Claire DELGOVE
S359263

# Deep Learning
# 21/22

## Assignment

## Image classification on the MNIST handwritten digits recognition

## UAV Object Detection using Transfer Learning

Claire DELGOVE – S359263

*Number of words ≈ 3000*

**Department**: MSc AAI

**Date**: January 2021

## Contents

1. **Context** 3

2. **Task 1: Image classification on the MNIST handwritten digits recognition** 4

2.1. Objective 4
2.2. Approach 5
2.2.1. *Model A* 5
2.2.2. *Model B* 5
2.3. Results and Evaluation 7
2.3.1. *Model A* 7
2.3.2. *Model B* 9

3. **Task 2: UAV Object Detection using Transfer Learning** 11

3.1. Objective 11
3.2. Approach 12
3.2.1. *Data Pre-processing* 13
3.2.2. *Model* 13
3.2.3. *Transfer Learning* 14
3.2.4. *Settings* 15
3.3. Results and evaluation 17
3.3.1. *Performance* 17
3.3.2. *Validation* 18
3.3.3. *Metric used* 19

4. **Conclusion** 20

5. **Contents of figures** 21

6. **Bibliography** 22

7. **Code** 23

# 1. Context

Over the progress of AI, more and more models have been created from scratch in order create object detectors capable of localize and classify a single or several objects on an image or a video stream. In this project, simple deep learning models will be studied in order to firstly, recognize handwritten digits with the MNIST dataset, secondly, detect an UAV on an image by localization. By localization, it means entailing the UAV to a bounding box which is predicted by regression in this case.  For this, training an object detector from scratch would take several days on a CPU. Then, a pre-trained CNN will be used applying transfer learning to fine-tune the model with the given dataset of UAV pictures.
Over both tasks, different methods will be used to find the best performing model for each one. In each part, the main objective, the method used, the results and evaluation will be discussed.
All codes will be undertaken with Python (Jupyter Notebook for Task 1 and Spyder for Task 2).

# 2. Task 1

## Image Classification on the MNIST handwritten digits recognition

### 2.1 Objective

The MNIST dataset contains handwritten digits and is standardly used in deep learning for digits recognition. Thus, it has also been used as the basis learning for developing a convolutional deep learning neural network for image classification. It contains 60,000 small square 28x28 pixel grayscale images of handwritten single digits between 0 and 9 (*Figure 1*). Thus, the task is to apply classification among 10 classes (0 to 9). At the end, the model must be able to associate a class to a given image according to the digit it represents.

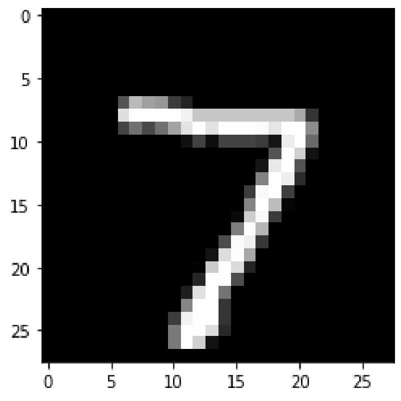


*Figure 1: Image example from the MNIST Dataset*

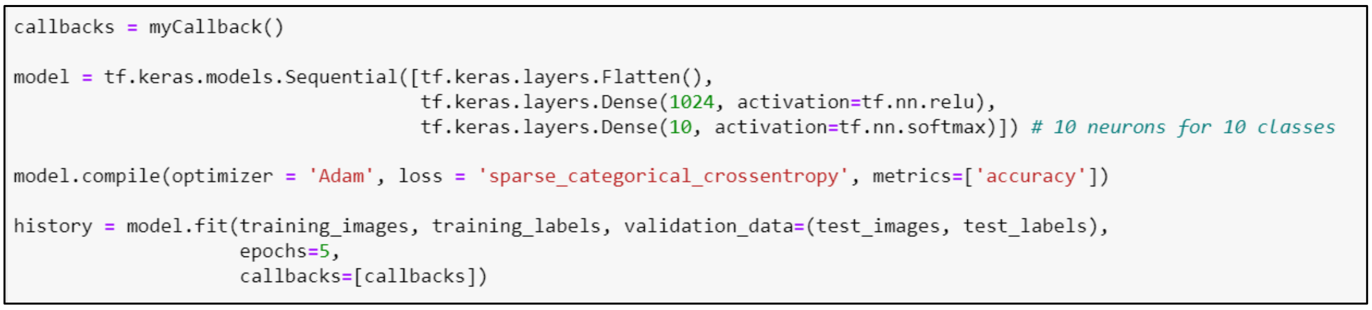
In this task, two models will be trained to develop a CNN for handwritten digit classification.

## 2.2 Approach

The first step is to pre-process the data. The pixel values for each image of the MNIST dataset are integers from 0 to 255. To begin with, it is needed to normalize the data of grayscale images. To do that, the pixel values of each set (training and testing) are rescaled in the range from 0 to 1 (dividing each pixel by 255 which is the maximum value of a pixel).

### Model A

For this model, only the fully connected layers will be recurred. The term Fully Connected translates the fact that each neuron in a layer is connected to the entirety of the neurons in the two adjacent layers. The different steps are visible on the code below (*Figure 2*).



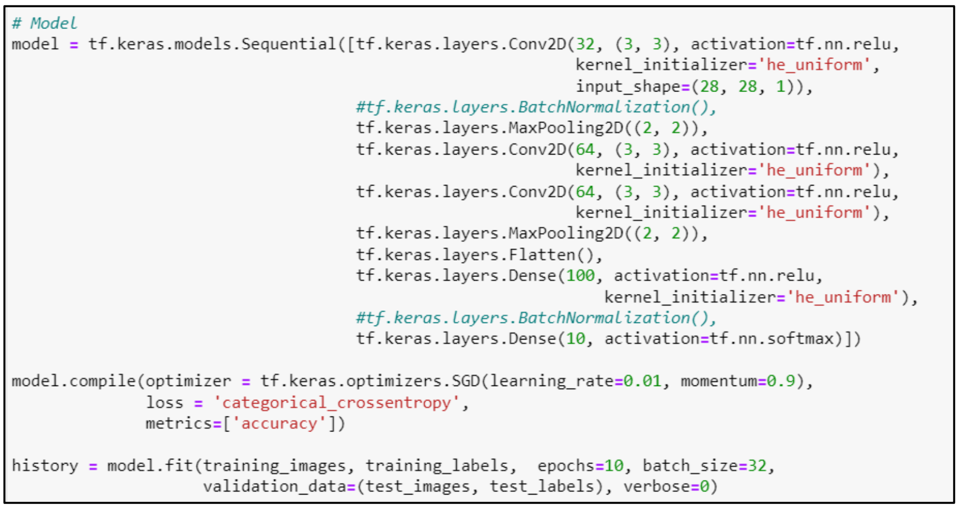
```
callbacks = myCallback()

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(1024, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)]) # 10 neurons for 10 classes

model.compile(optimizer = 'Adam', loss = 'sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(training_images, training_labels, validation_data=(test_images, test_labels),
                    epochs=5,
                    callbacks=[callbacks])
```

*Figure 2: Code detailed for Model A*

First, a sequence of layers is initialized with a *Flatten* layer, and two layers of fully connected neurons. The first layer is activated with the *ReLU* function and 512 neurons to have accurate results.
For the second layer, it is needed to have 10 neurons since it must match with the number of classes (10 in this case). Hence, 10 neurons are in the final layer. This time, the layer is activated with the *Softmax* function to squash and interpret the output directly as a probability. The *Softmax* function is very suitable for multi-class classification problems.
After that, the model is built and compiled with a loss suitable for classification such as *sparse categorical cross entropy* since truth labels are integer encoded.
A *call-back* is set in order to stop the training when the accuracy reaches 99%, and 5 epochs are enough for that.

