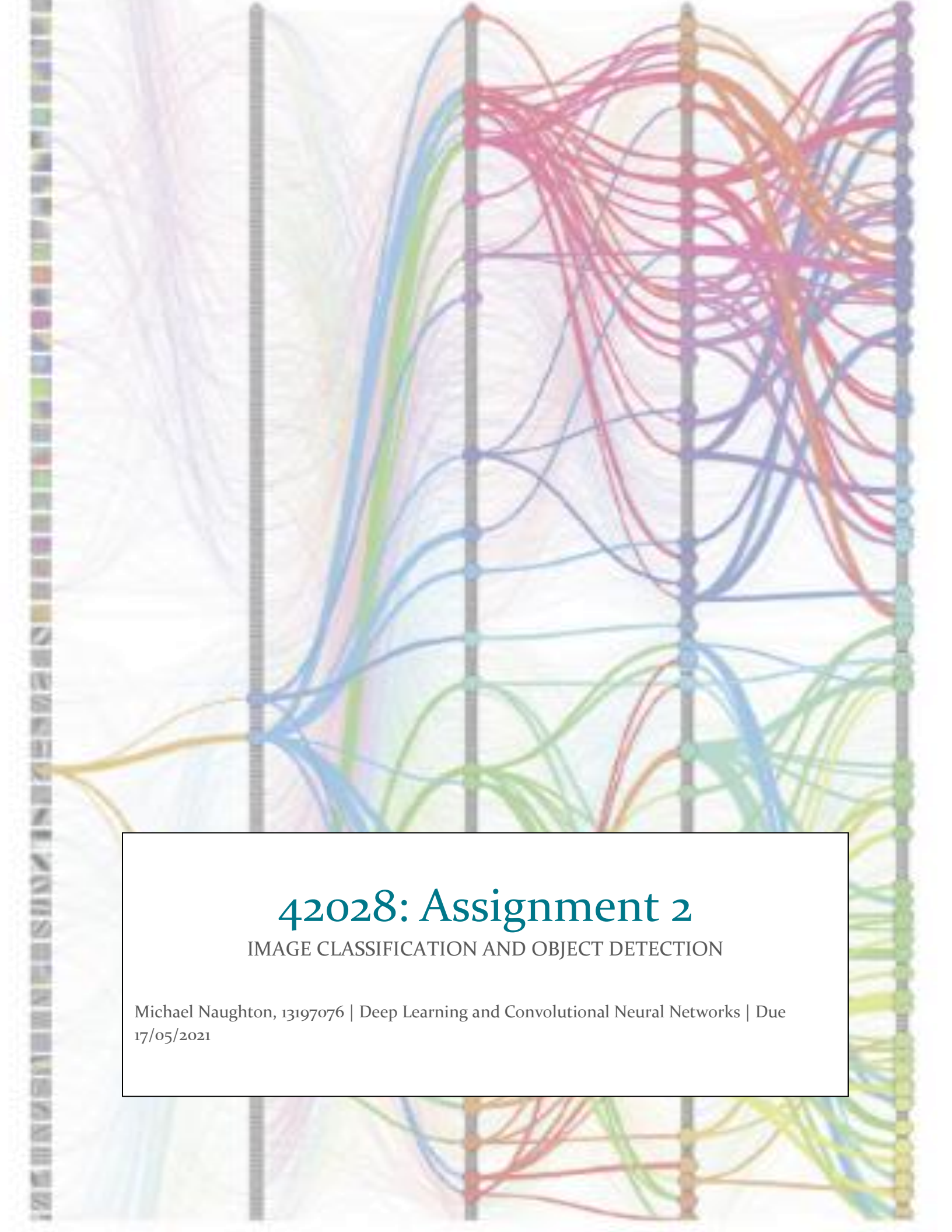

42028: Assignment 2

IMAGE CLASSIFICATION AND OBJECT DETECTION

Michael Naughton, 13197076 | Deep Learning and Convolutional Neural Networks | Due 17/05/2021

1. Introduction

For the image classification problem, I selected the Fruit 360 dataset[1] and used a baseline deep learning architecture implemented by Mihai Oltean[2]. For the object detection problem, I selected the PKLot[3] dataset, and also the Red Blood Cell dataset. For these, I utilized the Faster R-CNN and SSD-MobileNet-V2 pretrained models offered by the Tensorflow Object Detection API.

ABOUT THE DATASETS

In the Fruits 360 dataset, there are a total of 90,483 images, of which are split into a 1/4 train/test split. Within the dataset the are a selection if about 131 different classes of fruits and vegetables. Each image was obtained by planting the target object in a shaft of a low-speed motor, and a short movie was captured by a Logitech C920 camera. The researchers also wrote a dedicated flood fill algorithm to extract the objects from the background as to minimize the lighting differential between samples in the training process. The images were then scaled down to 100x100 pixels.

