COMP3330 - Assignment 1c Report

Leala Darby (c3279478), Radhika Feron (c3256870), Cody Lewis (c3283349), Zac Welsh (c3279122)

August 1, 2019

1 A Bottle and Can Detector Using Deep Neural Networks

1.1 Supervised Approach

1.1.1 Getting Data

This task required the development and annotation of a new data set specific for the given task. We acquired images of cans and bottles using a Google image scraper API, and the particular functions of this library which we used are included within the code of this program. However, we found that the majority of the resulting images had white backgrounds, so we retrieved more images from `http://www.image-net.org/`. These images have a good amount of variation in their lighting and backgrounds. For our test data, we used several extra Google queries constructed using more specific keywords, such as brand names and locations. We also took a few photos ourselves, both with and without bottles and cans. The images were taken in a variety of outdoor locations including gardens, in parks and in the Australia bush on the University's campus. The final set of training data we used had 2784 images, with 1805 containing cans and/or bottles, while our test data consisted of 206 images with 97 containing cans and/or bottles.

To add bulk to the image data set, models of cans and bottles were synthetically generated and placed in random positions within 360 degree High Dynamic Range (HDR) images. HDR images reproduce a greater dynamic range of luminosity than in standard digital imaging. It has realistic lighting that can be used to produce light rendered scenes with the 3D models of cans and bottles. The HDR images were obtained from the HDRI Haven (`https://hdrihaven.com/`).

Blender 2.79b was used to generate models to mimic real bottle and cans, and to establish the virtual HDR scene. Metal materials were used to create the cans, and transparent plastic materials were used for the plastic bottles. Figure 1 shows the untextured models of the bottle and can created in Blender. Labels from well-known brands were added to each bottle and can. The orientation of the bottles and cans, as well as the camera orientation, position and distance from the bottles and cans were randomly varied to generate synthetic data with variation. This data set of 405 synthetic images was added to the data set containing the real images.

Additionally, we experimented with the Python library Imgaug (`https://github.com/aleju/imgaug`) in order to further diversify the dataset. This allowed a single photograph to be replicated in multiple augmented forms, extending its utility and quickly increasing the size of the dataset from the small sample of photos originally tested on. Techniques applied in various randomised combinations included Gaussian blur, adjustments to brightness, colour and contrast, the addition of noise, mirroring flips, image rotation and cropping.