### Model B

For this model, a pre-trained CNN will be used and fine-tuned for handwritten recognition. The model is organized with at the top the feature extraction to recognize the digit, and the classifier backend to make the prediction. The different steps are visible on the code below (*Figure 3*).



```
# Model
model = tf.keras.models.Sequential([tf.keras.layers.Conv2D(32, (3, 3), activation=tf.nn.relu,
                                                           kernel_initializer='he_uniform',
                                                           input_shape=(28, 28, 1)),
                                    #tf.keras.layers.BatchNormalization(),
                                    tf.keras.layers.MaxPooling2D((2, 2)),
                                    tf.keras.layers.Conv2D(64, (3, 3), activation=tf.nn.relu,
                                                           kernel_initializer='he_uniform'),
                                    tf.keras.layers.Conv2D(64, (3, 3), activation=tf.nn.relu,
                                                           kernel_initializer='he_uniform'),
                                    tf.keras.layers.MaxPooling2D((2, 2)),
                                    tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(100, activation=tf.nn.relu,
                                                              kernel_initializer='he_uniform'),
                                    #tf.keras.layers.BatchNormalization(),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(training_images, training_labels,  epochs=10, batch_size=32,
                    validation_data=(test_images, test_labels), verbose=0)
```

*Figure 3: Code detailed for Model B*

First, a single convolutional layer (*Conv2D*) with 32 filters of size 3X3 is set, followed by a max polling layer. Convolution is the primary processing step of the image. It is divided into zones called tiles. Inside these tiles, each pixel is associated with a weight by the work of an artificial neuron. Each tile has its own artificial neuron, but these all have the same basic parameters. As a result, all tiles are analysed in the same way. In practice, the tiles overlap according to a predefined step. A convolution calculation is then performed within each tile for each feature sought. Then, the various convolutions are stacked, allowing to obtain a feature map which translates the presence and location of the features in the image. The *ReLU* function activation allows to separate negative values in the convolution output. All negative values are transformed into 0. More, it prevents the exponential growth in the computation. The kernel initializer is *he_uniform* to draw samples from a uniform distribution to initialize weights. *Batch normalization* can be used after convolutional and fully connected layers to change the distribution of the output of the layer, standardizing the outputs. This has the effect of stabilizing and accelerating the learning process. Since our task is easy doing for the model, the batch normalization isn't used in this case.

After several convolution steps, a pooling step is performed, which corresponds to the maximum value of each 4-pixel (2x2 square) portion of the layers (*Max-pooling*). This step reduces the dimensions of the processed data by a factor of 2.

To have enough performance, the depth of the feature extractor is increased with more filters, adding more convolutional and pooling layers with the same sized filter (double convolutional layer with 64 filters each, followed by another max polling layer).

Flattening (*Flatten* layer) is the final step in image processing, which involves transforming the stacking of convolution layers into a single vector. It occurs after several convolution cycles. The filter maps are then flattened to provide features to the classifier.

As before, the last *Dense* layer has 10 neurons (for the 10 classes) and is activated with the *Softmax* function for the same reasons as before.

After that, the model is built and compiled with the stochastic gradient descent (*sgd*) optimizer for the cost function helping the model to learn as fast as possible (learning rate of 0.01 and a momentum of 0.9), and the loss *categorical cross entropy* suitable for multi-class classification. The learning rate needs to be not too low to enable the model to learn, and not too high in order to not miss any minimum of the cost function in the gradient descent. More, the momentum is chosen to smooth the progression of the learning algorithm and can accelerate the process. The number of epochs is first fixed to 10 and will be adapted later in the validation if it is needed.

## 2.3 Results and evaluation

### Model A

The accuracy value at the end of the final epoch is **99,16%**, which is strongly good (*Figure 4*). The neural network is about 98% accurate in classifying the training data. Regarding the unseen data, it returns an accuracy of **97,98%** (*Figure 7*).

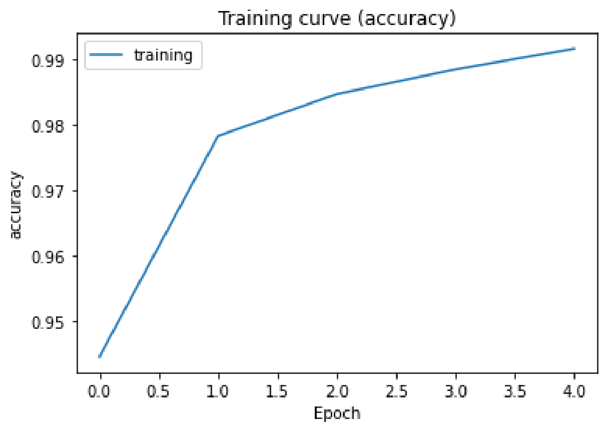


*Figure 4: Training curve for the accuracy (Model A)*

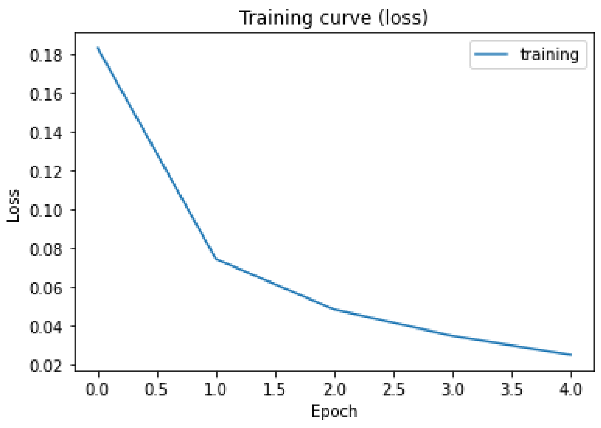


*Figure 5: Training curve for the loss (Model A)*

However, results seem to be good, it is essential to check further the performance of the model regarding a potential overfitting or underfitting. Indeed, the built architecture must be trained properly, that is, not too little, not too much (*Figure 6*). In fact, we use a repetitive training by successive iterations (epochs). At each epoch, the dataset is fully exploited to learn. This method allows you to adjust the characteristics of the model as you go.