The proposed use case for this dataset is for use in automated produce farming, and farming analysis.

Figure 1: Left side is the original image, right is after the background removal

[1] https://github.com/Horea94/Fruit-Images-Dataset

[2] https://www.researchgate.net/publication/321475443_Fruit_recognition_from_images_using_deep_learning

[3] https://public.roboflow.com/object-detection/pklot

In the PKLot dataset, there are a total of 12,416 images, with 711,856 corresponding annotation objects. Each annotation labels the state of the parking spot to be either occupied (335,735 samples), or non-occupied (376,121 samples), and includes the dimensions of the bounding box for each respective parking space.

For each parking spot object, the original dataset's annotations contained bounding box co-ordinates, and also skew adjustment for each bounding box as seen below. The data provided through roboflow did not have this information, so the bounding boxes had no rotation feature in the output.

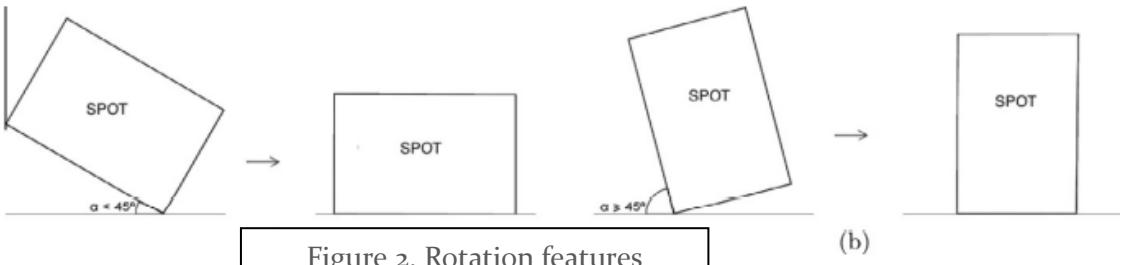

Figure 2. Rotation features

The dataset was built from images captured of parking lots in the Curibata region of Brazil. They focused on a low cost means of collection, using a Full HD Microsoft HD-500 webcam to capture images every 5 mins over a period of 30 days. The camera was positioned from a high vantage point as to minimize occlusion from potential adjacent vehicles. There is a fair representation of images in different weather and lighting conditions to make the dataset more applicable for practical use. All images are the same resolution of 1280 x 720.

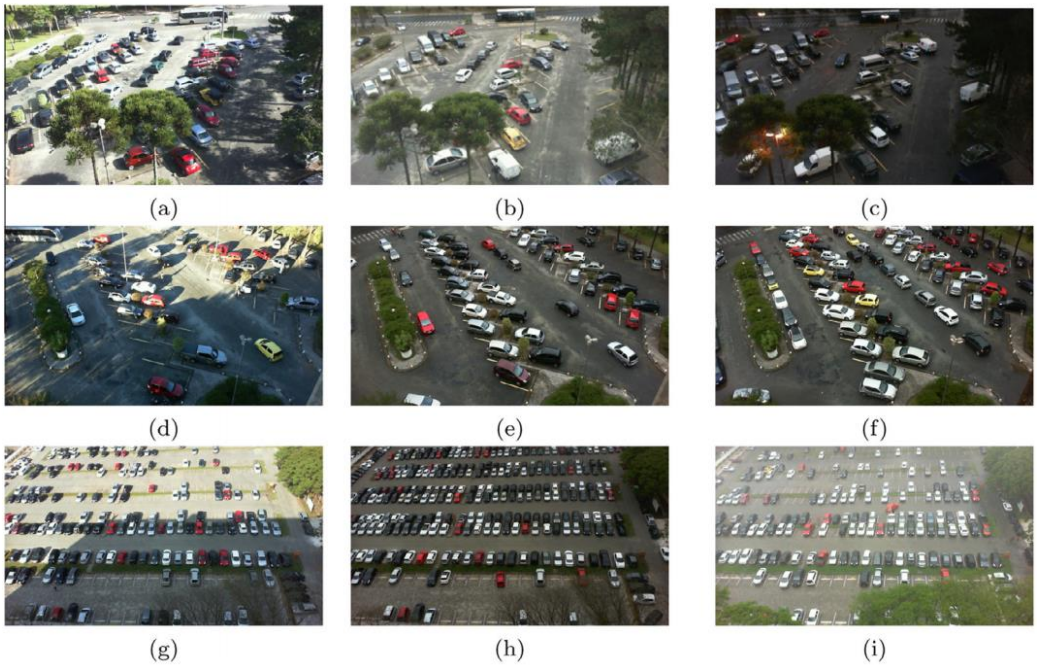

Figure 3. Samples from the PKLot dataset

Further, I experimented with the Red Blood Cell dataset, as I was not getting satisfactory results with the PKLot dataset. This dataset consisted of 299 images of red blood cells under a microscope. The images were of 680x480 resolution, and had largely uniform lighting conditions. It is also important to note that some cells are purple in appearance. These are oxidized blood cells, and still are classified as red blood cells in this instance.

