Data and Visualisation 22/23

Assignment

David Mihalcea - 391326

Date: 9th January 2023

Contents

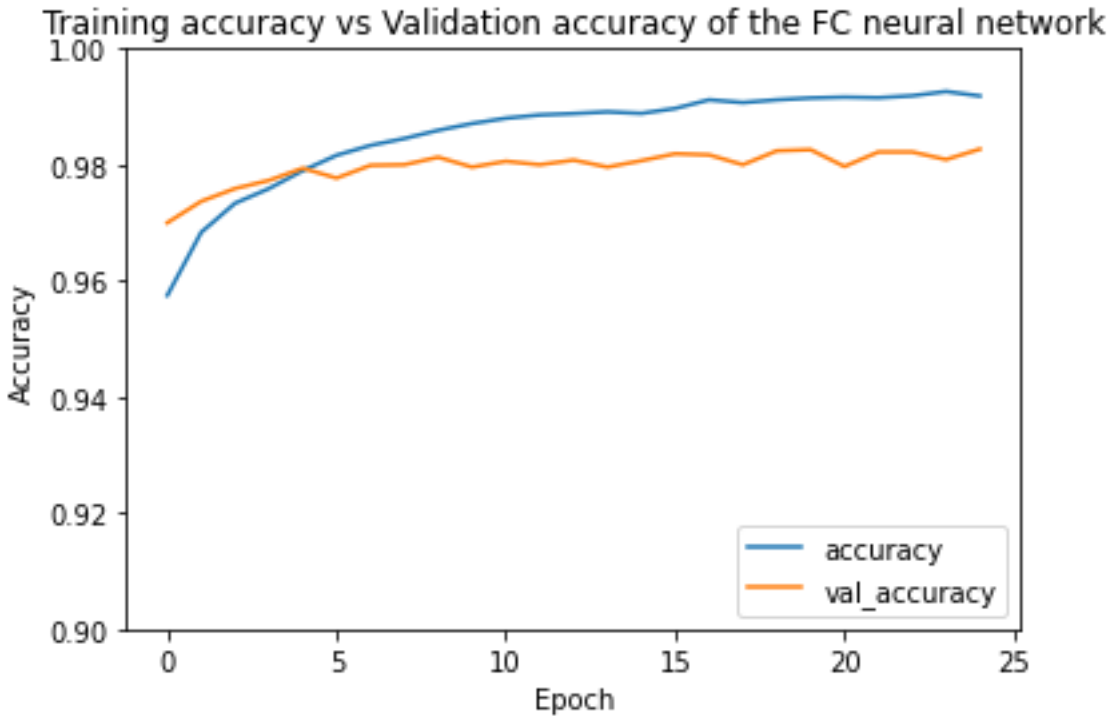
1 Task 1

1.1 Fully connected neural networks

1.1.1 Introduction

A FC neural network can have multiple layers and multiple neurons in every layer just like any other neural network, however, to make it an FC all neurons in a layer must be connected to every neuron in the layer before and after it. The base architecture of this neural network tends to be more computationally expensive but provides better accuracy than traditional manual image classification techniques used prior to the development of neural networks.
A FC neural network was used to perform image classification on the MNIST dataset provided by TensorFlow and the accuracy and loss were recorded and graphed.

1.1.2 The results taken from the FC neural network.



Graph 1. The accuracy of the training and validation for the FC neural network presented for 25 epochs.

1.1.3 Brief description of the results.

Overfitting a dataset can occur when the model relates too closely to the training dataset and therefore performs poorly on new, previously unseen data.
The data in graph 1 presents the accuracy of the traig dataset and validation dataset are similar, suggesting the data was not overfitted in the process of applying the FC neural network to it.
The models competency can equally be shown using the loss found in the training and validation data,

After 25 epochs the model managed to reach an accuracy of 98.27% when evaluated.

