A machine learning approach to solve the Waldo puzzle

Aswanth P P, Mohammed Ameen, Joe Antony, Manjunath Mulimani and Dr. Shashidhar G. Koolagudi
Department of Computer Science and Engineering
National Institute of Technology Karnataka, Surathkal

***Abstract*—This paper explains an approach to solve the Waldo puzzle by using machine learning concepts. The Waldo Puzzle consists of a picture or an image with a lot of different characters and objects. The objective is to locate the position of a certain character named Waldo in the image. The difficulty of the puzzle is usually high, and it generally takes a considerable amount of time to find him using the naked eye. Through machine learning, and using object detection strategies, we locate Waldo in the order of a few seconds, making it a much more effortless approach.**

***Keywords*—Machine Learning, Object Detection, Tensorflow, Inception v2 Model**

I. Introduction

The Waldo puzzle can be solved by using template matching methods or using machine learning concepts. In template matching, the user needs to provide the parent image as well as the small template image consisting of the character (in our case, Waldo). The problem with this approach is that if we are giving a different Waldo image as template, the answer won't be as accurate. Hence in the template matching method, we actually need to identify Waldo in the parent image in order to feed its cropped image as the template image.

In our approach, we are using RCNN with Inception v2 model which is already trained to detect pets from an image (trained on COCO data sets). While the model could be trained from scratch starting with randomly initialized network weights, this process would probably take weeks. Instead, we use a method called transfer learning.

II. Literature Review

The principle of object detection has a variety of applications like vehicle detection, people counting, self-driving cars, image classifying and much more. It deals with detecting instances of semantic objects of a certain class (humans, cars, etc.) in digital images and videos. The application of this concept into our problem statement has been inspired by some of the existing methods.

There are some existing functions in OpenCV, which are useful for detecting faces and patterns by giving a source image and a subset of the image containing the same desired face or pattern[1]. But the main problem here is that the template should be exactly same as that of desired pattern. Which means that, different brightness and differently oriented images will not give an accurate result. We tested this method of template matching in OpenCV on our problem statement but it resulted in inaccurate results.

In today's world, almost any real life object's features can be trained into a model, and can be used for further applications. There are APIs and frameworks available that enable us to do the same, and perform the detection of certain objects in images. For example, some of the existing models perform the detection of certain pets, like dogs and cats, in an image.[2] This must, however, be supported with an apt dataset covering a wide range of possible pets (objects) of the same kind. Providing a good dataset was essential to the training of Waldo in our case. The existing implementation of detecting pets was the inspiration to derive a solution to the Waldo puzzle. This was trained on the Common Objects in Context (COCO) dataset available on the internet. This model (which was available in the Tensorflow Object Detection API) was taken, and was retrained into a brand new model using Waldo's physical features, a method known as transfer learning[3].

III. Proposed Methodology

As mentioned in the previous section, the methodology proposed here to perform the detection of Waldo, is called transfer learning. This method, simply stated, involves taking an existing model trained to solve some general problem, and using that model we create our own model to detect Waldo by retraining the existing model[3].

In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Also, by not developing a model from scratch and instead perform transfer learning, we can use the knowledge obtained in the pre-trained model and transfer it out to new one[3]. This saves us a lot of time because the time spent for training can be invested into obtaining only the knowledge specific to our problem.

The dataset which was used for the initial model training was based on the Common Objects in Context (COCO) dataset available on the internet. There are mainly two important factors to consider before performing transfer learning. One is the size of the new dataset (small or big), and the other is its similarity to the original dataset. In our case, the new dataset is relatively small, and doesn't have a distinguishable similarity to the original dataset, since the COCO dataset differs quite a lot from the Waldo dataset. Since the data was small, we thought it was likely best to only train a linear classifier. And also, since the dataset is very different, it might not be best to train the classifier from the top of the network, which contains more dataset-specific features. Instead, it would be better to

train the classifier from activations somewhere earlier in the network.

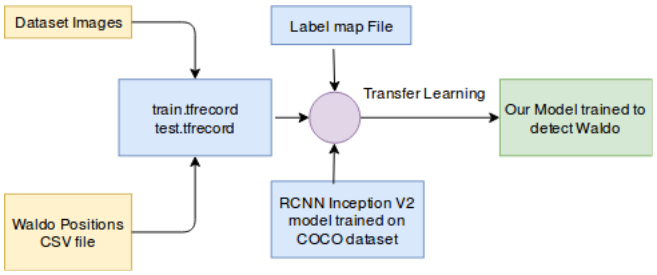

Fig. 1: Flow diagram for retraining Inception v2 Model

There are a few things to keep in mind, when we perform the process of transfer learning. One, is the constraints from pre-trained models. That is, if we wish to use a pre-trained network, we may be slightly constrained in terms of the architecture we can use for our new dataset. For example, we cant arbitrarily take out Convolutional layers from the pre-trained network. Although some changes are straightforward. Also, it is common to use a smaller learning rate for Convolutional Network weights that are being fine-tuned, in comparison to the (randomly-initialized) weights for the new linear classifier that computes the class scores of our new dataset.