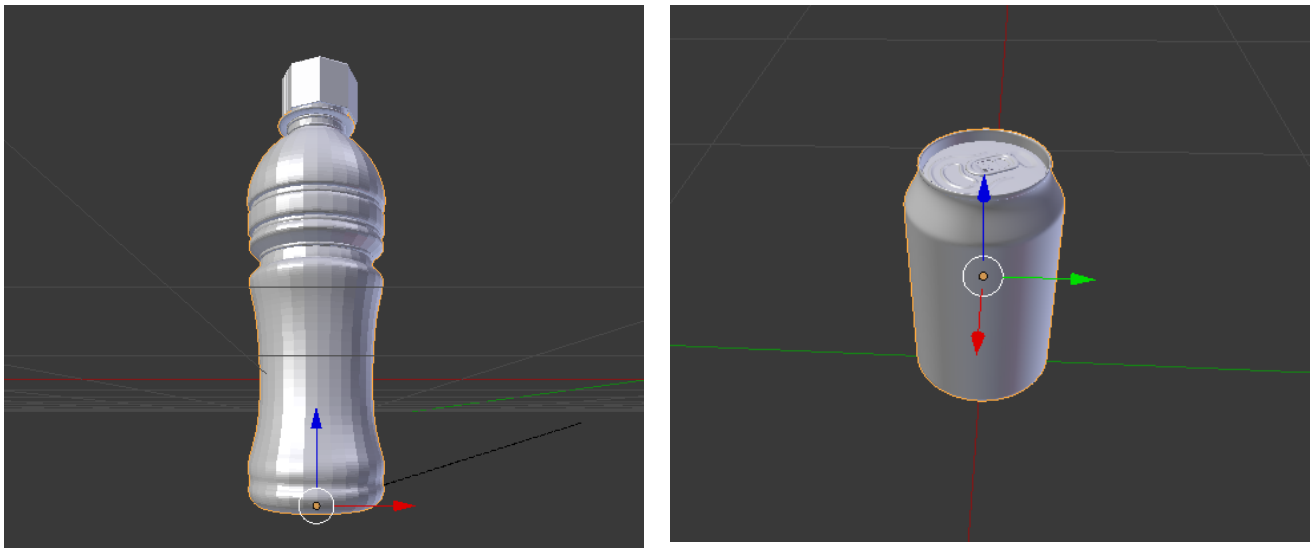

Figure 1: Untextured model of plastic bottle (left) and aluminium can (right)

Example images generated synthetically are shown in Figure 2 with their annotations. The bounding boxes were drawn around cans and bottles using the image annotation tool, LabelImg (`https://github.com/tzutalin/labelImg`).

Figure 2: Examples of synthetic cans and bottles placed in real HDR images, with annotations

1.1.2 Constructing and Training the Network - VGG19

We decided to perform transfer learning from VGG19 proposed by Simonyan et al. [4]. As such, we changed the top layer to a Flatten, Dense with 512 neurons and relu activation, a Dropout of 0.25, a Dense with 512 neurons and relu activation, and an output layer of Dense with 2 neurons and softmax activation. The network was trained for 26 epochs, taking 2 days and 16 hours, giving the error curve shown in Figure 3 and the accuracy curve shown in Figure 4. We chose to use the network from epoch 20 as it gets to the maximum acquired accuracy while having the lowest risk of overfitting.

