Using Neural Networks for Object Detection in Robotics Competitions

Nicholas Hubbard
Team 1701, The Robocubs

25 February 2019

Abstract

The goal of this paper is to serve as an overly simplified profile of how to build a neural network for detecting objects specific to a singular situation.

To this end, in the following sections, the basics of the linear algebra behind the neural network will be explained, followed by the process of gathering, profiling, and training data for the neural network.

1 Introduction

Much progress has been made in recent years on object detection through the use of convolutional neural networks (CNNs). Modern object detectors based on these neural networks have simultaneously decreased in complexity of usage and increased in performance through many years of complex research, study and critical analysis. The most common uses of CNNs are in systems like Google Photos automatic memories, which studies videos and automatically clips sections to create short memorable moments. Another example of CNNs is Pinterest Visual Search, which sort pictures into searchable categories without any user intervention and allows the user to virtually try on outfits from their search results that the algorithm believes to match the user's taste.

However, it can be very difficult to decide what architecture is best suited for applications like these. CNN architectures are like dog breeds: there are far too many to consider all of them, and some of them are polar opposites of what would make a good choice for your lifestyle. The main architectures considered for object detection with neural networks are Faster R-CNN [1], R-FCN [2], Multibox [3], SSD [4], and YOLO [5] [6] [7]. These architectures have been thoroughly tested in competitions such as COCO [8], where the fastest architecture with the most accuracy wins.

Typically, before these architectures, competitions of this sort were won by a multibox scanner; this system would take random samples of an image, search for specific image qualifiers like texture, color, or shape, and then resize the box until the whole item is selected. Unfortunately, while accurate, this solution required a high amount of compute capability and had very high latency; it could take up to ten seconds to return a result, which made this solution impractical for processing videos or large amounts of images at once.

While it is impractical to compare every recently proposed architecture, most of them have been tested with their thousands of tuning parameters. Therefore, The main contributions of this project are as follows:

- Automating the labor-intensive process of marking images using typical computer vision processes, and the similarly labor intensive process of writing annotation files for each marking in the image.
- Explaining how the different models affect results and why the singular architecture chosen for this application is the best for this application.
- Showing the results of the 100,000 training cycles for the final model.

2 An Introduction to Linear Algebra and Neural Networks

An object, as defined by a computer processing it through a static image, prerecorded video, or a live stream from a camera, is a simple sequence of rows and columns of a set size with one (grayscale), three (RGB, HSV, HSL) or four (CMYK) "channels". These channels are show as tuples within a matrix, like so:

$$\begin{bmatrix} (a_1, a_2, a_3) & (b_1, b_2, b_3) & (c_1, c_2, c_3) \\ \vdots & \ddots & \vdots \\ (x_1, x_2, x_3) & (y_1, y_2, y_3) & (z_1, z_2, z_3) \end{bmatrix}$$

This matrix of tuples is intended to replicate what human eyes see as lines of light in the form of individual pixels. It makes up the first layer of a neural network, a system designed to replicate the function of the brain in interpreting various types of information. For each row and column in the matrix, there is one individual input *tensor* (variable size 2D or 3D matrix for storing information) in the initial layer of the neural network. If a neural network accepted inputs in the form of 28 by 28 pixel RGB images, there would be 784 tensors with three values from 0 to 1 representing each portion of color in the pixel. Each number in each individual tuple is called the *activation* of each channel of color in each individual pixel. Assuming the same image composition as before, there are now three times as many inputs in the initial layer, for a total of 2,352 individual inputs.

All of these individual inputs make up the first layer of the network. Moving far forward to the last layer of the network, there is one final tensor for each possibility that the network is attempting to predict. If the goal of this network is to predict two different possible items for an image, then there are two tensors at the end representing the desired results, each with a label attached to them to provide a human-readable understanding of the final product.