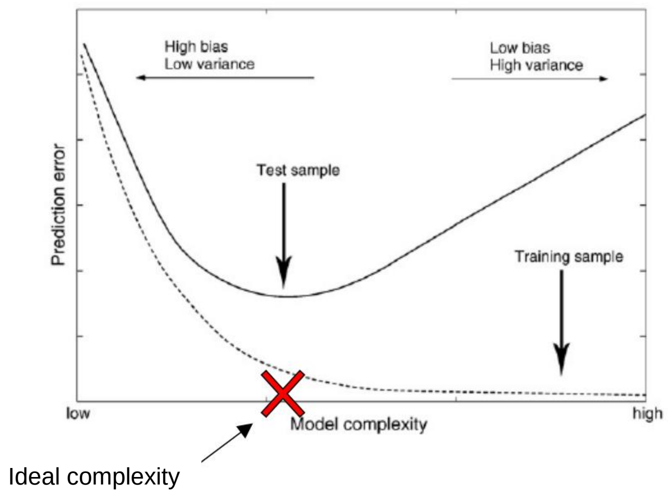


*Figure 6: Visualization of the ideal complexity*

It is important to choose an appropriate number of epochs. If the number of epochs is not large enough, the model will not have had enough time to learn (underfitting), and the prediction will not be optimal, because based on criteria still too vague. Conversely, when the number of epochs is too large, the model will have "too" learned (overfitting), which means that its prediction will be too specific to the dataset used. The difficulty of achieving maximum predictive quality thus lies in determining the optimal learning rate. The number of epochs being a parameter dependent on a large number of factors, it is not possible to predict it theoretically. Thus, performances on the train and validation dataset over each epoch are plotted to visualize learning curves and be sure there is no overfitting or underfitting for the epoch chosen (*Figure 7 and 8*).

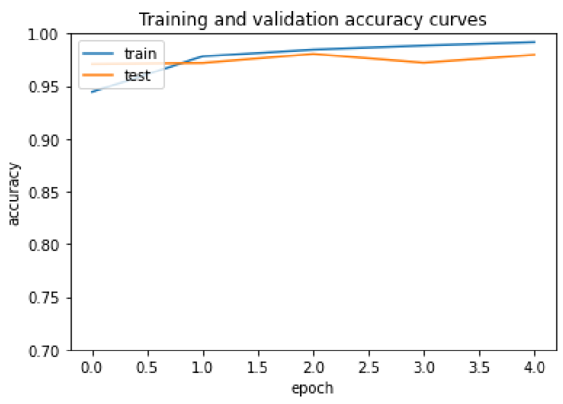


*Figure 7: Learning curves for the accuracy (Model A)*

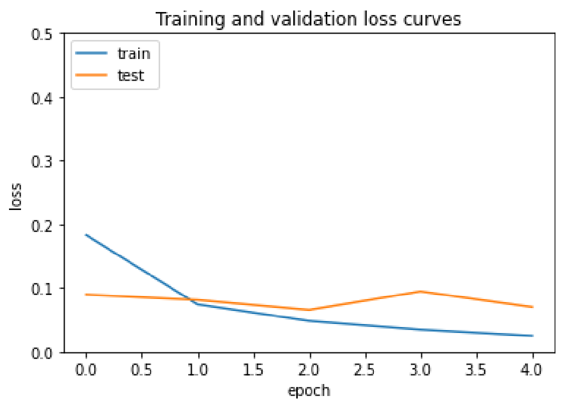


*Figure 8: Learning curves for the loss (Model B)*

After conducting a cross validation, the curves indicate that for the 5 epochs configured there is no overfitting. Indeed, the curves are very close and fluctuate little (0.05 average magnitude of the test curve for the number of epochs greater than 2. It is possible that the optimal number of epochs is greater than 5, but the algorithm learning time is with this value already very substantial, and the results obtained are very good. Therefore, 5 as the number of epochs is the final value selected.

## Model B

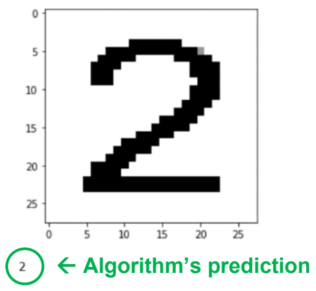
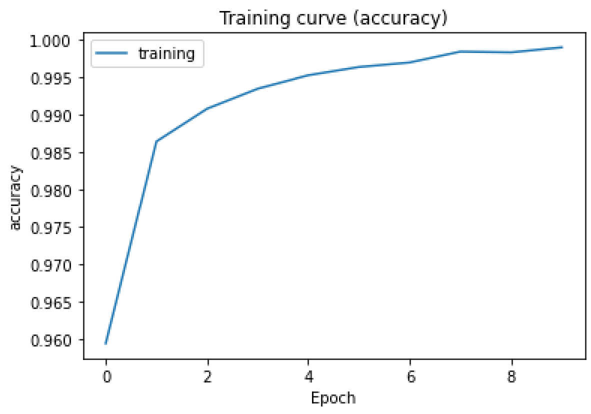


*Figure 9: Prediction example for a digit image picked on the net*

The accuracy value at the end for the unseen data is **99.13%** (*Figure 10*)**,** which is better than Model A. At first sight, this model seems to be able to beat *LeNet* 1998 model (98.7% on the test set).



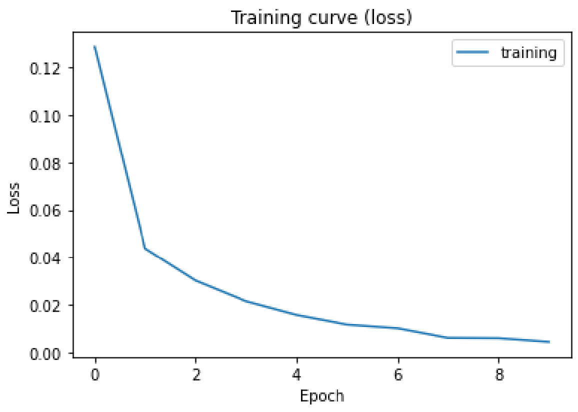
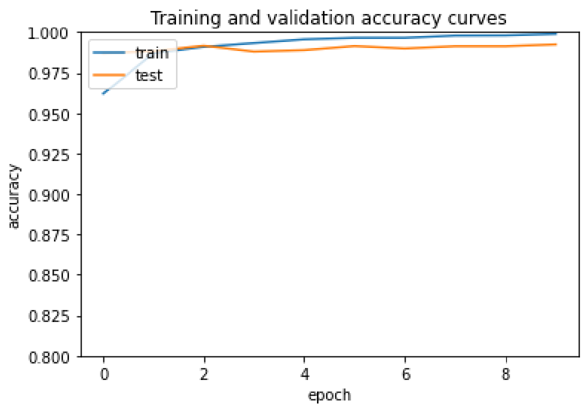


*Figure 10: Training curve for the accuracy (Model B)*

*Figure 11: Training curve for the loss (Model B)*

But in the same way as Model A, learning curves also need to be plotted in order to evaluate further the performances (*Figure 12 and 13*).



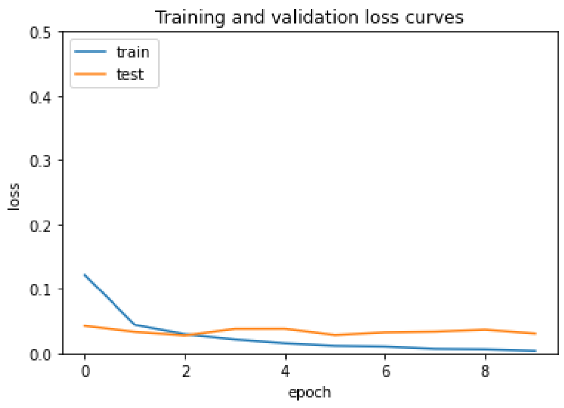


*Figure 12: Learning curves for the accuracy (Model B)*

*Figure 13: Learning curves for the loss (Model B)*

After conducting a cross validation, the curves indicate that for the 10 epochs configured there is no overfitting. Indeed, the curves are very close and fluctuate little (0.0005 average magnitude of the test curve for the number of epochs greater than 2. It is possible that the optimal number of epochs is greater than 10, but the algorithm learning time is with this value already very substantial, and the results obtained are very good. Therefore, 10 as the number of epochs is the final value selected.

