Individual Project: Object Detection with Tensorflow

Ryan Swope

April 20, 2021

1 Introduction

1.1 A Foreword about Tensorflow

Tensorflow is an open source library used primarily for machine learning applications. It is an incredibly powerful tool capable of leveraging GPUs and ASIC, and gives its users access to optimized neural network geometries. As its name implies, Tensorflow utilizes tensors rather than the two dimensional arrays typically associated with neural networks. Unfortunately, as is true with many things, the versatility and power of Tensorflow bring with it a steep learning curve and a general lack of unified tutorials. Following one article about writing an object detector may read very different from another tutorial depending on the implemented version of Tensorflow, the model trained on top of, or the operating system being run on. This lead to several extensive road blocks in my work that I'll detail later. However, despite these shortcomings, Tensorflow provides anyone with experience in computer programming and $\sim$50 hours to spare with perhaps the most accessible and powerful machine learning library available to the public at no cost.

1.2 Goals of this Project

Typically our modeling assignments have focused on one topic or application and then built off that concept or built on it, so I feel its appropriate to lay out what my goals were for this project. During the Neural Networks assignment, I attempted to train an object detection network to detect images of knives. I thought this would be an interesting application of object detection as it could theoretically be used by law enforcement and it wasn't a readily available pretrained model (many networks exists that can detect pets, cars, etc.). I was met with very limited success, and the network didn't really detect the knives. Although there were other applications of Tensorflow I could've explored, I felt unfulfilled by the results of that network, so I wanted to revisit it. My goals for this project were:

1. Create a training set of images of knives

2. Train a network that can successfully detect knives in pictures (and possibly video).

3. Try training again on a GPU to compare training speed and accuracy

4. Time permitting, try and find a larger dataset online and compare a model trained on that to mine.

5. Time permitting, try again in a Colab notebook with TPU acceleration.

6. Demonstrate the detection!

I was able to complete all these except for the TPU training. I'll detail my successes and failures for the others throughout the reports, but I'll address the TPU one now. Essentially, there were several hoops to jump through just to set up the model to be trained by a TPU, and ultimately the possible increase in training speed I would get using the TPU over a GPU as not worth the trouble. Among other things, the training I was doing wasn't constructed to leverage TPUs. TPUs are a type of ASIC which stands for Application-Specific Integrated Circuit. Essentially, these are cards which are developed to do VERY specific mathematical computations with extreme efficiency, and nothing else. Nowadays, Bitcoin miners use ASIC cards that are designed specifically to do work on Bitcoin's hashing algorithm. Similarly, TPUs are a form of ASIC designed to do tensor math, but they need to be "spoon fed" the appropriate tensors. As a results, I couldn't leverage them to their full potential and ultimately a GPU was sufficiently fast for my purposes.

1.3 The RCNN COCO Model

Up to this point I have simply referred to the object detector as a "model". This is true, but there are many object detection algorithms including YOLO (You Only Look Once), GAN (General Adversarial Network), and RCNN (Regional Convolution Neural Network). My network was trained using an RCNN network initially trained on the COCO data set from Microsoft. Convolution neural networks work by apply filters to images to extract features, such as vertical lines, color intensities, or contrast. That simple data is then fed through the network (in which at least layer uses convolution rather than matrix multiplication, hence the name) in an effort to recognize more complex patters. CNNs have the disadvantage of being fully connected (each neuron in one layer is connected to every neuron in the next) and thus are prone to over fitting data. This was apparent in my testing, but I'll get to that later. A regional convolution neural network simply splits an image into regions (usually this is done several times with different sized regions) and looks for patterns in the regions, before the entire image is put together. I specifically used the faster-rcnn model, which is essentially just an rcnn model with some optimized hyperparameters and network topology.