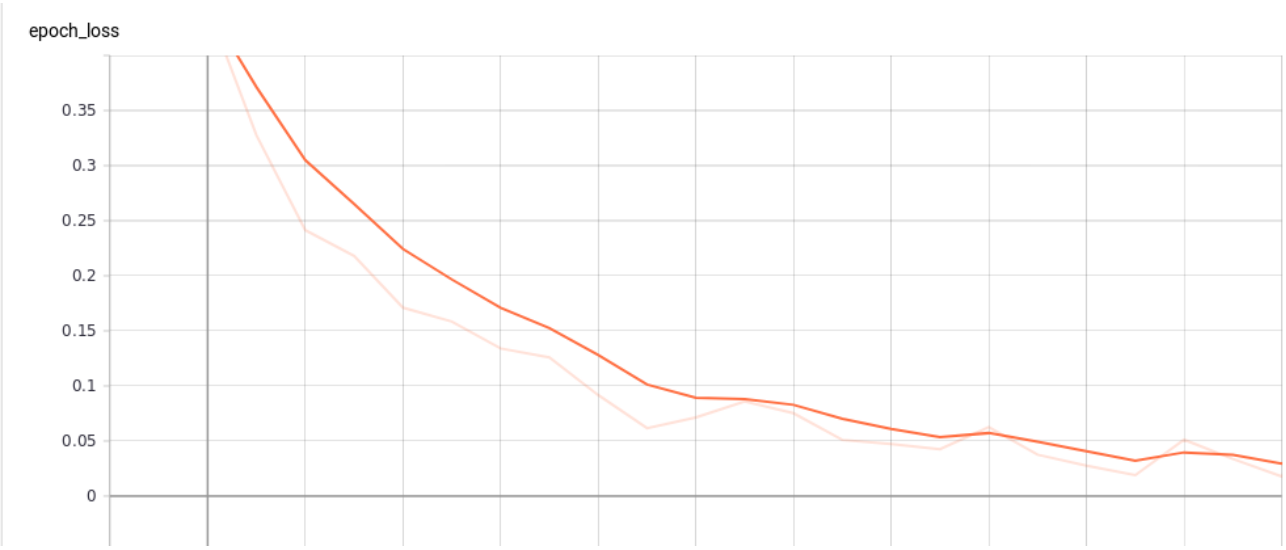

Figure 3: Loss for the VGG19 transfer

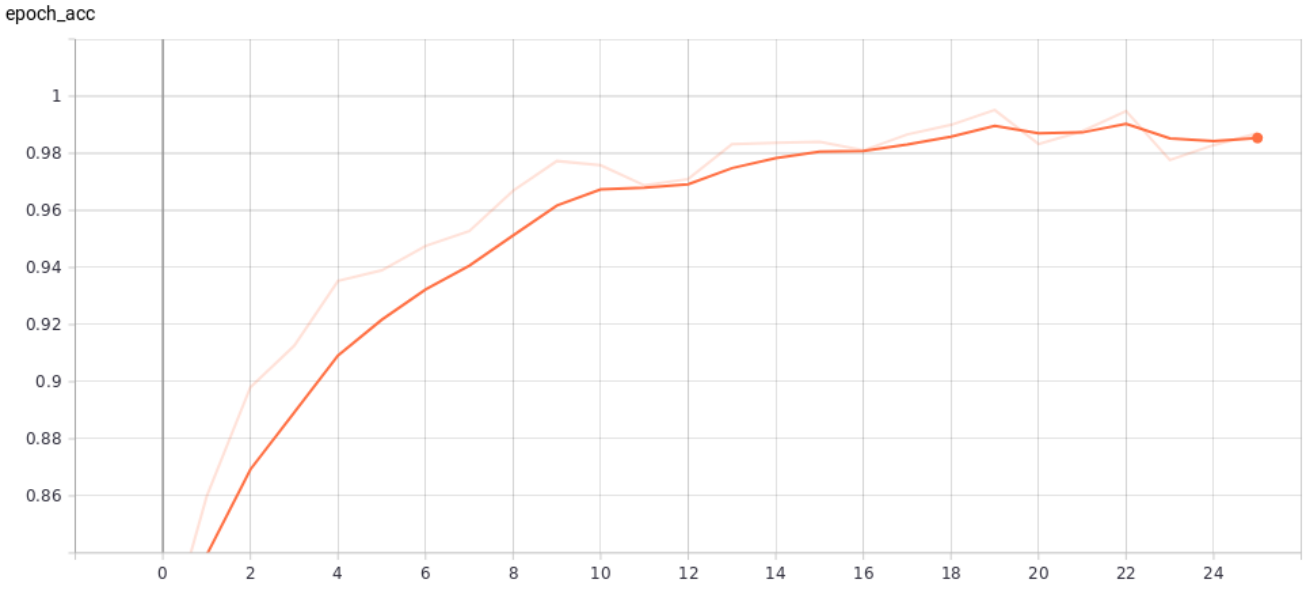

Figure 4: Accuracy for the VGG19 transfer

However, this ended up being overfitted and quite bad at finding bottles and cans, so we tried alternative strategies.

1.1.3 Constructing and Training the Network - SSD300

We implemented transfer learning with the Single Shot MultiBox Detector model, specifically using the SSD300 network architecture. Our training was based upon the SSD300 Keras implementation at `https://github.com/pierluigiferrari/ssd_keras`. This network has been trained on the PascalVOC 2007 dataset which contains 20 object classes. To train the original SSD300 model on our own dataset, we sub-sampled the weight tensors of the classification layers for fully convolutionalised SSD300 model. The weight tensors of the classification layers of the Pascal VOC model does not have the correct shape for our new model which only has 1 object class. This

single class is also entirely different from the classes in fully trained SSD300 model. However, the trained weights are still a better starting point for the training than random initialisation.

The weights for all the layers, except the classification layers, don't need to be changed. SSD300 has 6 classification layers. Iterating over these classifier layers of the source weights, the kernel and bias tensors are obtained and subsampled, ensuring the first 3 axes of the kernel remain unchanged. As an example of how we subsampled, let us consider the "con4_norm_mbox_conf" layer. SSD300 Pasval model has 20 classes, but also one 'background' class, resulting in effectively 21 classes. This layer predicts 4 boxes for each spatial position where it has to predict one of the 21 classes for each. Thus the kernel and bias will have a size of $4 \times 21 = 84$ elements. For our dataset we have 1 class, and if we include the 'background' class then we have 2 classes effectively. Similarly, we have to predict one of those 2 classes for each of the four boxes, meaning we need $4 \times 2 = 8$ elements. Since our object classes have no relation to the classes in the Pascal VOC dataset, it is appropriate to pick these 8 elements randomly from the bias and kernel vectors. This process was repeated for the other 5 classifier layers.