IV. Implementation

A. Preparing the Datasets

Even though dealing with neural networks is the most notable process in deep learning, it turns out that data scientists spend the most time on the process of preparing and formatting training data.

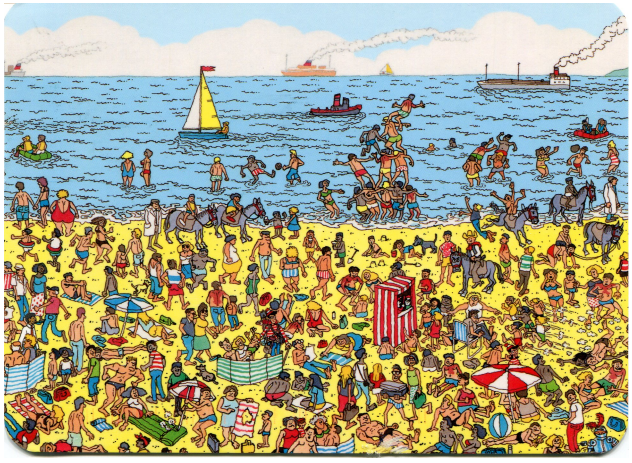

Fig. 2: Source image

Target values for the most simple machine learning problems are usually scalars or a categorical string. The Tensorflow

Fig. 3: The Waldo character

Object Detection API training data uses a combination of both. It consists of a set of images accompanied with labels of desired objects and locations of where they appear in an image. The locations of the objects are defined with two points since two points are enough to draw a bounding box around an object.

So in order to create the training set, we needed to come up with a set of Wheres Waldo puzzle images with locations of where Waldo appears. The collection of Waldo puzzles and marking the locations of Waldo on each image, was a tedious task. There weren't an ample number of Waldo puzzles that we could find, although we did manage to find a sufficient number to train our model to a respectable accuracy. Marking the locations of Waldo in the training dataset had to be done manually as well, which proved to be a considerable task. Once this was done, the final step was to pack our labels containing the images and the locations into a single tfrecord file.

B. Preparing TFRecord File

We have collected images for the dataset and created a csv file containing the position of Waldo in those images. The fields in the csv file are :

filename width height class xmin ymin xmax ymax

This binary file conversion has been done because, we dont have to specify different directories for images and ground truth annotations. While storing the data in binary file, we have our data in one block of memory, compared to storing each image and annotation separately. Opening a file is a considerably time-consuming operation, especially if you use hdd and not ssd, because it involves moving the disk reader head which takes quite some time. Overall, by using binary files you make it easier to distribute and make the data better aligned for efficient reading.

This binary file conversion is done by using a python script where it is stored as numpy array in python and it returns test.tfrecord and train.tfrecord which is used further for retraining the model using transfer learning.

C. Preparing the Model

Training a model from scratch requires a large number of labeled training data and a lot of computing power (hundreds of GPU-hours or more). Transfer learning is a technique that shortcuts much of this by taking a piece of a model that has already been trained on a related task and reusing it in a new model[4].

Tensorflow Object Detection API has a collection of pre trained models with varying performances (differing in what is optimized - speed or accuracy) trained on a number of public datasets. We made use of the RCNN with Inception v2 model already trained on the COCO dataset. COCO is a large-scale object detection, segmentation, and captioning dataset.

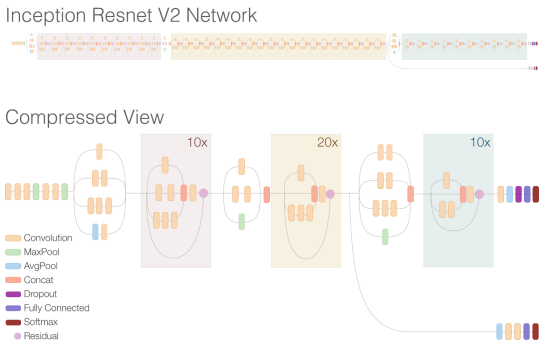

Fig. 4: Schematic diagram for Inception V2 model

The first phase in transfer learning is to analyze all the images on disk and calculate and cache the bottleneck values for each of them. 'Bottleneck' is an informal term we often use for the layer just before the final output layer that actually does the classification. This penultimate layer has been trained to output a set of values that's good enough for the classifier to use to distinguish between all the classes it's been asked to recognize[5]. That means it has to be a meaningful and compact summary of the images, since it has to contain enough information for the classifier to make a good choice in a very small set of values. The reason our final layer retraining can work on new classes is that it turns out the kind of information needed to distinguish between all the 1,000 classes in ImageNet is often also useful to distinguish between new kinds of objects.