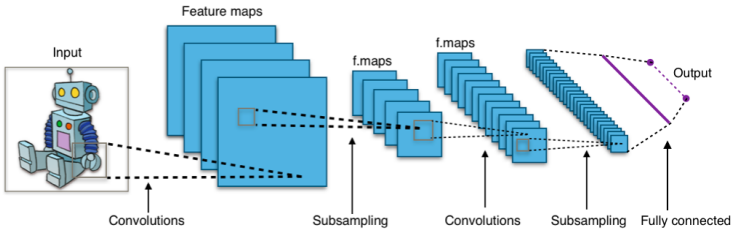

Figure 1: An example of an RCNN network

The COCO data set, or Common Objects in COntext [1], was not created by a gorilla that knows sign language. It is a data set which contains hundreds of thousands of images that have been hand labelled. The images are split into three subclasses: iconic, iconic scenery, and non-iconic. Iconic in this context refers to if there is an obvious and distinguishable subject of the image, or if the image captures a more random and chaotic scene. As the name suggests, the data set labeled images of everyday objects such as animals, vehicles, furniture, and many others. Due to the huge number of images, almost all possible arrangements are encountered: multiple objects, overlapping objects, different lighting conditions, and more. Training a network on this data set will create a model that, in general, is able to differentiate and recognize objects of interest.

2 (Re)Training A Model to Recognize Knives

2.1 Retrain

The question you may ask yourself is, "The COCO model is trained to find lions and tigers and bears, planes, trains, and automobiles, but certainly not knives. So how does it help? Wouldn't you have to start from scratch?" The fact of the matter is, COCO can't detect knives yet. But, rather than starting from scratch, we can simply retrain retrain it on a data set of knives to make it... edgier. And although we are retraining it on a new data set, the idea is the network weights that allowed the network to recognize the aforementioned objects are pretty close to what they'll be after training on knives, so rather than climbing the whole mountain, we kind of get to take a gondola to the top. In a lot of ways this is how children begin to learn objects. It takes a very long time for them to learn what a few objects are, but once the learn how to learn, the process is much faster.

2.2 Data Preparation

To retrain our model, we need to supply it with labeled data. I did this by taking ∼275 pictures of several different knives around my house in a variety

[1] "Microsoft COCO: Common Objects in Context", Lin et al (2015)

of lighting conditions, angles, background, etc. I also included images of me holding the knives at various angles.

Figure 2: The knives I took pictures of.

I combined these with ∼100 pictures from the internet of people holding or fighting with knives. I then went through and manually labeled the knife (knives) in each image with `LabelImg`. `LabelImg` allows you to draw boxes around the knife (knives) in an image and quickly annotate them. It then also automatically exports a .xml that saves the coordinates of the vertices of the bounding box(es). During this point in my work I made the first of many grave errors: I named the class 'knive', which of course is not how you spell knife. I never caught the mistake because after the first image it auto-filled the class in for me. I could *in theory* go back and fairly easily change them in bulk, but I chose not to for two reasons. The first is the model works with the misspelling, and I seriously question whether its worth retraining it just to correctly spell the class. Second of all, it proves that this model that I claim to be mine is mine. Google or other developers online wouldn't keep an error like that in their models. Thus I have proof by error that this model is mine. There were more errors with the name which I will grace the reader with later in the report. After this, our data was essentially ready to be used. The only additional preparation was to compile the training data into a few files that would point to the necessary data.

Figure 3: A image from the internet.

Figure 4: A training image internet.

Figure 5: A training image that I took.

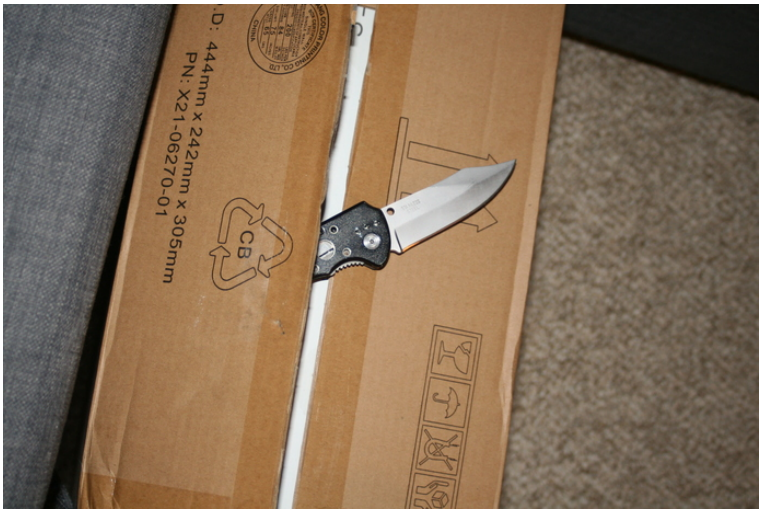

Figure 6: A training image that I took.