The CNN classifier Model B is a bit more performant since it has more depth that just the fully connected layers of the Model A. For this task, the difference is not so important since the classification is easy doing for the network. However, for more consistent tasks, it would have been unthinkable to use just the fully connected layers for the training. It is in most cases needed to have a deeper neural network. But for this task, some convolutional layers were enough.
More, pre-trained models are often used for different tasks since building them from scratch would take too much time. This kind of practice will be illustrated in Task 2.

# 3. Task 2

## UAV Object Detection using Transfer Learning

### 3.1 Objective

Using a small dataset with images of UAV, the objective is to build an object detector able to localize an UAV on a picture. It is done by predicted the bounding boxes by regression to detect an UAV. Since it would take too much time to train a neural network from scratch for this task, a pre-trained detector is used and to apply transfer learning on it with the dataset provided (*Figure 14*). The aim is to train this detector in a way to achieve the best performance. In this regard, techniques and hyperparameters choices will be explained.

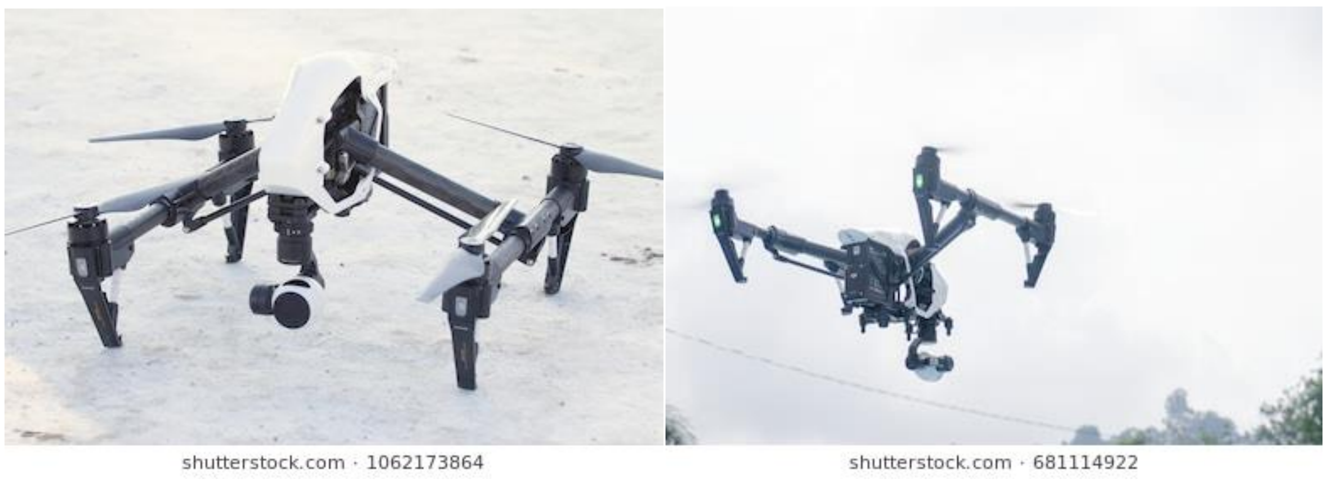


*Figure 14: Image examples from the given dataset*

## 3.2 Approach

### Data pre-processing

The dataset provided contains 314 images of drones with ground truth labels for the bounding boxes (4 coordinates) associated with each image (*Figure 15)*.
To begin with, data are cleaned and prepared to be used efficiently by the future neural network. Filenames from both images and text files containing the ground truth bounding boxes are linked. Therefore, images with ground truth boxes can be visualized to check if the data have been imported correctly.

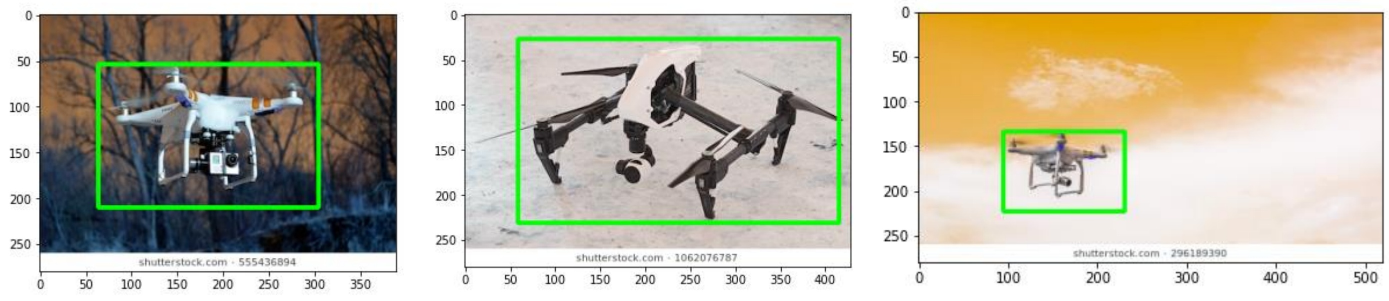


*Figure 15: Image examples with ground truth bounding boxes plotted*

The future model selected can take into entry data having the same size. However, raw pictures have different sizes. Consequently, images are loaded and transform into array with the same target size (224X224). In this regard, ground truth bounding boxes need to be adapted (*Figure 16*). They are normalized to fit with the images of size 224X224.

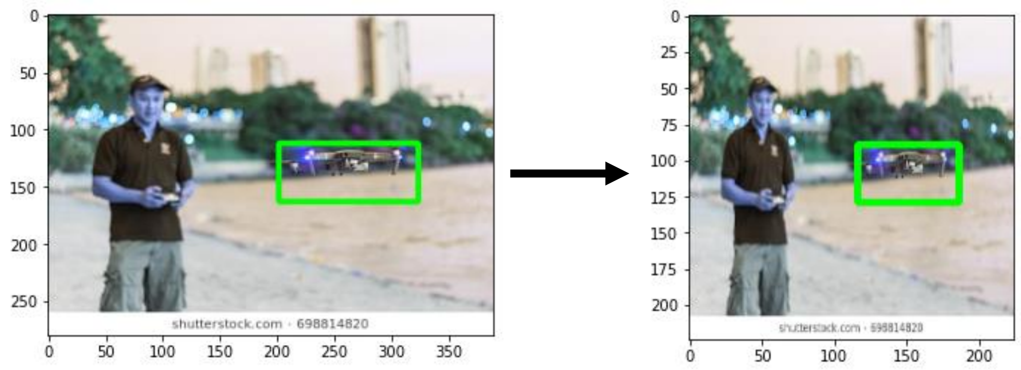


*Figure 16: Image resize visualization*

After that, data are normalized to a range from 0 to 1 to be adequate for the training, before being split into a training and a testing set.