Figure 5 shows the loss obtained from the training and validation datasets during 50 epochs at 120 steps per epoch.

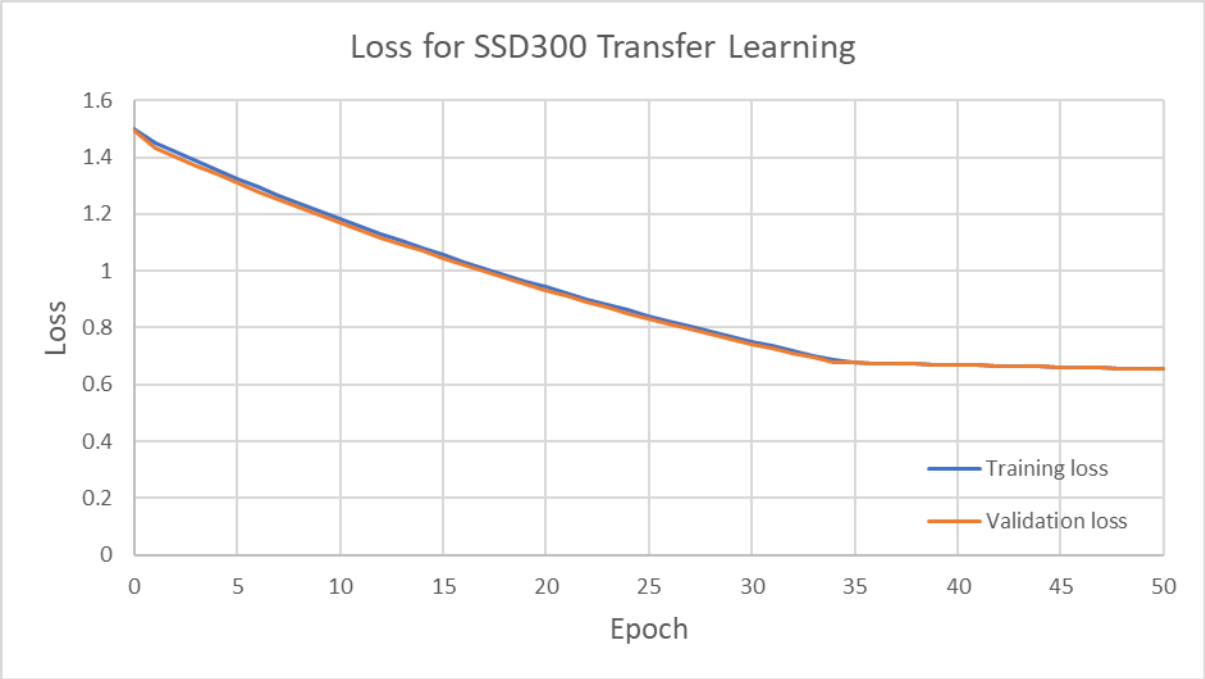

Figure 5: Validation loss and training loss for 50 epochs for SSD300 transfer learning

The loss was steadily decreasing and appeared to be quite promising. However, when the model was used on images from the validation set, the model drew bounding boxes all over the image despite the low loss. An example is shown in Figure 6. The high confidence levels suggest that there was an error in the way the model was trained; possibly in the way the training data was fed to the network, or in the formatting of the data.