Between the beginning and the end of the network, there are several *hidden layers*, which hide the complex logic required to extract features and colors, match portions of images and transform areas of images to match a specific contour or figure. Activations of the initial layer are fed into the first of many hidden layers, which can be theoretically infinite in amount, only limited by compute resources available to the operation. These activations are loosely analogous to how scientists explain neurons in the brain: one initial activation is fired by a neuron, then numerous dependent actions occur, and finally, the final neuron activation fires to finish the chain reaction and produce the desired result. Each hidden layer processes a subcomponent, or partial area, of an image to classify; as the layers progress, these subcomponents are merged together to create a resulting classification of the detected object.

Each layer of the network has an innumerable amount of dials to adjust to cause hidden layers to function in different ways. These dials adjust different functions in the network, but are all interlinked in some way. To ensure that only the important pieces of inferred information are released, they are all assigned *weights* to mark the more important and less important pieces of information as the network progresses. When the final layer is reached, all of these weights are added together and combined to make a final confidence in the result:

$$w_1a_1 + w_2a_2 + w_3a_3 + w_4a_4 + \ldots + w_na_n$$

If the network was trying to find the portion of an image that has a boundary assigned to it, all of the irrelevant information is assigned a negative weight to reduce the possibility of finding bad information, and the important information is assigned a positive weight to ensure it is used in the final calculations of the result.

After computing the weighted average, the network must rebase the weighted average to ensure that it falls between 0 and 1, as the raw activation value could fall anywhere from 1.17549×10^{-38} to 3.40282×10^{38} (this is the standard range for floating point numbers defined by IEEE-754). The most common solution for this problem is the *logistic curve*, also known as a logistic curve:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The neural network designer can also introduce *bias*, which ensures a tensor can only be activated when a number is within a specified range. This may be the simplest part of the neural network: to introduce bias, a static number is added (to ensure activation below that

number) or subtracted (to ensure activation above that number) is added to the weighted average of the tensor.

Every layer has weights and biases, making networks highly optimized input and output systems. Complexity of these networks is determined by adding the number of initial tensors times the weights for each layer, and adding the sum of every bias to the total. This quantifies the extreme complexity of the problems faced when using neural networks.

Wrapping up all of the information in this section, this is what the final resulting network looks like mathematically:

$$\begin{bmatrix} w_{0,0} \pm b & w_{0,1} \pm b & \dots & w_{0,n} \pm b \\ w_{1,0} \pm b & w_{1,1} \pm b & \dots & w_{1,n} \pm b \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} \pm b & w_{k,1} \pm b & \dots & w_{k,n} \pm b \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ \vdots \\ ? \end{bmatrix}$$

The left matrix is the weighted average (calculated using the sigmoid function), the middle matrix is the layer's tensor activation, and the right matrix is the final output sent to the next tensor. This process is repeated for every layer of the neural network, to finally retrieve a single final value for the end of the neural network.

3 Automated Generation of Training Annotations

The most labor intensive process is not attempting to understand how neural networks work, but is the process generating all of the required information to ensure that training goes smoothly. Typically, this is a very manual process using tools such as GIMP (The GNU Image Manipulation Program) and a text editor to load images of objects and mark rectangles around each object to detect, then writing down the coordinates of each object.

Instead of completing this process by hand, we automated our process using C++ and OpenCV, an open source computer vision toolkit. OpenCV defines a concept called *contours*. These are the outer edges of an object in an image. Usually, before contours become involve, the image is *thresholded* to cut out the majority of other items in the image, *eroded* to cut out small areas of an image that thresholding does not catch, and *dilated* to ensure the eroded contour has the same size as the actual object in the image. Finally, the resulting image is run through the Suzuki-Abe [9] algorithm to find the borders of the detected object, and is run through the Ramer-Douglas-Peucker [10] [11] algorithm to fit the curved contours into a rectangular shape.

The coordinates of this rectangular shape are retrieved, and written into an annotation file about the image, describing to the computer how to delineate portions of an image from the rest of the image. It is stored in the XML-based PASCAL VOC training descriptor format.