1.1.4 Brief discussion of the FC neural network architecture and hyperparameters.

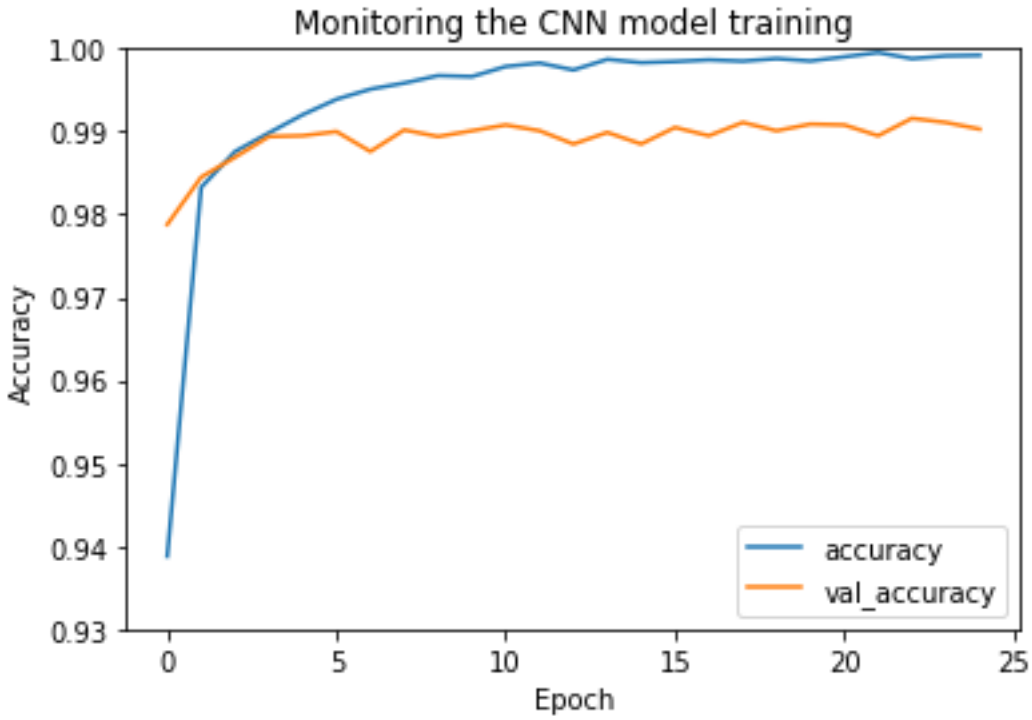
The base architecture was formed of 4 layers and no hidden layers. The initial weights and biases for an FC neural network are 0 because the broad spectrum of tasks the FC neural network is used for requires them to begin as 0.
The first layer of the neural network is a ‘Flatten’ layer because the FC neural network requires the input dataset to be one-dimensional, and the initial dataset was multi-dimensional.
In addition, the next layer was a ‘Dense’ layer which makes the neural network fully connected, and it used 128 neurons in this layer. The activation function used was ReLu which stands for ‘Rectified Linear Unit’. This activation function is used to ensure there are no negative input values because it could prevent the FC neural network to provide the results of best possible accuracy.
Further, a ‘Dropout’ layer is implemented to reduce the effects of overfitting on the dataset by randomly setting some neurons to zero in each forward step, forcing the network of neurons to learn multiple independent representations of the same data.
Lastly, another ‘Dense’ layer is used of 10 neurons before the data is presented at the end.

1.2 Convolutional neural networks (CNN)

1.2.1 Introduction

A CNN is an extension of a FC neural network because they are often formed of fully connected layers as well as convolutional layers and pooling layers. CNNs are specifically designed to handle image classification making them more tuned to be used in image classification.

1.2.2 The results taken from the CNN.



Graph 2. The accuracy of the training and validation dataset plotted on the same graph to be easily compared over 25 epochs.

1.2.3 Brief description of the results.

The clear concern is the almost 5% increase in training data accuracy from epoch 1 to epoch 2 which could potentially be a sign of overfitting in the data, however, because the validation data accuracy is also improving it suggests there is no overfitting of the model taking place and the CNN model is simply learning the data's patterns of the features and textures. The rule of thumb for overfitting is generally 10-15% in the difference between the training and validation error, however this case it is general close to 1% when the two begin to plateau after the 11th epoch.
After 25 epochs the accuracy of the model levels off and the evaluation of the model states the accuracy of the model is 99.03%.

1.2.4 Brief discussion of the CNN architecture and hyperparameters

Convolutional layers are comprised of filters also known as kernels which are convolved with the input image of the layer. These filters are learned by the CNN during the training process which are used to identify features of the input image such as edges and patterns as well as other features and these features are mapped out in the output of the layer which summarises where these features are typically seen. The activation functions in these layers are used to introduce non-linearity to allow the model to identify more complex relationships between the input and out of the layer.

When combining more convolutional layers one after the other it allows the CNN to learn more abstract high level features which are also extracted in turn and put in the output of the layer.