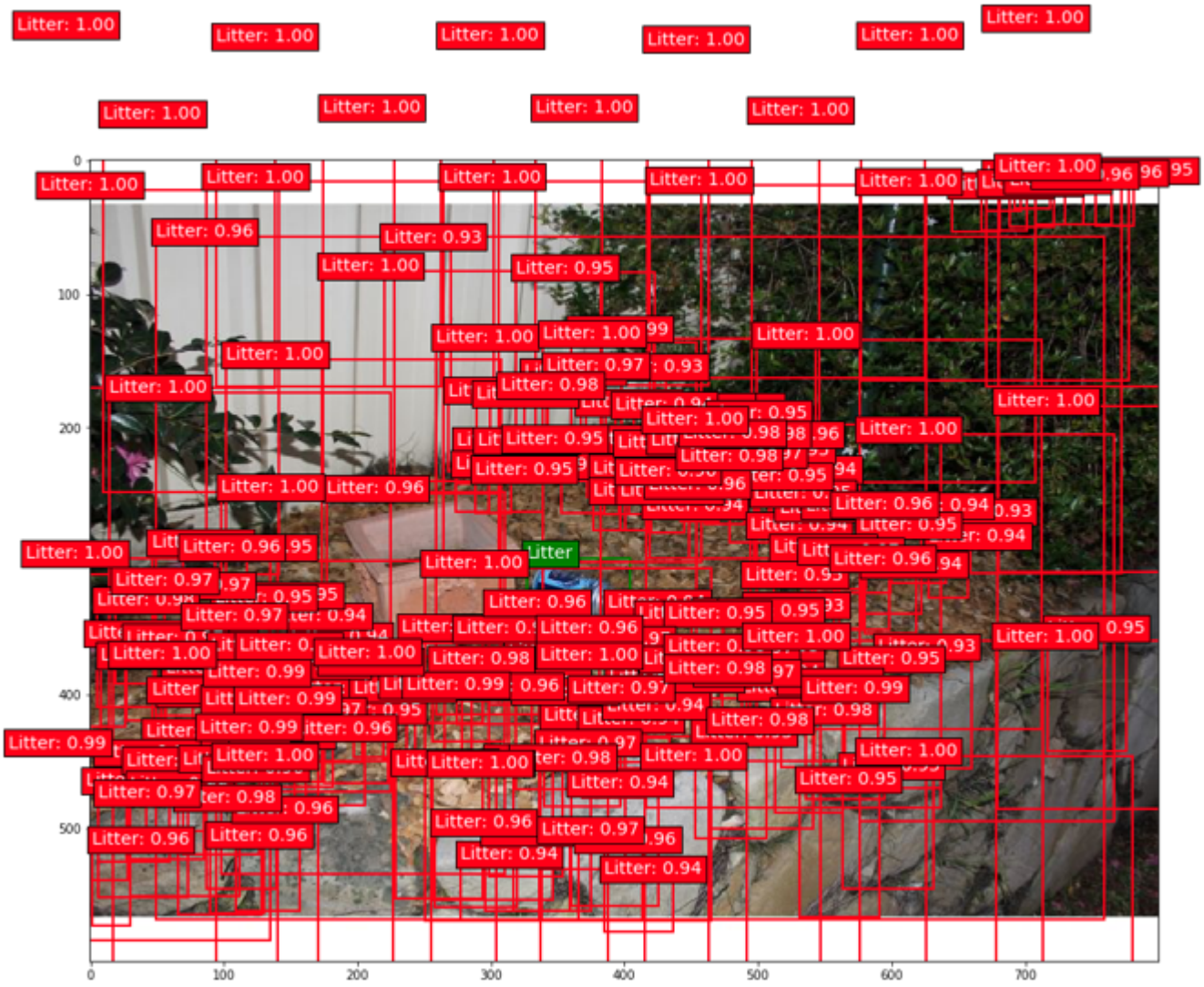

Figure 6: Example predictions made for an image from the validation set

This trained SSD300 transfer learning model can be found on OneDrive `https://uoneduau-my.sharepoint.com/:f:/g/personal/c3256870_uon_edu_au/EuzPp2IciopEgf1ZHJ5TAiQBTh9Nj1cmnKz7JtUkimt48A?e=Pr2iVv`. The use the model, run the Jupyter Notebook "ssd300_inference.ipynb" and follow the instructions. The main parts that will require changes will be the paths to the test image and to the trained model located within the zip file.