2.3 Configuration

At this point, the model was essentially ready to train. All that was left was to prepare a configuration file for the network that would tell it how many classes (types of object) it would be looking for — in this case one, where the training data was, and the hyperparamters with which it would train.

3 Training

3.1 Hardware

The time to train a neural network, especially complex ones such as this, can be astronomical for CPUs. My desktop computer, which has an 8 core processor, could train on the order of 5,000 steps of the fast rcnn model in the span of a day. While it has a GPU, the GPU is manufactured by AMD. AMD does not have a proprietary framework to communicate with Tensorflow, and thus I had to rely on open source software called ROCM. While it looked promising, and there are plenty of people who got it to work, I was unable to get ROCM to work despite hours of dealing with graphics card drivers and kernels. In contrast, my parent's laptop (with its Nvidia GTX 1060 GPU) can run Tensorflow with GPU acceleration thanks to Nvidias CUDA and cuDNN libraries. It was able to hit the soft-coded limit of 200,000 steps within ∼12 hours of training (this limit was set by the people who released the COCO model, as they found there was no appreciable return after this many steps).

3.2 Loss Convergence

When the model is done training, or has effectively reached its accuracy limit, the loss value will have converged to a value. For the fast rcnn model, this convergence value is ∼.05, although after 200,000 steps was closer to .03 in my experience. However, for two and a half weeks — this goes back to the last assignment — I was plauged by exploding loss values. The loss would decrease as expected for ∼500 steps, before exploding to values exceeding 10E21. Obviously, something was wrong, but it took me upwards of two weeks to figure out what it was. The problem was painfully simple. I had labeled my images with the class name 'knive', but in providing the list of possible objects to the model, I called it 'Knive' with a capital K. I cannot stress enough how long it took me to find this bug, but after weeks of searching for it, I was able to fix it and train the network as expected.

4 The Three Models

4.1 Architecture

For the purposes of examination, I decided to look at the results given by three models: the one trained on my desktop with my training set (5523 steps), the one on the GPU with my training set (200,000 steps) and one on the GPU (80,000 steps) with a training set I found online (about 800 training images as compared to about 350 from mine). Despite the difference in steps, the accuracy of the two GPU models (as measured by loss) are equivalent as the loss value had converged in both cases to the value of about .05. All three models used the faster rcnn architecture pretrained on the COCO data set, starting from the same standard checkpoint.

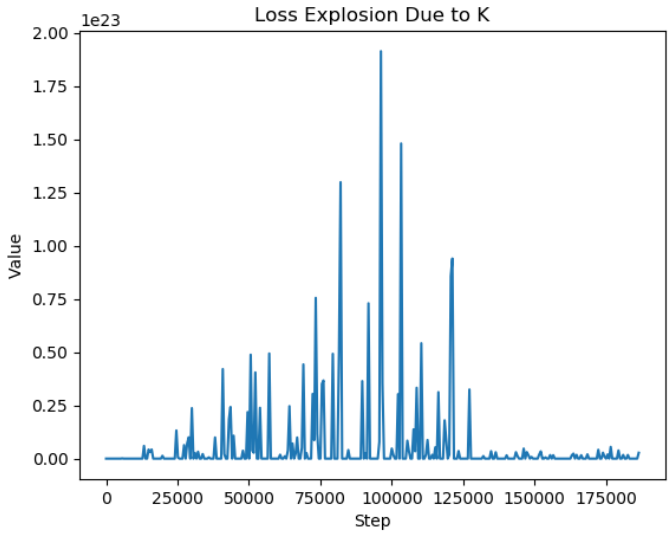

Figure 7: Loss Explosion due to errors.

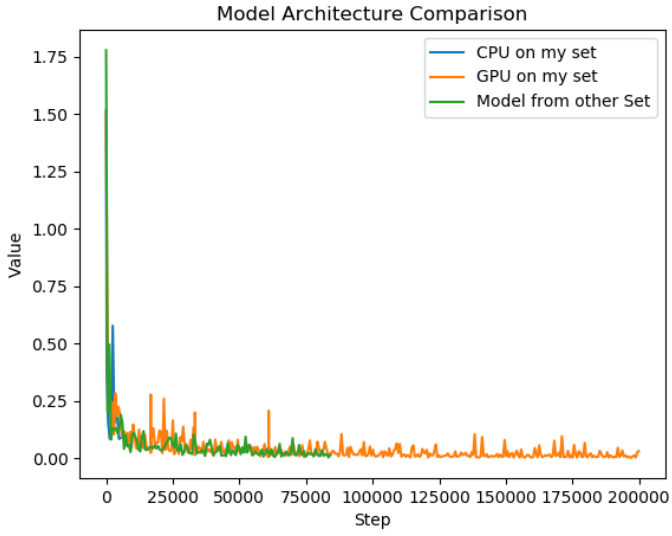

Figure 8: Loss correctly converging.