The first layer of this model is a convolutional layer with 32 filters of 3x3 meaning 9 pixels from the image can be scanned for features at a time. Further, the activation used was ReLU because it is very simple and easy to use, it also does not face the vanishing gradient issue other activations such as sigmoid sometimes faces.
The next layer used is MaxPooling2D which down-samples the special dimensions of the input data, in turn reducing the computation and parameters of the input data. This layer achieves this by dividing the input data into smaller squares and choosing the largest value as the output, this case the input data was divided into 2x2 squares.
These two layers often come together and in this case they are done twice each, the second convolutional layer having 64 layers instead.
Further, following those layers they are converted to one dimensional output data as it runs this data through two fully connected layers.

1.3 FCNN vs CNN

CNNs are generally better at preserving special information and local correlations in the input data making them very effective for image and video recognition, which was required for the tasks given above, whereas FCCNs are poor at doing so exposing them to the possibility of overfitting and causing a modelling error. As clearly visible from the results in part 1.1.2 and 1.2.2 the FCNN did not fall at fault of this error and was able to provide good classification accuracy, although still not as good as the CNN.
However, CNNS are very complex due to their many hyperparameters that must be tuned depending on the task in hand and the size of the dataset being used making it difficult to design and train from scratch; FCNNs are simple and has a flexible architecture making them much easier to design and train from scratch; in this case where a database from TensorFlow was used, and predefined models were already designed, the CNN would be the better choice as the performance for the CNN was better than the FCCN as can be seen from the results in 1.1.2 and 1.2.2.

1.4 VGG16 pretrained network applied to the MNIST dataset.

1.4.1 Introduction

VGG16 network is a CNN that consists of 13 convolutional layers, 3 fully connected layers and 5 MaxPooling layers in total. The VGG16 only uses small 3x3 filters because it allows for an increase in the depth of the networks understanding of the dataset while not increasing the number of parameters. The VGG16 is also very easy to implement because it is an application in in Keras so the high-level architecture including the hyperparameters can be used directly from the keras library.

1.4.2 The visualised activation layers.

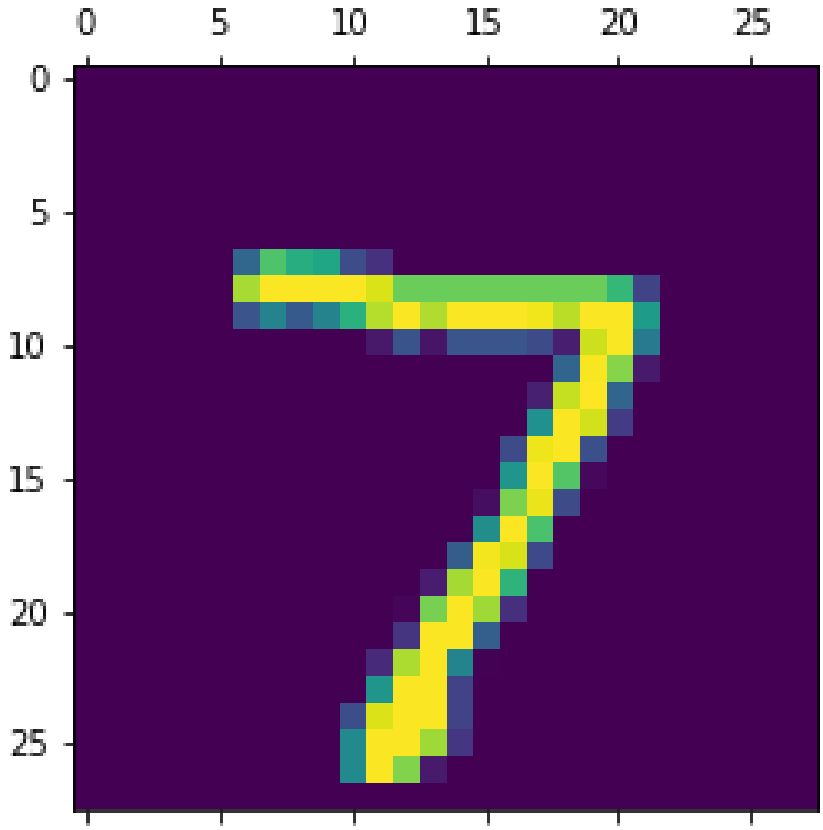


Figure 1. The visualisation of the first layer output in the VGG16 neural network.

