to detect objects. To get a better performance, we proposed a new method to track the motion between sequential frames as an auxiliary way to increase fps.

As the graph shows, we ran two threads. One was responsible for calculating the optical flow between pairwise sequential frames. The other one was to send frames to GPU, calculating by YOLO. Once the result of GPU had been received and processed, it will send the next request. The objective for the former thread was to keep a fps while tracking the bounding box as accurate as possible. The target for the latter thread, which communicates with GPU, was to correct the bias generated by optical flow as it can not run in real time on iPhone.

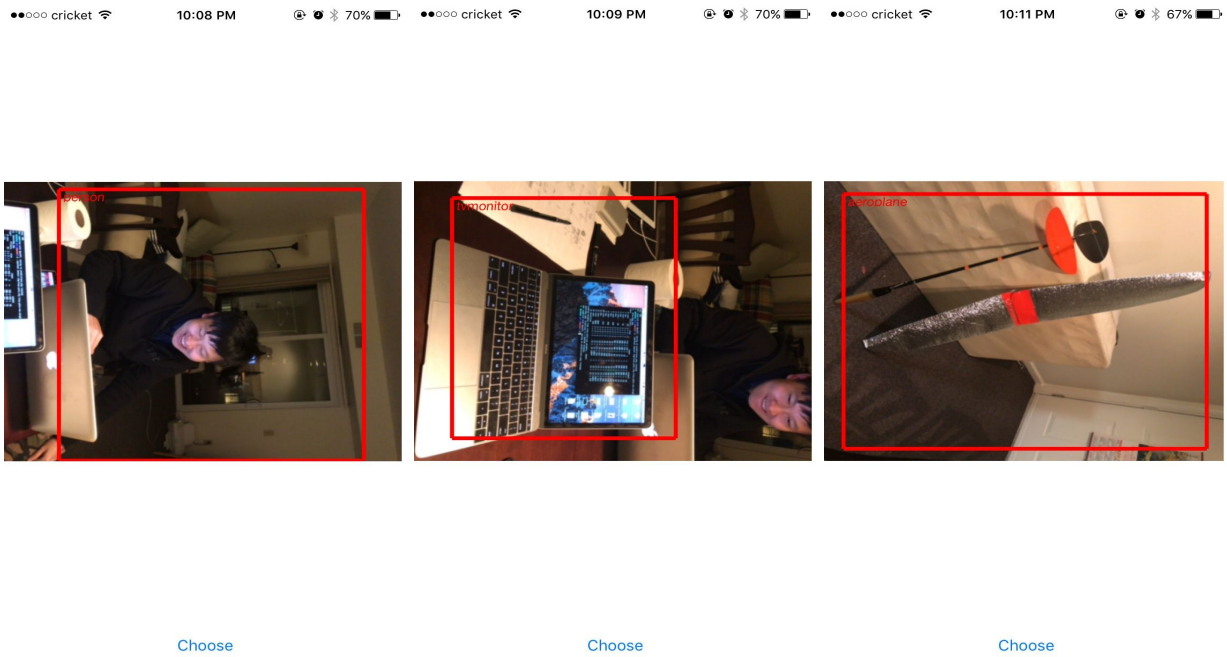
Experiments

We made experiments on both the ordinary PC server and iPhone. We do experiments on ordinary PC server using default configuration the paper provided as a baseline. We compiled the YOLO network using darknet (the one that the author of that paper developed) without CUDA (we cannot compile with GPU on the server since the server lacked some libraries that darknet required). By using the standard YOLO architecture (30 layers) and common CPU, it took 7.41 seconds to detect a single image. While using the Tiny YOLO architecture (15 layers) and common CPU, it took around 1.55 seconds to finish detecting one image.

On the iPhone device, we adopted a deep learning kit for iOS, leveraging the computational power of iPhone GPU (using Metal framework). With the help of that framework, we rebuilt the entire Tiny YOLO architecture, using the trained weights provided by the author of the paper. Without any optimization of the original code, we made it around 0.75 seconds to detect an

image. However, this is far too slow than our expectation. We then optimized the original framework by rewriting the convolutional layer and the low-level Metal interface. After that, we reached the performance at around 0.5 seconds per frame.

Results



The above three photos were screenshots from iPhone using the application we developed. As shown in the pictures, we can use the back camera of iPhone to capture videos and detect object in near real time. The detection is quite accurate and robust.

Conclusion and Future Works

We mixed deep learning network with openCV optical flow method, to achieve a high performance, real-time image classification solution on iOS. We believe this will be extremely useful since similar Apps on market still rely on servers to compute and broadcast results, which enormously increases servers' loads and pressure. Meanwhile, it requires fast net speed to maintain the strong connection stability and real-time effects. When it comes to videos, this is hardly possible. Instead, our work is done entirely on client side and is not limited by different network situations.

In the future, we are going to simplify the Tiny YOLO model further to make it more efficient, utilizing methods such as deep compression. In addition, we'd like to optimize the GPU part codes to get a better performance when running on GPU.