## Model

A CNN contains some convolutional, pooling and fully connected layers (number depending on the amount of data). The last layer is the output which gives a probability of an object detection on an image (as much as the number of possible classes) in a classification problem. In an image localization algorithm, it is the same except the output layer. In classification algorithms, the final layer gives a probability value ranging from 0 to 1. In contrast, localization algorithms give an output in four real numbers, as localization is a regression problem which returns numbers instead of classes. Those four values (x, y, height and weight coordinates) are used to draw a box around the object of interest which is the UAV.

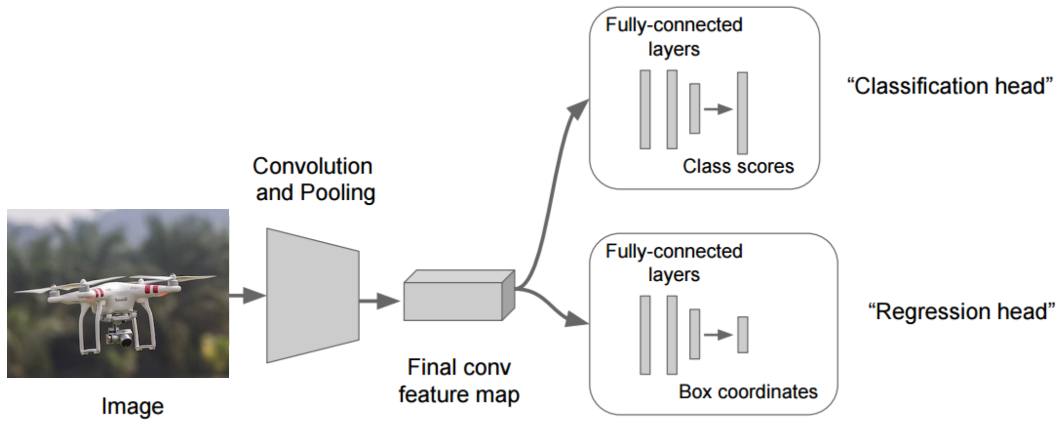


*Figure 17: Pipeline used*

The convolutive neural network model requires training using ground truth images, it means weight modification parameters to learn the key characteristics of the images. These are initialized randomly, then adjusted empirically as a dataset is processed to improve accuracy. The number of parameters being very large, training is very expensive and takes a lot of time. Indeed, the whole issue of a good prediction lies in the ability to model to define useful characteristics to detect in the image. Then, a distance (*Figure 23*) is used to calculate the loss between the predicted bounding box and the ground truth. However, it is not necessary to build a CNN in its entirety to develop a prediction desired. Indeed, the advantage of this type of algorithms is that it is scalable and adaptable for other predictions. The Transfer Learning, which corresponds to this transposition of the algorithm for a new prediction, consists in retaining the induced convolution layers and replacing the fully connected prediction at network output (*Figure 18*).

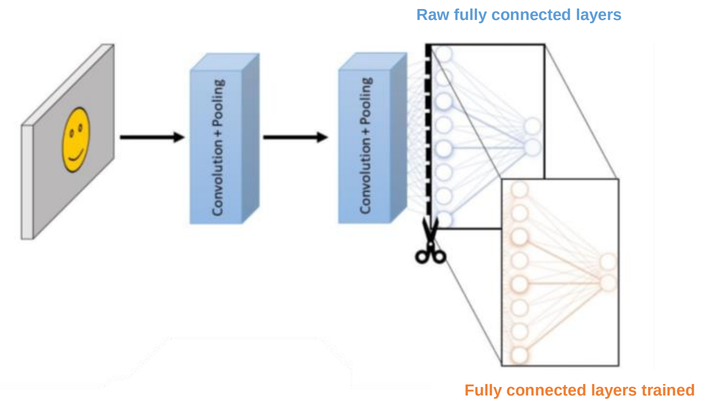


*Figure 18: Transfer Learning principle*

Since the dataset is quite small with only 314 pictures, it would not be adequate to build a model from scratch. Consequently, transfer learning is applied with the pre-trained model ***VGG16***, which are widely used and recognized for their good performance.

## Transfer Learning

***VGG16*** is a version of the convolutive neural network VGG-Net. This version consists of several layers, including 13 convolution and 3 fully connected, and therefore must learn the weights of 16 layers. It takes input a colour image of size 224x224 pixels and realize a regression in order to predict the four bounding boxes' parameters. It therefore returns a vector of size 4, which contains the localization points for the boxes. (*Figure 19*).

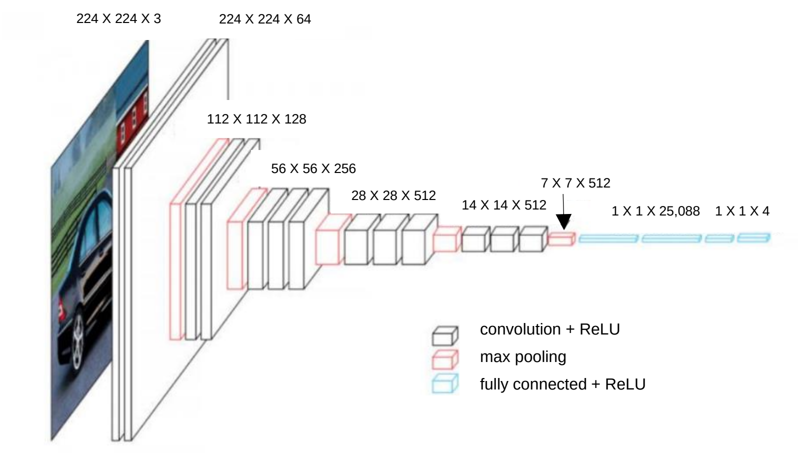


*Figure 19: VGG16 architecture*

To apply transfer learning, the 13 convolution layers are kept (*Figure 20 and 21*), and the fully connected layers are replaced with five new ones (*Figure 21*), returning a vector of size 4, which contains the four parameters for the bounding boxes. For this new network, the training is only done on the five new fully connected layers. This means:

$$\underbrace{(25{,}088*256+256)}_{\textit{1st FC layer}} + \underbrace{(256*128 + 128)}_{\textit{2nd FC layer}} + \underbrace{(128*64+64)}_{\textit{3rd FC layer}} + \underbrace{(64*32+32)}_{\textit{4th FC layer}} + \underbrace{(32*16+16)}_{\textit{Final prediction}} = 6{,}446{,}148 \text{ weights to retrain}$$