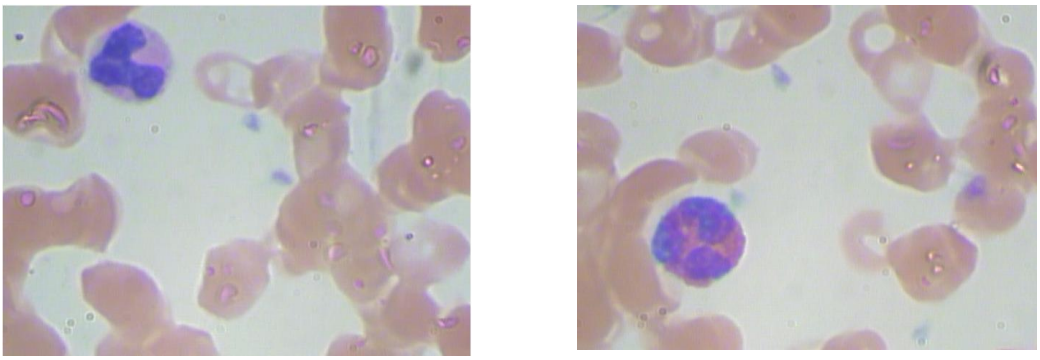

Figure 4. Samples from the RBC dataset

Image Classification

BASELINE ARCHITECTURE

FRUITS 360 ALEXNET

The initial framework I used as a baseline was the generic AlexNet[4] implementation. My impementation of the AlexNet Architecture was simple. I constructed the model according to the description in the original AlexNet paper (Krizhevsky, Sutskever and Hinton, 2017).

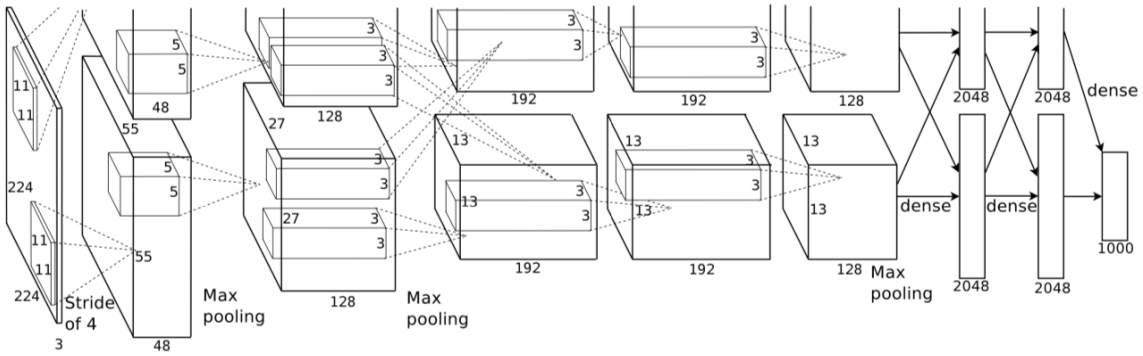

Figure 5. Alexnet Architecture

```
model = tf.keras.models.Sequential([
    #Conv_1         #original model was built for input shape of 224X224
    tf.keras.layers.Conv2D(96, (11,11),strides=4, padding='valid', activation='relu', input_shape=(224, 224, 3)),
    # Pooling_1
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    # Batch Normalisation_1
    tf.keras.layers.BatchNormalization(),
    # Conv_2
    tf.keras.layers.Conv2D(256, (11,11),strides=1, padding='valid', activation='relu'),
    # Pooling_2
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    #Batch Normalisation_2
    tf.keras.layers.BatchNormalization(),
    # Conv_3
    tf.keras.layers.Conv2D(384, (3,3),strides=1, padding='valid', activation='relu'),
    # Batch Normalisation_3
    tf.keras.layers.BatchNormalization(),
    # Conv_4
    tf.keras.layers.Conv2D(384, (3,3),strides=1, padding='valid', activation='relu'),
    # Batch Normalisation_3
    tf.keras.layers.BatchNormalization(),
    #conv_5
    tf.keras.layers.Conv2D(256, (3,3),strides=1, padding='valid', activation='relu'),
    #pooling_3
    tf.keras.layers.MaxPooling2D((2, 2), strides=(2,2),padding='valid'),
    #Batch Normalization_4
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Flatten(),
    #Dense layer_1
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),
    #Dense layer_2
    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),
    #Dense layer_3
    tf.keras.layers.Dense(1000, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(1, activation='sigmoid', name='predictions')
    ])
```

Figure 6. Code implementation

[4] https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

I believe this model worked so well due to the incorporation of neuron "dropout". It is used in the first two fully connected layers seen in figure X. When dropout is removed, the model experiences substantial overfitting.