1.2 Un-Supervised Approach

1.2.1 Constructing and Training the Network

We implemented a Deep Convolutional Generative Adversarial Network based on the one proposed by Radford et al. [2], with reference to the Keras implementation at `https://github.com/eriklindernoren/Keras-GAN/blob/master/dcgan/dcgan.py`, which our implementation was initially incredibly similar to. However, we modified the hyper-parameters such that the network would hopefully generate and discriminate images resembling bottles and cans with the dimensions $224 \times 224 \times 3$. After 4000 epochs, the discriminator reached an error of 7.971192359924316, and the generator reached an error of $1.192093321833454 \times 10^{-7}$. Figure 7 shows an example image produced from the resulting generator.

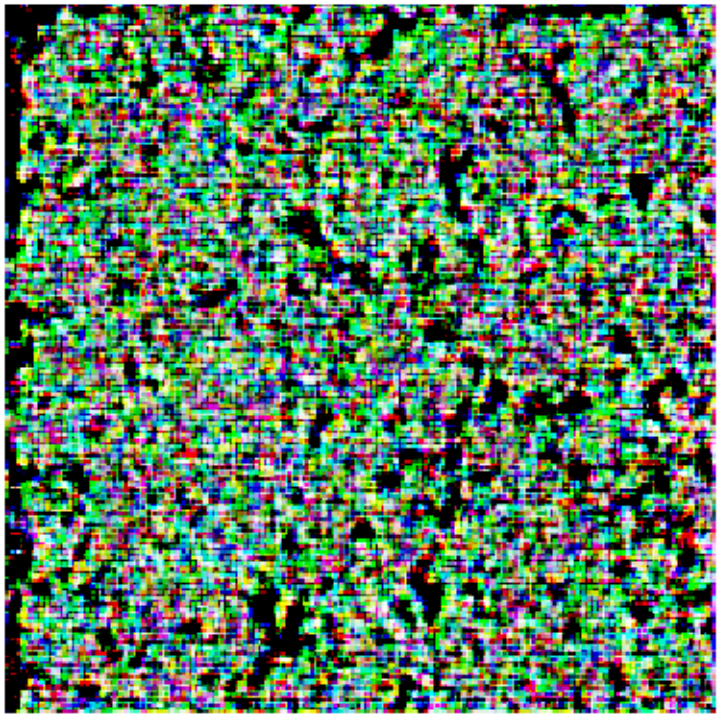

Figure 7: The final generated image

This initial attempt was unsuccessful, so we modified the models to have 4 times the number of neurons, as the image produced here is four times the size of the MNIST images. However, this took up more resources than we had available, so we tried reducing it down to 2 times the number of neurons, and using greyscale images. This ended up converging to the same error values, and creating similar images, therefore rendering this attempt unsuccessful. Despite this, we retained the code we used and modified for this approach: it is located at **Q1/GAN/__init__.py**.

1.3 Final Approach

1.3.1 Constructing and Training the Network

For our final attempt, we constructed a DNN using transfer learning from MobileNetV2, and replaced the final layer with a Flatten, Dense with 1024 neurons and relu activation, a Dropout of 0.25, a Dense with 1024 neurons and relu activation, and an output layer of Dense with 2 neurons and softmax activation. We used the same data set as previously, but also applied random flipping, rotations, and color shifting in order to create more training data. This model gained a greater degree of accuracy much faster than that of our first attempt. During its first epoch it had already developed 90% accuracy.

1.3.2 Predictions and Creating Bounding Boxes

Since we have created a classifier that generates a binary outcome, we decided to implement a sliding window system inspired by [3]. We use the intersection of various overlapping predictions to create the final bounding

box.