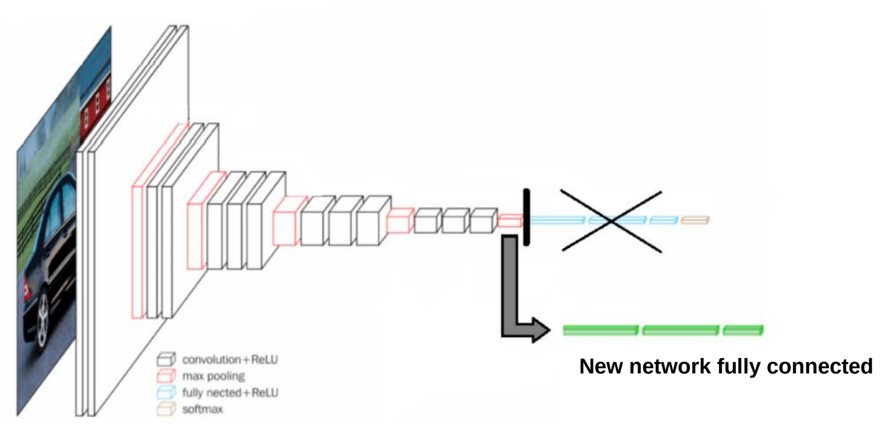
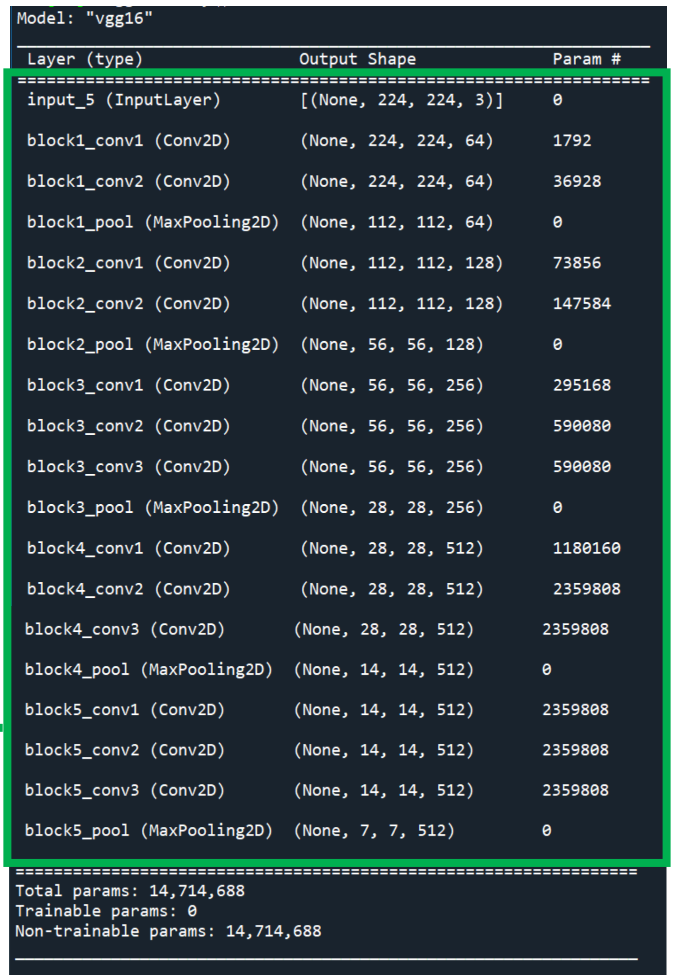
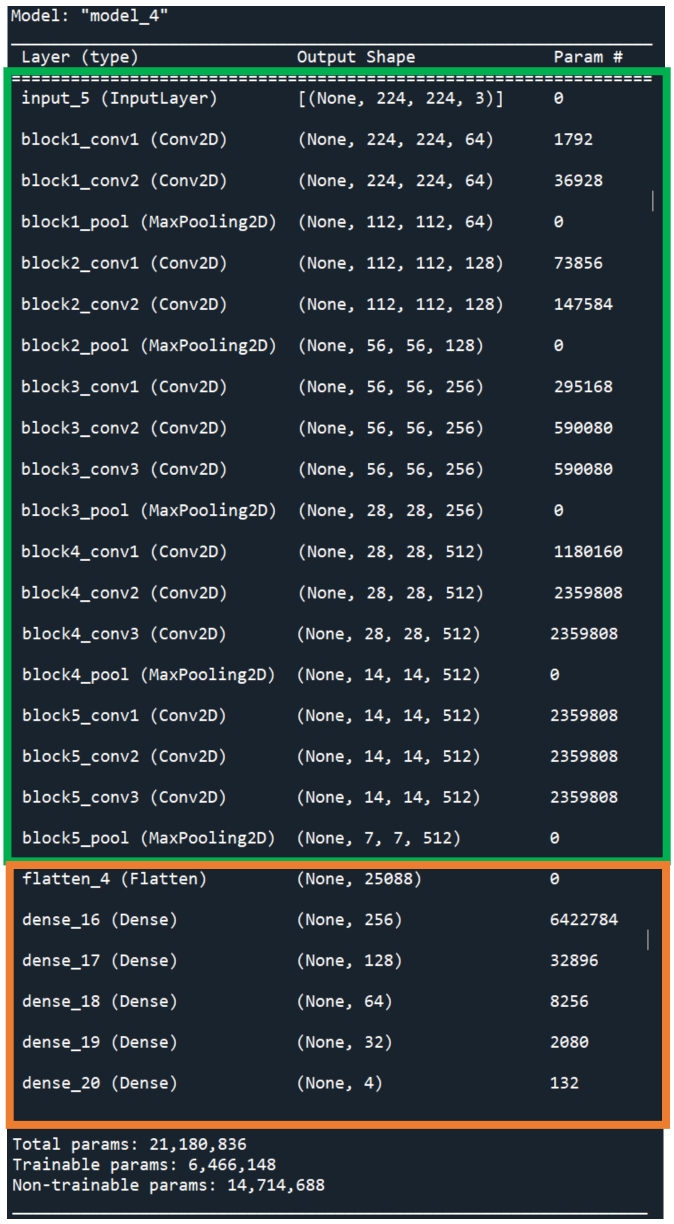


*Figure 20: VGG16 architecture with transfer learning considerations*

## Settings



```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_________________________________________________________________
```

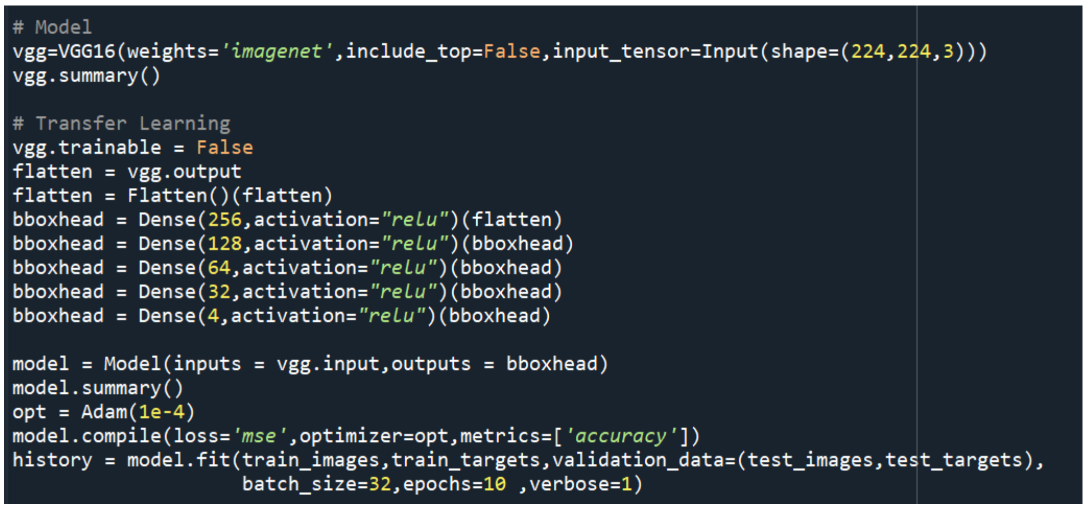


```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

 flatten_4 (Flatten)         (None, 25088)             0

 dense_16 (Dense)            (None, 256)               6422784

 dense_17 (Dense)            (None, 128)               32896

 dense_18 (Dense)            (None, 64)                8256

 dense_19 (Dense)            (None, 32)                2080

 dense_20 (Dense)            (None, 4)                 132

Total params: 21,180,836
Trainable params: 6,466,148
Non-trainable params: 14,714,688
_________________________________________________________________
```