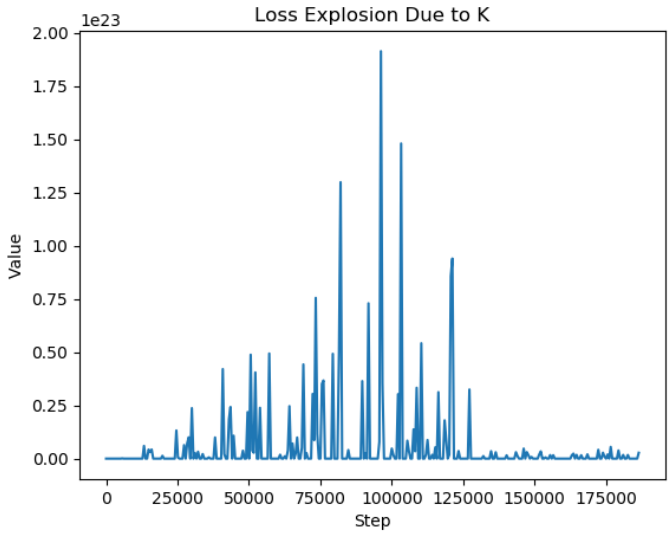

Figure 9: Loss correctly converging (zoomed in). Notice the CPU and how few steps it managed to train.

Figure 10: From left to right: GPU, my training set; GPU, other training set; CPU, my training set.

4.2 Image Performance

I selected four images from the test set to demonstrate how each of the three models performed. Although not quantitative, its fun to see and the entire point of the project was to arrive at this point. I will put all these files in the Google drive folder which I will share along with this report.

Unsurprising was the fact that the more developed, GPU trained models do better. What was surprising was the fact that the model trained on my training set seemed to do better than the model trained on the downloaded data set. Despite the larger number of images, I think that data set suffered a few short comings. Among these was the fact that the images were not taken to be used to train a network. I intentionally took images of knives that I knew would be advantageous for the network, particularly me holding knives at various angles.

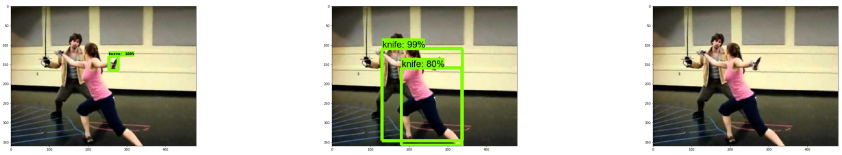

Figure 11: From left to right: GPU, my training set; GPU, other training set; CPU, my training set.

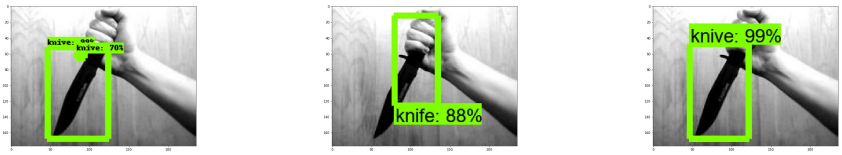

Figure 12: From left to right: GPU, my training set; GPU, other training set; CPU, my training set.

Figure 13: From left to right: GPU, my training set; GPU, other training set; CPU, my training set.

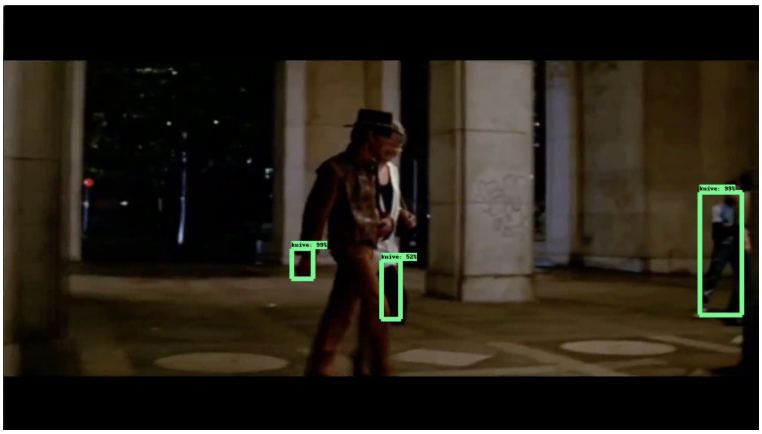

Figure 14: Example of over fitting. Notice that the features do vaguely look knife-shaped.