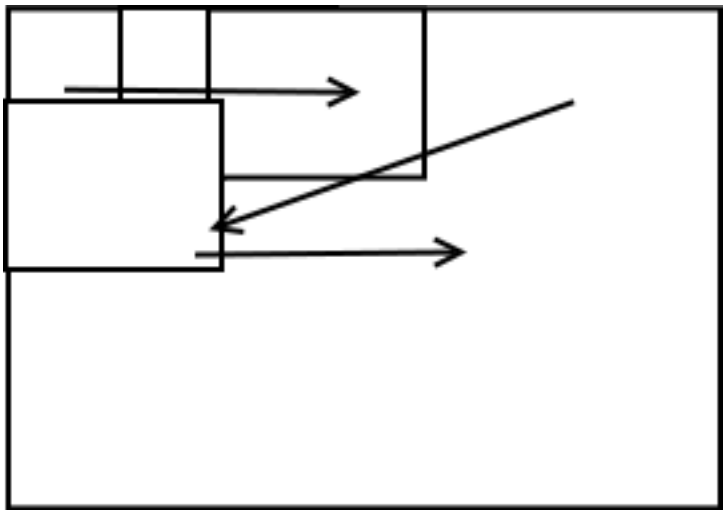

Figure 8: The movement of a sliding window

Each window is 224×224 pixels, and has a fraction of overlap with both the next and previous window. The window moves across the image from left to right, and from top to bottom.

We deployed a slightly modified algorithm for this, in order to account for a bounding box intersecting multiple times. This is shown in Algorithm 1. The intersection/union is chosen based the amount of times the box has intersected with another, and the algorithm itself simulates a multi-tape Turing machine where one of those tapes

are used to simulate a push down automata, with a stack size of 1.

Algorithm 1: Intersect Bounding Boxes

```
Data: A list of the predicted bounding boxes
Result: The least amount of bounding boxes that contain the predicted contents
initialize result boxes list;
counter = 0;
stack = NULL;
while counter < length of bounding boxes list do
    if stack is NULL then
        stack = bounding boxes[counter];
    end
    stack changed flag = FALSE;
    for i = counter to length of the bounding boxes list do
        if stack ∩ bounding boxes[i] ≠ ∅ then
            if stack has intersected > 4 times ∨ bounding boxes[i] has intersected > 4 times then
                stack = stack ∪ bounding boxes[i];
            end
            else
                stack = stack ∩ bounding boxes[i];
            end
            stack changed flag = TRUE;
            break;
        end
    end
    if ¬ stack changed flag then
        add stack to the result boxes list;
        stack = NULL;
    end
    counter = counter + 1;
end
if stack ≠ NULL then
    add stack to the result boxes list;
end
return result boxes list
```

The main issue with this form of object detection is that it takes a considerable amount of time, due to the repeated process of the window sliding across the image.

Figure 9: Before application of the Algorithm (left), after application of the Algorithm (right)

1.4 Results

Our first attempt at training this network gave constant false positives as shown in Figure 10. This led to us acquiring more data for training the model, so as to require fewer epochs.

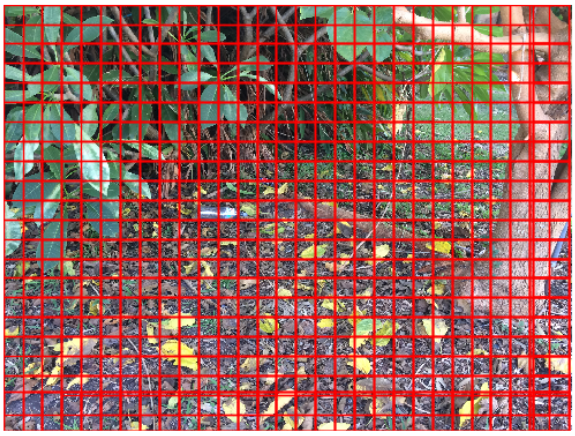

Figure 10: First object detection attempt, with no application of the bounding box intersection algorithm

We created a new data set by taking slow motion videos and converting those into a series of images. A 20 second video would produce a few thousand images, each of which was useful for classification training. After training the same model, with its default weights, we got the result shown in Figure 11.