The standard way or modelling a neurons output *f* as a function of its input *x* is with Equation 1 or 2 below. In terms of training time, these far underperform nonlinear neurons; also known as Rectified Linear Units (ReLU's)

1. $f(x) = \tanh(x)$
2. $f(x) = (1 + e^{-x})^{-1}$

ReLu functions help the loss converge to its optimal value much quicker. This is very helpful for increasing the model efficiency. Another optimiser that was employed was the ReduceLROnPlateau, which reduced the learning rate by ~50% when the loss gradient approached 0. This helped the model to converge to its minimum loss.

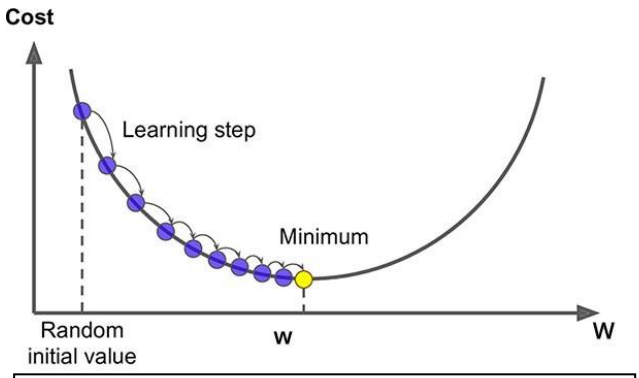

Figure 7. Gradient descent loss minimization

AlexNet Results

I ran this model for 50 epochs on a Nvidia GTX1660s. I plotted the model accuracy, model loss and a confusion matrix locally. I also used tensorboard for some additional visualisations.