Figure 1 presents the feature extraction from the MNIST dataset image previously mentioned that is undergone in a convolution layer of the neural network. It is clear the edges and textures identified by the convolutional layer that outlines the number seen is a 7. The following layers will attempt to better define these edges ad textures until the 7 is clear and can be classified much more accurately. In this case, it is pretty clear, however, in the case of other numbers such as a 4, it may not be as clear in the first layer, which is why there are multiple convolution layers in the VGG16 neural network.

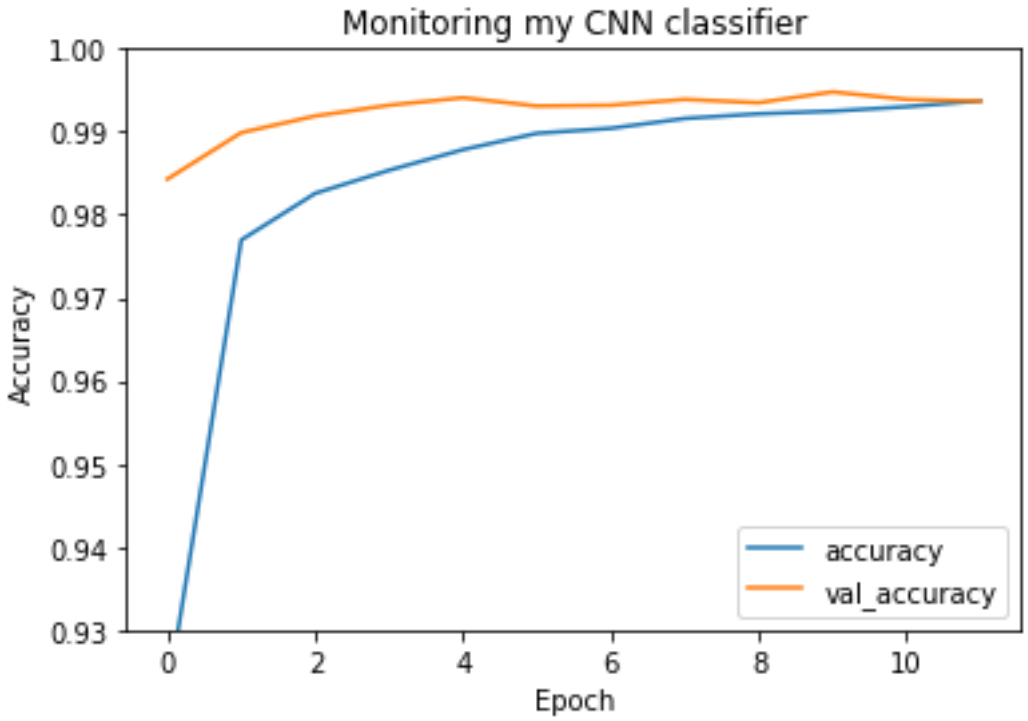
1.5 State-of-the-art classifier

1.5.1 Introduction

Initially, the type of neural network that would be used was chosen to be a CNN given previous research it was clear it would be the better option, however, as the CNN would be difficult to design given the many hyperparameters, an attempt to minimise these parameters was made, in doing so, only 3 convolved layers were used, with filter size 3x3 and a simple and easy activation like the ReLU activation function. This would still allow for effective image classification but reduce the computation and chance of overfitting the model.

Further, the tensor was reduced in size using the MaxPooling2D layer with a pool size of 2x2, this meant that the input data would be divided into 2x2 squares and the largest of the values would be put forward to the output. This was implemented to reduce the computational requirements of the neural network while maintaining the key features previously mapped in the convolution layers.
Then a final convolution layer is implemented with 128 filters in an attempt to maximize the accuracy of the model by learning high level features after already being filtered twice, each time in more detail. This will allow the model to be very familiar with the features required to perform optimal classification.
Furthermore, a 'Dropout' layer was then implemented to reduce the number of neurons by 25% before the data would enter the FC to further reduce the chance of overfitting because the 'Dropout' layer forces the neural network to form multiple independent representations of the same data.
The overfitting factor was a major concern because as previously mentioned overfitting is one of FCs biggest issues.
Further, then a 'Flatten' layer was implemented to reduce the data to a one-dimensional array because the FC dense layers follow, and FC layers require one dimensional arrays.

1.5.2 Results



Graph 3. The training and validation accuracy of the classifier throughout the total amount of iterations.

The data in graph 3 suggests the data has reached a stable state of performance on both the training and validating datasets, meaning the model has not been overfitted and can perform well on previously unseen data.
The accuracy of the model when evaluated on the test data set was 99.37% meaning it scored higher than the accuracy of the first ever CNN done by LaNet in 1998.