Figure 11: The resulting classifier

The **README.md** file at the root of the **Q1** directory specifies our program's requirements and running instructions. Listing 1 details how to run the predictor across a directory or images, assuming the requirements as per our document are already fulfilled.

```
python3 BottleFinder.py -l Final.h5 -p <path to directory>
```

Listing 1: Command to run the detector

2 Manual Hyperparameter Tuning

We chose a baseline model based on MobileNet as proposed by Howard et al. [1]. The only change we made was implementing a third of the amount of neurons to speed up training. We chose this particular model as it is reportedly useful for binary classification tasks that use images as input. Additionally, it is generally efficient and light weight. We trained each of the models for three epochs, each consisting of 1000 training samples in batches of 50, and used 20% of the data as validation, using 200 samples. The following networks were trained using quad core Intel i7-4510U CPUs, as the GPU we had access to, an NVIDIA GeForce 820M, is only marginally faster for training than the CPU and holds legacy status. Hence, any version of tensorflow greater than 1.90 would refuse to run on the GPU.

2.1 Baseline Results

The baseline model consists of a MobileNet-based system, with a third of the regular amount of neurons, and uses the ADAM optimizer at a learning rate of 0.0002. After 3 epochs of training, each taking about an hour and 40 minutes to complete, the model reached 95.18% accuracy, and a loss of only 0.1388. The changes in accuracy and loss between the second and third epochs were minimal.

2.2 Effects of Halving Representational Capacity

The increase in accuracy was more steady, and the epochs were much quicker to complete, taking only 40 minutes. Each epoch had a substantial jump in accuracy, ranging from a 20% to 10% increase. The final accuracy and loss were similar to that of the baseline, being 90.15% and 0.2415 respectively, despite the differences in the training specifications.

2.3 Effects of Tripling the Representational Capacity

This resulted in similar accuracy to that of the baseline, and similar to the baseline, the changes in loss and accuracy between the second and third epochs were minimal. This model could return slightly higher accuracy rates in fewer epochs than that of the baseline, but the time and resources that each epoch took were greatly increased, as it took about 8 hours for a single epoch of training.

2.4 Effects of Changing the Learning Rate

The optimizer we chose for this project was ADAM, so the low rate had a value of 0.0002, the medium rate was 0.001, and the high rate was 0.01.

2.4.1 A low rate

The low rate was used in the baseline, resulting in a loss of 0.1388, and minimal changes in loss and accuracy between the second and third epochs. The accuracy changes from 91.16% to 95.18% between epochs 2 and 3.

2.4.2 A middle rate

This resulted in a similar, although improved, rate of loss and accuracy compared to the baseline, with 0.1284 and 95.63% respectively. The change had no significant impact on the time taken for each epoch. There were still minimal changes, albeit less significant, between the second and third epochs, increasing from 95.21% to 95.63%. It seems that the heightened learning rate allows the model to more quickly converge to an extrema within the hypothesis space.

2.4.3 A high rate

The high learning rate caused the accuracy to change radically between epochs, from worse than the baseline to slightly better, and finally, worse again in the third. The final accuracy was 92.28%, and the final loss was 0.2301. The inability to get significantly more accurate after the second epoch is likely due to the high learning rate causing large jumps, moving across either side of an extrema of the hypothesis space.

Hyperparameter change	First Epoch Accuracy (%)	Second Epoch Accuracy (%)	Final Accuracy (%)	Final Loss	Time Taken per Epoch (hours)
Baseline	74.80	91.15	95.18	0.1388	$1.\dot{6}$
Half Representational Capacity	65.22	80.07	90.15	0.2415	$0.\dot{6}$
Triple Representational Capacity	75.97	94.89	96.13	0.1129	8
Low Learning Rate	74.80	91.15	95.18	0.1388	$1.\dot{6}$
Medium Learning Rate	84.46	95.21	95.63	0.1284	$1.\dot{6}$
High Learning Rate	71.36	92.28	92.33	0.2301	$1.\dot{6}$

Table 1: Summary of the changes that the Hyperparameters make

References

[1] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[2] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[3] Johannes Rieke. Object detection with neural networks – a simple tutorial using keras. `https://towardsdatascience.com/object-detection-with-neural-networks-a4e2c46b4491`.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.