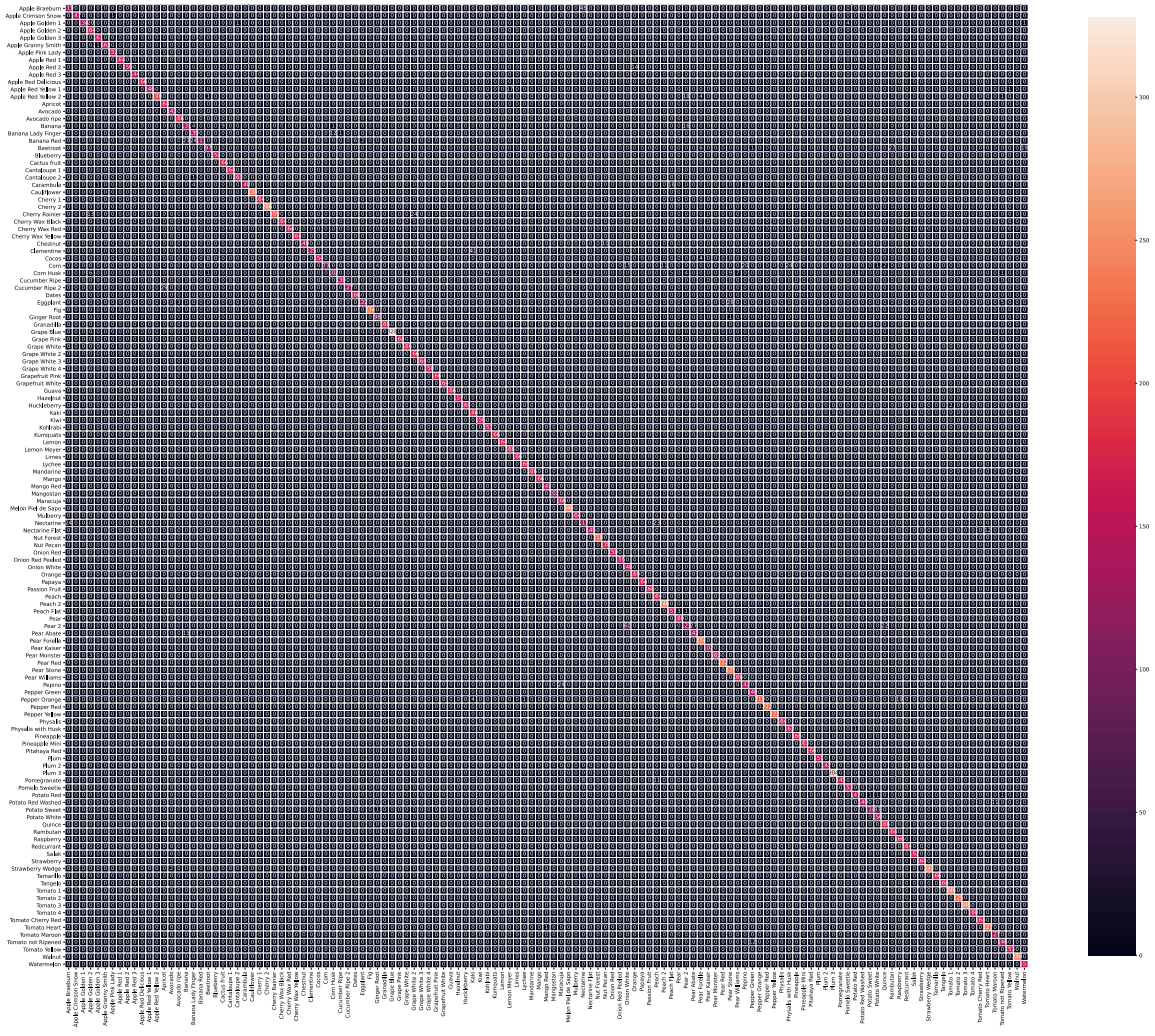

Figure 8. Fruits-365 classification confusion matrix using AlexNet

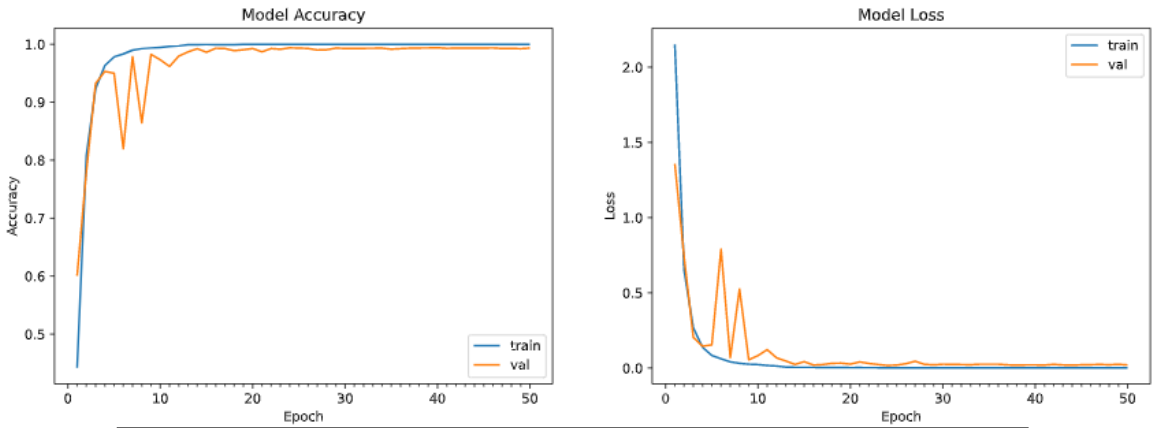

Figure 9. Fruits-365 classification Loss and Accuracy using AlexNet

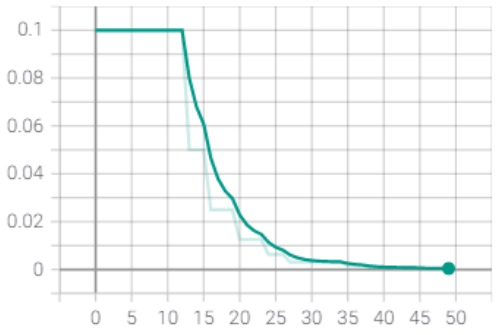

Figure 10. Learning rate to num epochs

CNN	Training	Validation
Loss	0.00177	0.01386
Accuracy	0.9999	0.9948

Table 1. Alexnet classification key statistics

CUSTOM ARCHITECTURE

FRUITS 360 CNN

For my custom architecture, I based my model on the implementation in Mikhail Oltean's paper[5]. The implementation I trained is included below.