This may be why the models trained on my data set were able to recognize the knife in Figure 10.

4.3 Overfitting and Video

I also ran each of these models on a clip from the movie Crocodile Dundee. The shortcomings of the RCNN become more apparent in these videos because the network frequently detects knives where there are none. I believe the majority of this error is due to the lack of distinguishable features on a knife. In general, its hard to describe it beyond a long, thing piece of metal. Thus many long thin lines could easily be misconstrued with a knife. However, I think these errors are amplified by the overfitting that's characteristic of CNN networks. This was true across all the models. However, when there is actually a knife on screen, the GPU model trained on my training set again does a better job of recognizing it! Funny enough, all three models fail to detect the mugger's switchblade but are able to recognize Crocodile Dundee's knife, proving him right after all. Again, these videos (along with the original) will be in the Google drive.

Figure 15: Frame from video, model trained on internet training set.

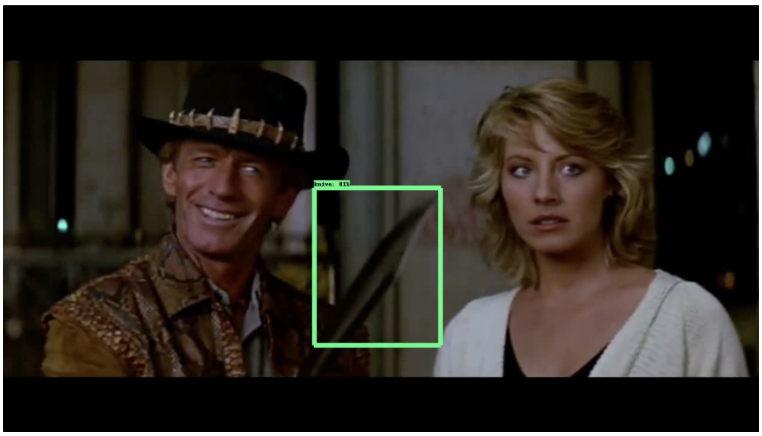

Figure 16: Frame from video, model trained on my training set.

5 Conclusion

5.1 Goals

Overall, I think I met the goals of this project. I already explained why the TPU goals ended up being unnecessary. Considering how bleak I felt about this at times in the last couple weeks, I am extremely happy with the results. Going forward, if I were to revisit this topic (which I would like to!) I would like to:

1. Figure out a way to speed up the video detection, which could be done with less accurate (but faster) models, and by downsizing the video file (the video I used in this was 1280x720).

2. Explore other types of models that could potentially do a better job of detecting knives

3. try out other objects

4. More general to Tensorflow, explore Deep Convolutional Generative Adversarial Networks, which generates images based on what an object detection network such as this one "thinks" that object looks like.

5.2 Improvements

Among some small improvements, perhaps the most drastic would be to develop a feature extraction regiment that is tailored to looking for knives. For example, you could look at reflectivity, or look for hands (which visually are easier to find than knives), and then look to see if there is a knife in the hand. However, I am very happy with the results I managed to get with a limited training set and computational power.

5.3 Reflection

Overall I am very happy that I chose this as my final project for modeling. It really challenged my skills in python and Linux command line, and definitely pushed me out of my comfort zone. It was definitely a trial by fire, but I gained an incredible amount of knowledge from the experience, most of which can't fit in this report. Ultimately the skills I learned in the past couple weeks using Tensorflow and object detection are completely unique from anything I've done before, and I expect they will come in handy at some point in the future.

5.4 Acknowledgments

I would like to thank Rachel Price for her review of my report and moral support. I'd also like to thank Evan, known as "EdjeElectronics" on Github, for his excellent tutorial on Tensorflow Object Detection. Although not perfect, it provided the framework I needed. Finally, I'd like to thank the teams that worked on COCO, Tensorflow, and the Tensorflow Object Detection API.