Although the accuracy is very good the model requires a lot of computational time, perhaps not making this model very efficient.

2 Task 2

2.1 Plotting bounding boxes.

2.1.1 The 3x4 subplot of the provided labels onto the images.

Figure 2. The 3x4 subplot of the images with their labels plotted on top.

2.1.2 Brief explanation on pre-processing of provided data.

Initially, the format of the labels provided had to be adjusted to fit the requirements of the ‘cv2.rectangle’ as well as removing unnecessary data such as the ‘0’ coordinate that did not change throughout the datasets. In doing so, a ‘for’ loop was used to scan through every file containing these coordinates and removing them column by column.
Then these coordinates were placed in a separate file in csv format, converting the coordinates from strings into floats. Then once these coordinates were now considered floats, they had to be converted into integers, as they are required for the specified library, cv2.

Further, to do so the coordinates were rounded off to the nearest whole number, to reduce the change in the bounding box's location. Then once the pre-processing was done, the coordinates were ready to be plotted.
A similar 'for' loop as previously mentioned was used to scan through every folder and extract each image. Once this was completed plotting the bounding boxes was done using another for loop that assigned each set of coordinates to the given image and the bounding box was plotted.
It is worth noting the 'labels' provided were inconsistent with the location of the drones.
The labels are on the x-axis of each image, so it is easy to check the values were not mistakenly plotted on the wrong image.

2.1.3 Brief explanation of how the data was split and allocated.

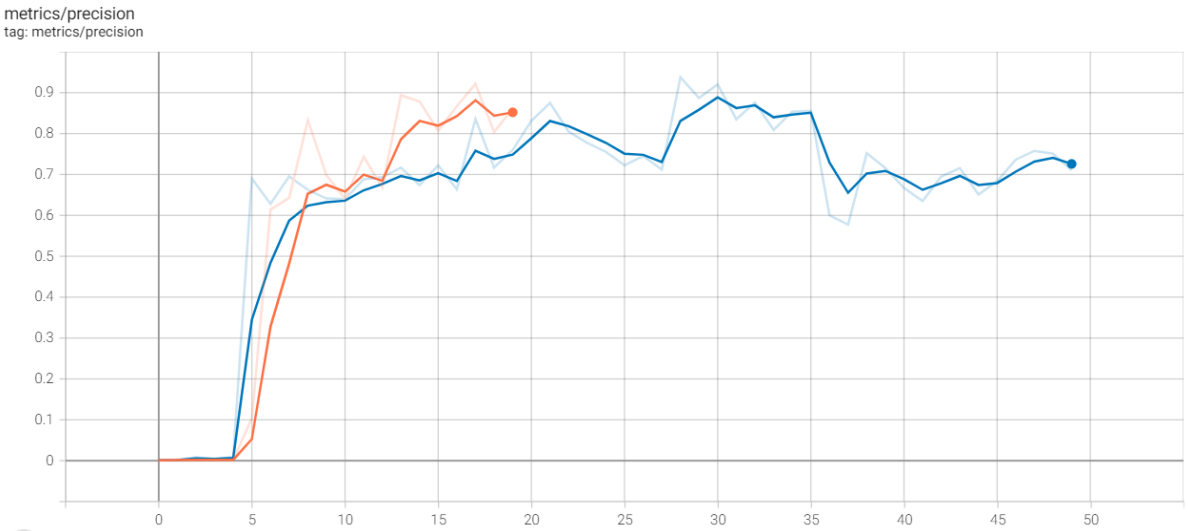
Furthermore, when pre-processing the images, it became clear that there were online pre-processing tools specifically designed for pre-processing images that would undergo object detection and image classification.
Using the tool Roboflow, it was simple to input all the images from the folder, and re-do all the labels, so it would fit the UAV's perfectly in the bounding box.
This tool also automatically split up the images and labels into the 3 categories, training, validating, and testing and it used a pre-defined rule for the percentage split: 70% of images went to training, 20% was dedicated to validating and 10% was left for testing. In using Roboflow, it also made the labels and images easier to export and use, because the data was arranged well in separate files within the zip export file while also offering an alternative option that allows for direct download of the files in Google Colab.