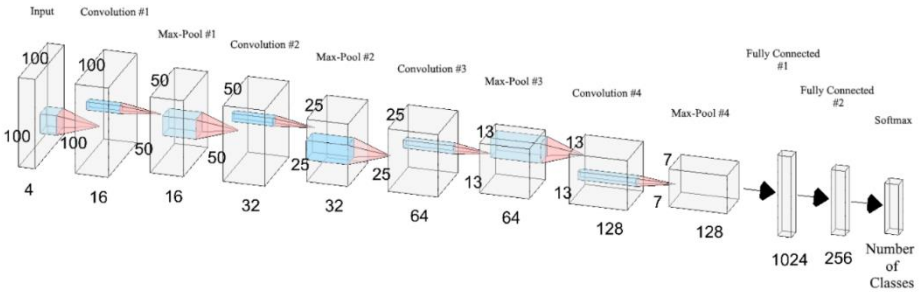

Figure 11. Custom CNN architecture

```
def network(input_shape, num_classes):
    img_input = Input(shape=input_shape, name='data')
    x = Lambda(convert_to_hsv_and_grayscale)(img_input)
    x = Conv2D(16, (5, 5), strides=(1, 1), padding='same', name='conv1')(x)
    x = Activation('relu', name='conv1_relu')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), padding='valid', name='pool1')(x)
    x = Conv2D(32, (5, 5), strides=(1, 1), padding='same', name='conv2')(x)
    x = Activation('relu', name='conv2_relu')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), padding='valid', name='pool2')(x)
    x = Conv2D(64, (5, 5), strides=(1, 1), padding='same', name='conv3')(x)
    x = Activation('relu', name='conv3_relu')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), padding='valid', name='pool3')(x)
    x = Conv2D(128, (5, 5), strides=(1, 1), padding='same', name='conv4')(x)
    x = Activation('relu', name='conv4_relu')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), padding='valid', name='pool4')(x)
    x = Flatten()(x)
    x = Dense(1024, activation='relu', name='fc11')(x)
    x = Dropout(0.2)(x)
    x = Dense(256, activation='relu', name='fc12')(x)
    x = Dropout(0.2)(x)
    out = Dense(num_classes, activation='softmax', name='predictions')(x)
    rez = Model(inputs=img_input, outputs=out)
    return rez
```

Figure 12. Custom CNN implementation

[5] https://mihaioltean.github.io/fruit_recognition_deep_learning.pdf

Custom CNN Results

The results for this architecture were still very good, but unfortunately did not beat the AlexNet implementation. The results are included below

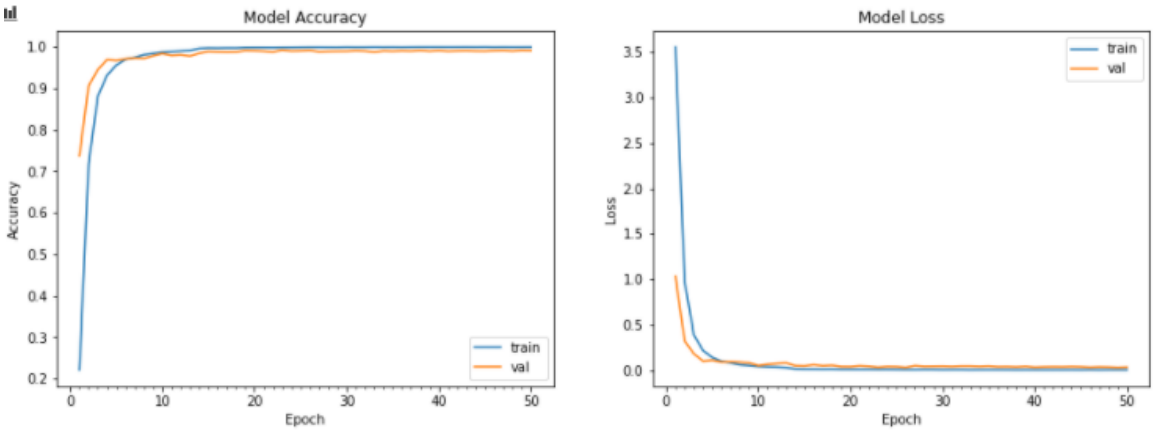

Figure 13. Custom CNN Accuracy and Loss plots

CNN	Training	Validation
Loss	0.0024	0.01366
Accuracy	0.9990	0.9954

Table 2. Custom CNN key statistics

Object Detection

SSD-INCEPTION V2 ARCHITECTURE

For the first object detection challenge, I used SSD-InceptionV2. SSD gets its name 'Single Shot Detector' from the fact that it skips the region proposal step that most other models such as Faster RCNN employ, doing all processing in a 'single shot'. Due to this fact, it is much faster than the other model architectures, and still has good performance.

The SSD training pipeline processes the image where at several convolution levels, several sets of feature maps are generated. In each of these feature maps, a 3x3 convolution filter evaluates a default set of bounding boxes, and gets its IoU value by comparing these maps with the ground truth.

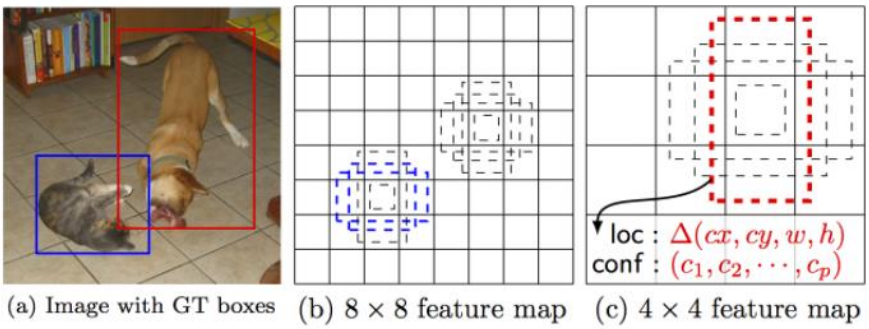

Figure 14. Region Proposal Network in SSD models

A unique feature of SSD is that it is fast, and good at detecting large objects, but not so much small objects. Due to this fact, I used SSD Inception V2 with the Red Blood Cell dataset, and Faster RCNN with the PKLot dataset.