4 Training on the Dataset

Training a dataset is an extremely complex and difficult process to complete. Training on a run-of-the-mill CPU for this dataset isn't even possible - the training process takes nearly sixteen times as long per iteration, rendering a final model that can take nearly fourteen days to complete training on. Instead, the dataset must be trained using the compute cores in a GPU (known as CUDA on Nvidia platforms, or ROCm on AMD platforms) or a TPU (Tensor Processing Unit) from the likes of Google or Intel.

This performance requirement unfortunately means that training the dataset is nearly impossible without expensive graphics cards, processors, memory and discrete hardware. Fortunately, such hardware is available with a low cost penalty through the use of cloud computing. Google, for example, offers their Machine Learning Engine platform for $1.21/node/hour. (One node contains an Nvidia Tesla K80 GPU, a custom Intel Xeon processor with 8 CPU cores, and 30 gigabytes of RAM.) When training the final model for this project, our configuration consisted of a total of 9 nodes, with one master node controlling the eight worker nodes. The master node is outfitted with the same configuration described above, sans GPU; the worker nodes are outfitted with the same configuration and a GPU. The training process with this configuration ran for 2 hours and 50 minutes, and cost a total of $34.46 to complete training to 100,000 iterations.

To train our model, we started with a pre-trained model provided by Google running on the MobileNet

v2 [12] platform. This is a model platform built from scratch to run on lower-power devices without large performance and accuracy hits. Using a concept called *transfer learning*, an existing model trained on a dataset can be re-trained on a new dataset provided by the operator.

Devices that have a capable C++ compiler, such as an Nvidia Jetson TX1 or TX2, can be made to run model *inference* (an implementation of the model) with a low amount of latency and power draw. This takes a large amount of time, however, as the TensorFlow package must be compiled for every platform necessary if a suitable binary package does not already exist.

5 Optimizing the Model for Inference

There are many optimizations which can be applied to both the runtime platform and the model itself to decrease inference latency. Here are some of the solutions we used for optimizing the model for low inference time.

- **Compiling the runtime for the target platform.** This can make a significant difference in inference time. TensorFlow is completely unavailable for non-Intel platforms in a pre-compiled fashion, and OpenCV is the same. To run TensorFlow and OpenCV on an Nvidia Jetson, for example, you can compile both software packages using GPU computations with NVIDIA CUDA and NVIDIA TensorRT, and CPU computations using ARM NEON extensions. These optimizations vastly improve inference times, sometimes by a factor of 15 to 30 times faster.
- **Optimizing the model with runtime tools.** TensorFlow provides two tools to optimize and improve models for inference on less powerful platforms. The first is the TensorFlow Lite Optimizing Converter (TOCO), an automatic model reduction tool that increases performance by eliminating expensive computations and reduces size by eliminating unused connections within the model. TOCO is typically used when running a model on Android or iOS. The second tool is the Graph Transformation tool, which is a highly modular tool with many options to improve performance and reduce model size.
- **Using the original runtime.** This may sound obvious, but OpenCV offers the option to run TensorFlow models without TensorFlow being present. While convenient, since TensorFlow takes a long time to compile, the results were less accurate and slower than using TensorFlow for inference and OpenCV to visualize the inference.
- **Changing training options.** This is the most difficult option, but reduces headaches in the end of the training process. There are options within the training configuration that can significantly reduce the model size and complexity, such as pre-quantization. *Quantization* is when the training process is done exclusively on 8-bit integer numbers instead of 16 or 32-bit floating point numbers, and results in a model that runs much faster on platforms that don't have many optimizations applied. Models can be quantized after-the-fact with the Graph Transformation tool, but the results are limited because post-quantizing the model can affect the accuracy significantly.

6 Running Model Inference

Running inference on the model is typically the most difficult part of the process. It requires a very high-speed device with the machine learning runtime being heavily optimized for the device (see above). Because of this heavy need for optimization, we selected the Nvidia Jetson platform, specifically the Nvidia Jetson Nano.