2.2 YOLOv5 model.

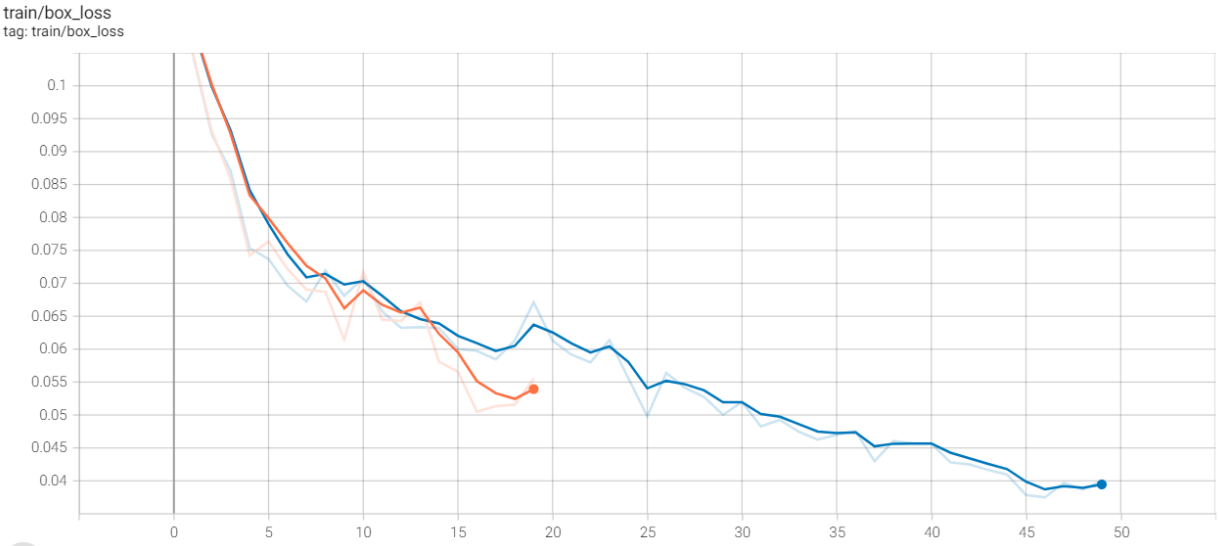
2.2.1 YOLOv5 finetuning process

The YOLOv5 pre-trained network was initially designed to detect up to 90 different objects, but neither of those were UAV.
Firstly, using the Roboflow URL option for all data including images and labels were imported into google collab, separated into their individual categories previously mentioned in part 1.6.3.
Then the YOLOv5 documentation and code from GitHub was cloned into the google collab so all data was available.
Then some adjustments to the documents from the GitHub repository were made, such as the COCO128.yaml folder was adjusted so the only classes it would check for is a person and the UAV, although the focus was the UAV.
The YOLOv5 was fine tuned to the UAV dataset using the transfer learning method. This means that initially the same weights are used as would have been seen in the previous dataset, but once the new UAV dataset is used to train the model, the weights will change automatically to fit the new data set, then once this was done, the weights and biases now found using the dataset can be frozen, to adjust other parameters such as epochs, batch sizes etc.

2.2.2 The results for YOLOv5 on the UAV datasets.



Graph 4. The graph representing the change in the precision of the refined YOLOv5 model with respect to epochs. The graph has been smoothed over, but the original graph can be seen as the faint lines in the background. There are two different results in this graph, where the orange line represents the initial results while the weights were still those of the previous, non-UAV dataset, and only run for just under 20 epochs, and the blue line represents the new weights applied to the model and 50 epochs instead of 20.