Red Blood Cell Results

This model performed quite poorly, and I was unfortunately unable to optimize it easily, as I was using transfer learning. The model was able to recognize RBC's when they are isolated, and have a clear circular form, but not when they are connected together in a chain.

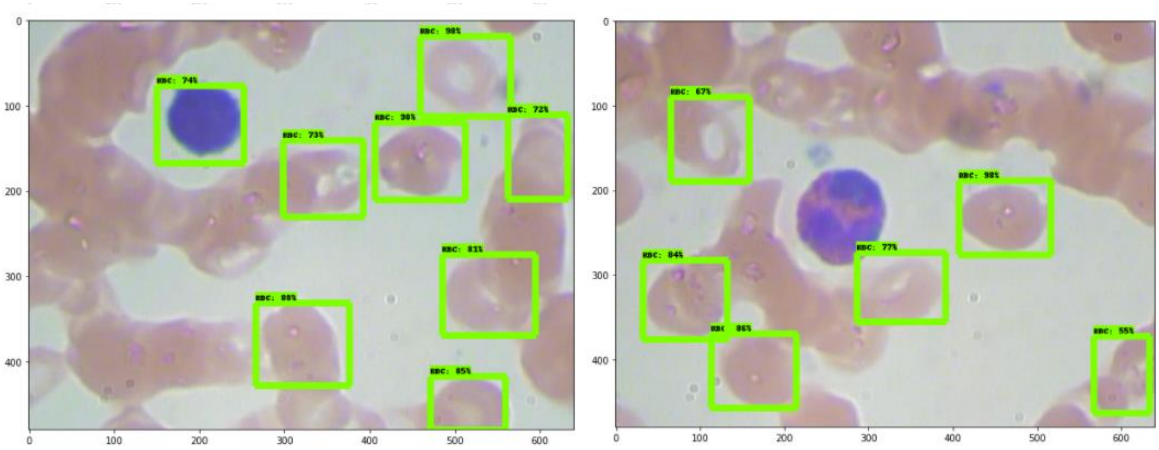

Figure 15. Sample SSD output on RBC dataset

Figure 16. Loss for SSD implementation

	Training	Validation
Loss	2.23	2.49
Average Precision @ IoU 0.5	0.685	0.354

Table 3. SSD model key statistics

FASTER RCNN ARCHITECTURE

The Faster RCNN architecture works by generating a number of 'anchors' in the input image. It then uses a Region Proposal Network (RPN) to select boxes in areas that look like an object (i.e do not look like background). Note that no classification has occurred at this stage.

The network then uses these RPN's it goes through a process called Region of Interest Pooling, which essentially crops each RPN instance and allows us to pass it through the final stages of the pre trained model.

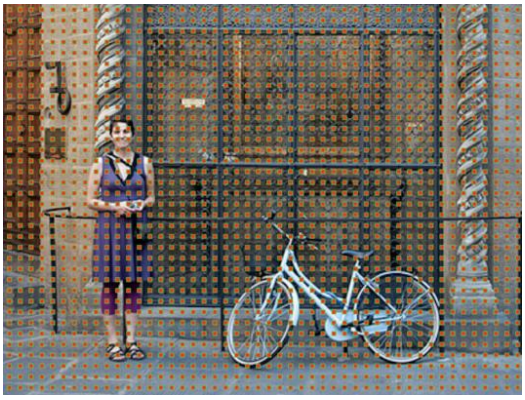

Figure 17. RPN illustration

When the feature mapped image gets to the Fully Connected (FC) layers, the Region-Based Convolutional Neural Network (R-CNN) takes the feature map for each proposal, flattens it and uses two FC layers of size 4096 with ReLU activation. These FC layers then give an output of the bounding box co-ordinates and the class probabilities.

For evaluation, the model compares it to the ground truth bounding boxes, and gets a IoU value. From there, the model attempts to maximise the IoU, and therefore minimize the loss.

The loss that is typically used in this implementation is Binary Cross-entropy. Cross entropy is a way of measuring how far a measured value $\hat{p}$ is from the ground truth p.

Faster R-CNN Results

The Faster RCNN model performed slightly better than the SSD model in terms of loss/accuracy, but as you can see from the output, was still quite poor at detecting many objects of interest.