Unfortunately, the Jetson Nano has not been released to the general public yet. For testing purposes before deployment, we ran initial model inference on a Dell XPS 15 with a 6^{th} Generation Intel Core i5 and an Nvidia GeForce 960M. Both TensorFlow and OpenCV were compiled explicitly for the target platform to ensure full optimization. On this platform, spectacular accuracy was observed in the setup. Because of neural network intrinsics, no calibration for lighting, shadows, or distance solving is necessary.

Here is a graph indicating performance averages over time for the various optimization methods we used. These methods are as follows:

1. **Unopt.** is an unoptimized model as created by the training process.
2. **Unopt. + TRT** is an unoptimized model combined with TensorRT live optimization.
3. **First Opt.** is a "partially" optimized model using the tactics described at the link https://git.io/fj3CV.
4. **First Opt. + TRT** is the same optimizations combined with TensorRT.
5. **Second Opt.** is a "fully" optimized model using the tactics described at the link https://git.io/fj3Cw.
6. **Second Opt. + TRT** is the same optimizations combined with TensorRT.

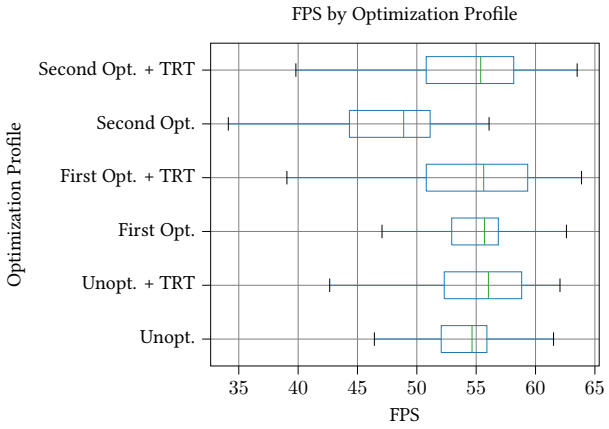

Figure 1: Range of FPS by Optimization Profile

The other problem with running inference is the sheer amount of code required to run inference is draining. To help fix this problem, we created a library for working with neural networks called **VTK**. It vastly simplifies the amount of code needed for working with neural networks. It is open source, fully documented and tested for others to use.

Acknowledgments

I would like to than the following people for their advice and support throughout the duration of this project: Noah Husby, Rich Wong, Peter Guenther, Jim Terry, Michael Vinarcik, and Elisabeth Wood. Without their support and encouragement, I don't think this massive undertaking would have been possible.

References

[1] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015. URL: http://arxiv.org/abs/1506.01497.

[2] J. Dai, Y. Li, K. He, and J. Sun. R-FCN: object detection via region-based fully convolutional networks. *CoRR*, abs/1605.06409, 2016. URL: http://arxiv.org/abs/1605.06409.

[3] C. Szegedy, S. E. Reed, D. Erhan, and D. Anguelov. Scalable, high-quality object detection. *CoRR*, abs/1412.1441, 2014. URL: http://arxiv.org/abs/1412.1441.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015. URL: http://arxiv.org/abs/1512.02325.

[5] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. URL: http://arxiv.org/abs/1506.02640.

[6] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016. URL: http://arxiv.org/abs/1612.08242.

[7] J. Redmon and A. Farhadi. Yolov3: an incremental improvement. *CoRR*, abs/1804.02767, 2018. URL: http://arxiv.org/abs/1804.02767.

[8] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. URL: http://arxiv.org/abs/1405.0312.

[9] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30:32–46, Mar. 1985. DOI: 10.1016/0734-189X(85)90016-7.

[10] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244–256, 1972. ISSN: 0146-664X. DOI: 10.1016/S0146-664X(72)80017-0.

[11] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or it's charicatures. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973. DOI: 10.3138/FM57-6770-U75U-7727.

[12] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Inverted residuals and linear bottlenecks: mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018. URL: http://arxiv.org/abs/1801.04381.