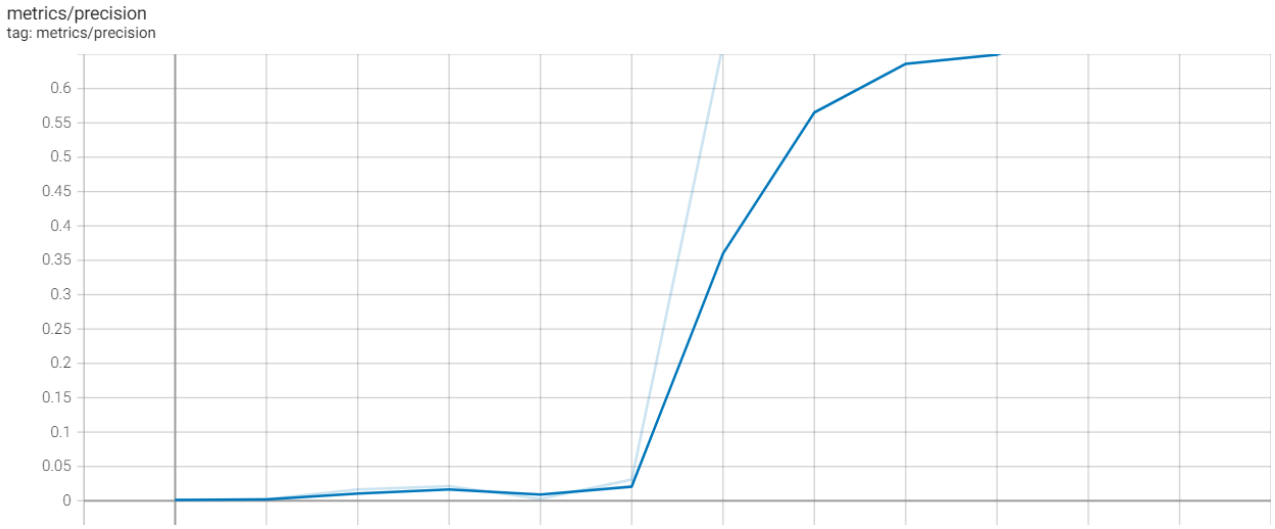
Graph 5. The box loss during the training process which presents how well the model was able to predict the location of the object in the presented image. The faint lines seen both blue and orange are the unsmoothed lines from the results.

As clear from graph 4, the optimal epochs for the YOLOv5 is 30 because the precision peaks at 30 epochs, however, the box loss is minimal at approximately 47 epochs, meaning there has to be a trade off between the two. Therefore, the optimal would be 30 epochs because the precision of the model is much more important than the loss.

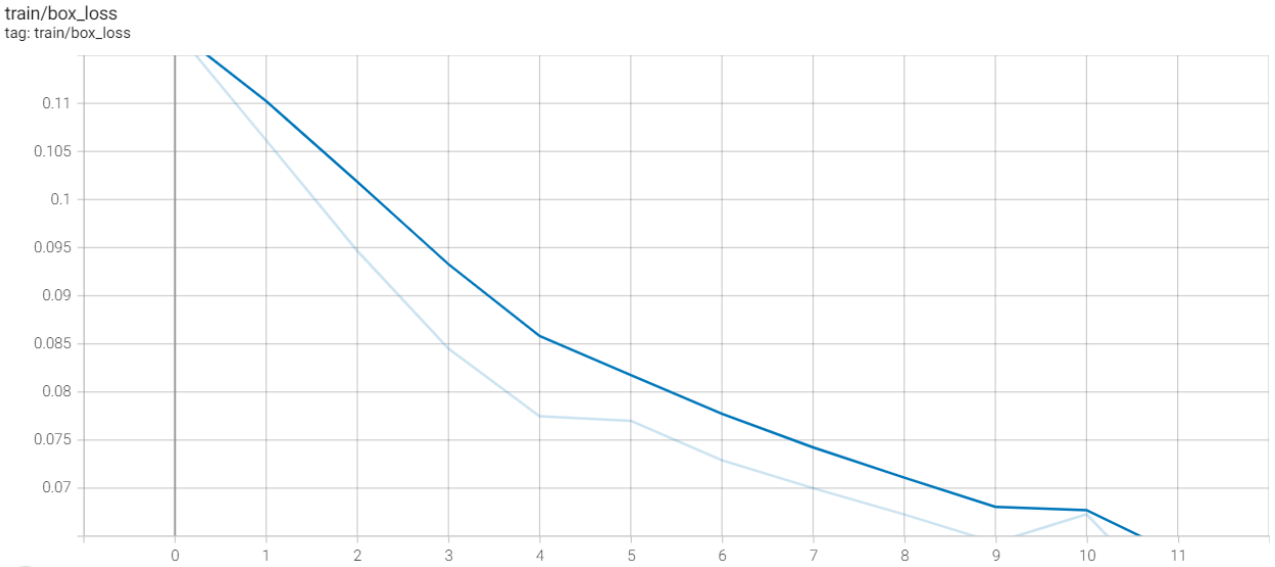
2.3 The results for the RCNN model on the UAV dataset.

The faster RCCN model is much slower than the previously seen one, making running the model very computationally expensive and with very limited GPU it became difficult to run for many epochs. A similar process of refining to that seen in the YOLO were done.