Figure 18. Sample output using F-RCNN on the PKLot dataset

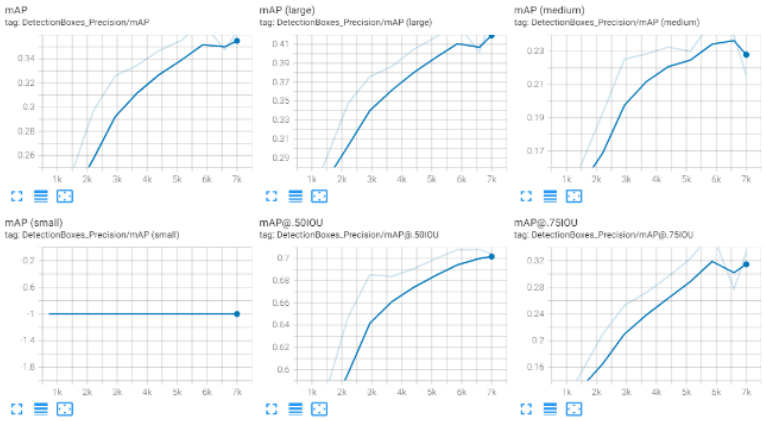

Figure 19. Average Precision F-RCNN model

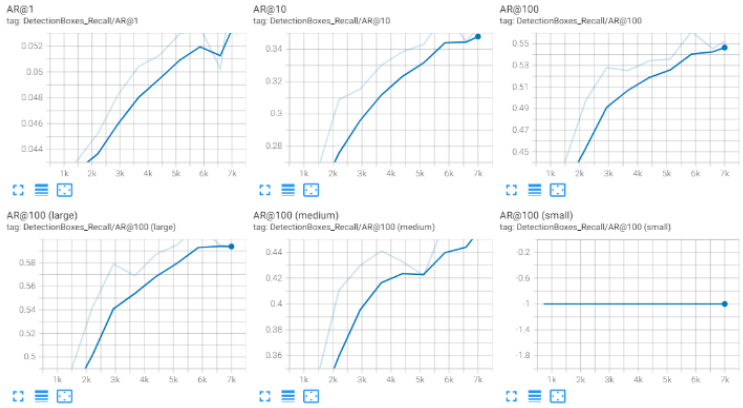

Figure 20. Average Recall F-RCNN model

Table 4. F-RCNN key statistics

	Training	Validation
Loss	2.13	2.32
Average Precision @ IoU 0.5	0.668	0.458

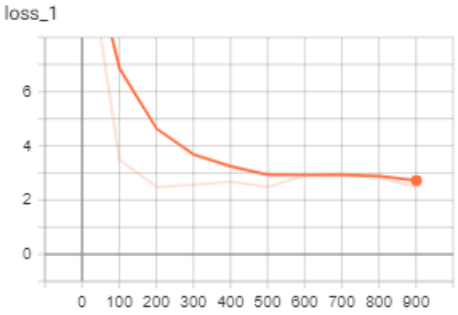

Figure 21. Loss function for F-RCNN model

Conclusion

In conclusion, in the image classification task, both my models performed very well, with both the base model and the customized architecture scoring with a 99%+ accuracy on the test set. I am satisfied with the results of this section.

On the other hand, for the Object Detection task, I had a lot of issues using the Tensorflow Object detection API. I did manage to run the SSD model with some modifications to the tutorial files. Firstly I needed to modify the xml_to_csv.py file to get the bounding box features read into the csv file. I then converted the csv to a TFRecord and was able to feed it into the API.

I was very limited by Google Colab due to the fact that the execution times are very slow and there are many dependencies such as network reliance, GPU availability and drive storage space. When I tried to get the API running on my computer using the tutorial materials, I was faced with Tensorflow v2 compatibility issues. I tried many ways to get the API running locally on my machine, and came very close to being able to train my model locally by using the

```
"""
classes_names = []
xml_list = []
for xml_file in glob.glob(path + "/*.xml"):
    tree = ET.parse(xml_file)
    root = tree.getroot()
    for member in root.findall("object"):
        classes_names.append(member[0].text)
        value = (
            root.find("filename").text,
            int(root.find("size")[0].text),
            int(root.find("size")[1].text),
            member[0].text,
            int(member[5][0].text),
            int(member[5][1].text),
            int(member[5][2].text),
            int(member[5][3].text),
        )
        xml_list.append(value)
column_name = [
    "filename",
    "width",
    "height",
    "class",
    "xmin",
    "xmax",
    "ymin",
    "ymax",
]
```

Figure 22. xml_to_csv.py snippet

TF1 docker image within the object detection API, but I was facing an unknown error when I got to the stage of training, and it would initialize, but never run.

It is due to these my results regarding the object detection section are not as comprehensive as I wanted them to be.

Bibliography

Krizhevsky, A., Sutskever, I. and Hinton, G., 2017. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6), pp.84-90.

G.E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R.R. Salakhutdinov, 2012. Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580, 2012