**Frozen convolutional layers:** not trained

**Fully connected layers:** trained with the dataset

*Figure 21: VGG16 layers in details*



```python
# Model
vgg=VGG16(weights='imagenet',include_top=False,input_tensor=Input(shape=(224,224,3)))
vgg.summary()

# Transfer Learning
vgg.trainable = False
flatten = vgg.output
flatten = Flatten()(flatten)
bboxhead = Dense(256,activation="relu")(flatten)
bboxhead = Dense(128,activation="relu")(bboxhead)
bboxhead = Dense(64,activation="relu")(bboxhead)
bboxhead = Dense(32,activation="relu")(bboxhead)
bboxhead = Dense(4,activation="relu")(bboxhead)

model = Model(inputs = vgg.input,outputs = bboxhead)
model.summary()
opt = Adam(1e-4)
model.compile(loss='mse',optimizer=opt,metrics=['accuracy'])
history = model.fit(train_images,train_targets,validation_data=(test_images,test_targets),
                    batch_size=32,epochs=10 ,verbose=1)
```

*Figure 22: Code detailed for the transfer learning part*

In the same way as before in the code (*Figure 22*), the ReLU function activation allows to separate negative values in the convolution output and prevent the exponential growth in the computation.

In the algorithm, the loss function must be minimized using weights and parameters optimization. Since object localization is a regression problem, it is needed to use a regression loss function (*Figure 23*). In this case, L2 distance is used (Mean Squared Root '*mse'* distance).

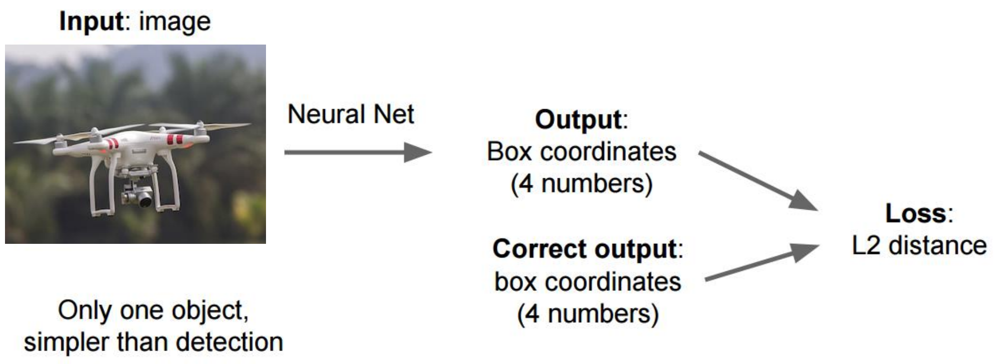


*Figure 23: Pipeline applied for the prediction*

After the training, bounding boxes parameters obtained (normalized and adapted for a 224x224 pixel image) are denormalized and adapted to be plotted on the real image (size of the picture in the dataset).

Other pre-trained models such as *InceptionV3, Darknet, Resnet* or *MobileNet* could also been used for this task. But because *VGG16* is known for its good performances and was suitable to obtain correct bounding boxes, it has been kept for the final results.
More performant systems regarding multi-objects classification and localization can be applied such as RCNN, Faster RCNN, YOLO or SSD. However, since this task was focused on a single object localization detecting UAV on an image, the bounding boxes regression has been performed without using the previous models mentioned.

## 3.3 Results and evaluation

### Performance

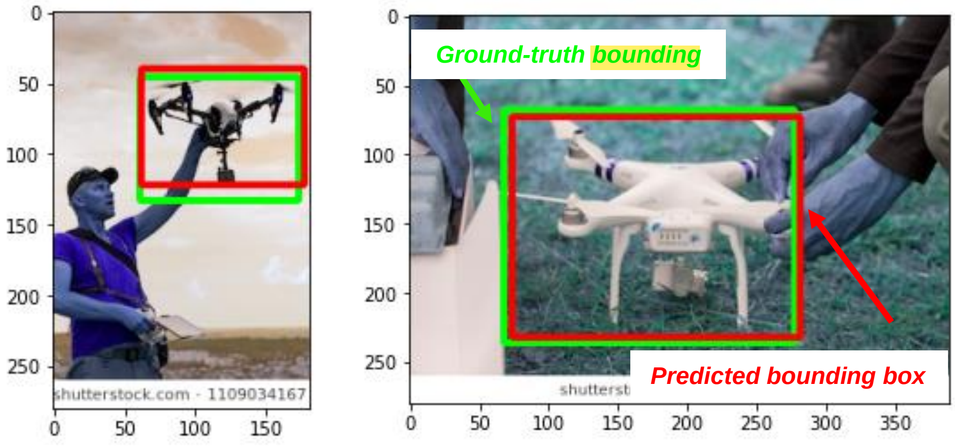
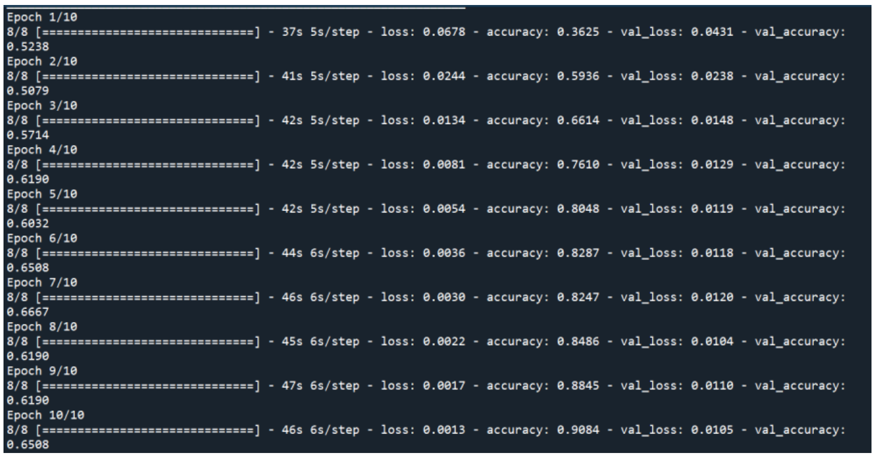