2.3.1 Results



Graph 6. The representing the development of the precision in the faster RCNN network. Given the provided annotations were not great and provided me with even smaller precision results, new pre-processed annotations were implemented and managed to bring the precision metric to plateau at around 65% which is better than previous results. The total number of epochs used were 12, due to very high processing time as the GPU had limited accessibility.



Graph 7. The representation of the development of the box loss during the training process which presents how well the model was able to predict the location of the object in the presented image.

2.4 Comparing the YOLOv5 and Faster RCNN model.

It is very limited what can be directly compared between the two models because the YOLOv5 model outperformed the faster RCNN model because it works faster and therefore it is less computationally expensive than the faster RCNN model making it more convenient to run for more epochs and larger batch sizes.
The YOLOv5 even after only 12 epochs had reached a 70% precision as can be seen from graph 4 meaning it was able to reach higher precision of prediction faster than the RCNN.
Furthermore, the box loss, for the RCNN however seems to fall at a quicker rate than the YOLOv5, although using the same dataset the faster RCNN performed slightly better than the YOLOv5.
The possible reason for RCNN having a quicker reduction in box loss than YOLOv5 is because it is a two stage model because it first generates a region proposal before providing bounding boxes, whereas, the YOLOv5 is a one stage model which reducing the computational time and power required but takes slightly more epochs to achieve the same reduction in box loss.
Lastly, it is easier to fine-tune the Faster RCNN than the YOLOv5 because it has a more modular architecture, however, it's for the same reason that Faster RCNN has a much longer training time than YOLOv5 because it makes the architecture more complex.
In conclusion, the YOLOv5 would be the better option in this case because it is less computationally expensive while also providing a better precision, also, the quicker training time makes it easier to adjust and play with the hyperparameters in order to better the performance of the model on any dataset.

3 References

1. Parashar, A. (2020, October 27). *Vgg 16 Architecture, Implementation and Practical Use*. Medium. https://medium.com/pythoneers/vgg-16-architecture implementation-and-practical-use-e0fef1d14557
2. Mahajan, P. (2020, October 23). *Fully Connected vs Convolutional Neural Networks*. Medium. https://medium.com/swlh/fully-connected-vs convolutionalneural-networks-813ca7bc6ee5
3. Darma, I. W. A. S., Suciati, N., & Siahaan, D. (2021, November 1). *A Performance Comparison of Balinese Carving Motif Detection and Recognition using YOLOv5 and Mask R-CNN*. IEEE Xplore. https://doi.org/10.1109/ICICoS53627.2021.9651855

4. Mahajan, P. (2020, October 23). *Fully Connected vs Convolutional Neural Networks*. Medium. https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5

4 Appendix

4.1 Code for 1C.

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

# loading the MNIST dataset directly into training and testing.
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# The image dimensions from the dataset.
img_rows, img_cols = 28, 28

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

# convert to float and normalize
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Defining how many classes there are.
num_classes = 10

# converting the  class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
```

```
y_test = keras.utils.to_categorical(y_test, num_classes)

# My own model developed using trial and error.
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))

# compiling the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# training the model
history = model.fit(x_train, y_train, batch_size=128, epochs=12,
verbose=1, validation_data=(x_test, y_test))

# evaluating the model
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

4.2 The code for task 2a.

```
#importing necessary libraries.
import csv
import cv2
import os
import random
# reading all the coordinates from the csv file made by me.
coords = []
with open("/content/drive/MyDrive/Coordinatess.csv") as file:
    reader = csv.reader(file)
    for row in reader:
        coords.append(row)

# reading every image from within the image folder and appending it to
a variable.
images = []
for filename in os.listdir("drive/MyDrive/image"):
    image = cv2.imread(os.path.join("drive/MyDrive/image", filename))
```

```
    images.append(image)
BB = []
for i, image in enumerate(images):
    if i >= len(coords):
        break
    coord = coords[i]
    x, y, w, h = [int(c) if c != "" else 0 for c in coord]
    B = cv2.rectangle(image, (x, y), (x+w, y+h), (0, 0, 255), 2)
    BB.append(B)

# cv2.imshow("image", image)
# cv2.waitKey(0)
#The cv2.imshow is commented because google colab was saying they are
having an issue processing it.

cv2.destroyAllWindows()
#I found another way to show only 12 images in a 3x4 subplot using the
code below.
#The libraries required were imported in a different cell.
plt.figure(figsize=(11,11))
for i in range(12):
    plt.subplot(3,4,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(BB[i])
    plt.grid(True)
    #plt.imshow(coords[i])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(coords[i])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
plt.show()
```