Because every image is reused multiple times during training, and calculating each bottleneck takes a significant amount of time, it speeds things up to cache these bottleneck values on disk so they don't have to be repeatedly recalculated.

Since we have already created test.tfrecord and train.tfrecord and RCNN with Inception V2 model has been trained with the COCO dataset, along with its pipeline configuration file, we can start training the model for our purpose.

The model includes a checkpoint .ckpt file which we can use to start the training.Also we need to add a label.txt file which specifies the class which we need to classify from the rest of the source image. In our case, that class is Waldo.

In conclusion, we should have the following:

1. A pretrained model with a .ckpt checkpoint file
2. Training and evaluation .tfrecord dataset
3. Label map file
4. Pipeline configuration file pointing to the files above

Tensorflow Object Detection API provides a simple-to-use Python script to retrain our model locally.It can be found in the Tensorflow GitHub repository. By executing this script along with specifying necessary file and labels, we retrained the model. We can stop the learning by terminating the script. But generally this termination is done when the loss on our evaluation set stops decreasing or is generally very low (below 0.01 in our example).

After training, we need to export an inference graph from the stored checkpoint (which is located in our train directory).The script to these conversion can be found in the Tensorflow GitHub repository. This exported inference graph can be used to detect Waldo using a python script.

V. Results and Discussion

The model was successfully trained using the method of transfer learning as stated above. The next step was to evaluate the correctness and reliability of our model. For this, we had prepared testing data so that we could run the dataset against our trained model to check if Waldo's position could be found out from the testing data images.

We created a python script to import our trained model along with the model weights, and to perform the actual detection of Waldo on any given testing image. We set the parameter of probability to be 10 percent, that is, if the program runs on the model, and does not find any occurrence of Waldo with probability greater than 10 percent, then it concludes that Waldo doesn't exist in the given image. Once Waldo has been detected, it generates an image with the position of Waldo and the accuracy of the detection.

The following image shows how the output of the detection script would look like.

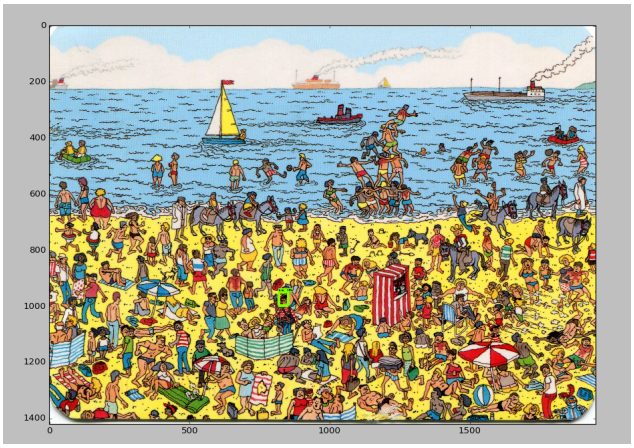

Fig. 5: Output of Waldo detection script with the green box showing the position of Waldo

It is worth noting that our trained model managed to find the position of Waldo for all the testing data collected. The model was also tested against random examples from the internet and it worked decently for them as well. Although one drawback was that our model failed to find Waldo where he was really large, which by intuition should be even easier to solve as opposed to finding him where hes really small. This indicates

that our model probably overfit our training data, mostly as a result of using only a handful of training images.

Since the availability of good dataset images for the Waldo puzzle is scarce, we wrapped up our research on the model based on the currently available images, and concluded that the method we used was effective to solve the Waldo puzzle.

REFERENCES

[1] Xianghua Fan, Fuyou Zhang, Haixia Wang and Xiao Lu , *The System of Face Detection Based on OpenCV*

[2] Bang Liu, Yan Liu and Kai Zhou, *Image Classification for Dogs and Cats*

[3] Sinno Jialin Pan and Qiang Yang, *A Survey on Transfer Learning*

[4] Lisa Torrey and Jude Shavlik, *Transfer Learning*

[5] Guoqiang Peter Zhang, *Neural Networks for Classification: A Survey*

[6] Xinyi Zhou, Wei Gong, WenLong Fu and Fengtong Du , *Application of Deep Learning in Object Detection*