Figure 24: Examples of a predictions

The accuracy value at the end of the final epoch is **90,84%**, which is good (*Figure 26*). The neural network is about 90% accurate in determining the bounding box for the training data. Regarding the unseen data, it returns an accuracy of **65,08%** (*Figure 25*).



```
Epoch 1/10
8/8 [==============================] - 37s 5s/step - loss: 0.0678 - accuracy: 0.3625 - val_loss: 0.0431 - val_accuracy:
0.5238
Epoch 2/10
8/8 [==============================] - 41s 5s/step - loss: 0.0244 - accuracy: 0.5936 - val_loss: 0.0238 - val_accuracy:
0.5079
Epoch 3/10
8/8 [==============================] - 42s 5s/step - loss: 0.0134 - accuracy: 0.6614 - val_loss: 0.0148 - val_accuracy:
0.5714
Epoch 4/10
8/8 [==============================] - 42s 5s/step - loss: 0.0081 - accuracy: 0.7610 - val_loss: 0.0129 - val_accuracy:
0.6190
Epoch 5/10
8/8 [==============================] - 42s 5s/step - loss: 0.0054 - accuracy: 0.8048 - val_loss: 0.0119 - val_accuracy:
0.6032
Epoch 6/10
8/8 [==============================] - 44s 6s/step - loss: 0.0036 - accuracy: 0.8287 - val_loss: 0.0118 - val_accuracy:
0.6508
Epoch 7/10
8/8 [==============================] - 46s 6s/step - loss: 0.0030 - accuracy: 0.8247 - val_loss: 0.0120 - val_accuracy:
0.6667
Epoch 8/10
8/8 [==============================] - 45s 6s/step - loss: 0.0022 - accuracy: 0.8486 - val_loss: 0.0104 - val_accuracy:
0.6190
Epoch 9/10
8/8 [==============================] - 47s 6s/step - loss: 0.0017 - accuracy: 0.8845 - val_loss: 0.0110 - val_accuracy:
0.6190
Epoch 10/10
8/8 [==============================] - 46s 6s/step - loss: 0.0013 - accuracy: 0.9084 - val_loss: 0.0105 - val_accuracy:
0.6508
```

Figure 25: Accuracy and loss report for the model

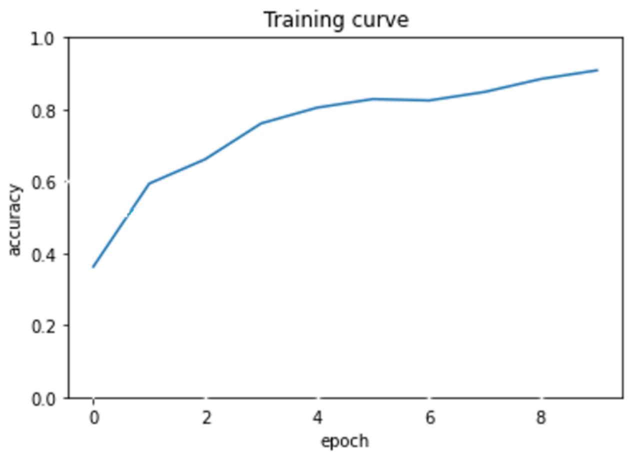


Figure 26: Training curve for the accuracy

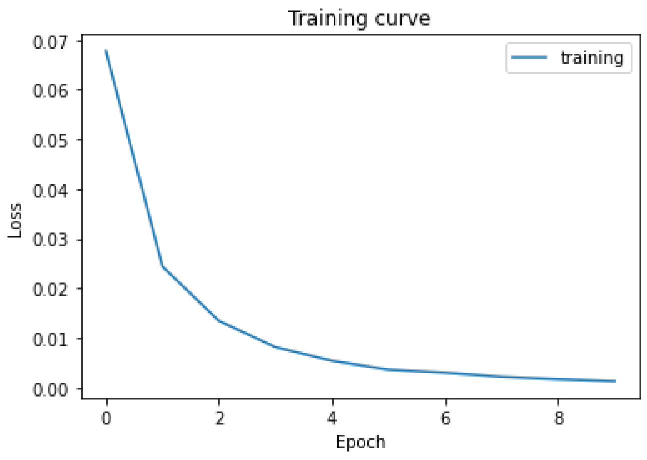


Figure 27: Training curve for the loss

## Validation

To continue, learning curves are plotted in order to evaluate further the performances (*Figure 28 and 29*).

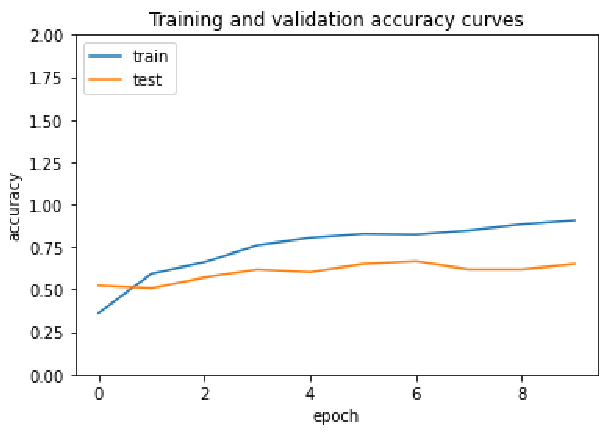


*Figure 28: Learning curves for the accuracy*

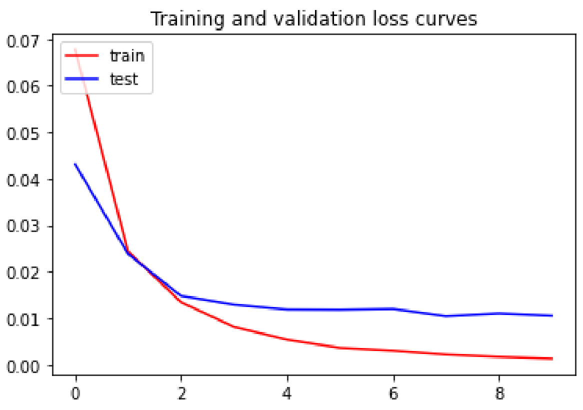


*Figure 29: Learning curves for the loss*

After conducting a cross validation, the curves indicate that for the 10 epochs configured there is no overfitting. Indeed, the curves are very close and fluctuate little (having low average magnitude of the test curve for the number of epochs greater than 2: 0.01 for the loss and 0.2 for the accuracy. It is possible that the optimal number of epochs is greater than 10, but the algorithm learning time is with this value already very substantial, and the results obtained are very good. Therefore, 10 as the number of epochs is the final value selected.

## Metric used

In a way to evaluate the precision of the predicted bounding box to check the performance on unseen data, the *Intersection over Union* (IoU) evaluation metric can be used. It measures the accuracy of the UAV detector considering the ground truth bounding box and the predicted one. IoU is the ratio of intersection and union, where the intersection is the area of overleap between the ground truth and the predicted bounding box. At the opposite, the union is the total area covered by both bounding boxes (*Figure 30*). It ranges between 0 and 1 (ideal performance).

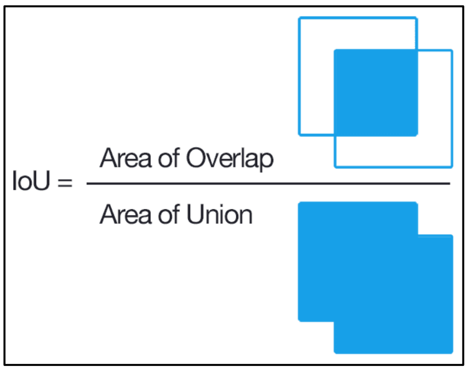


Figure 30: Intersection Over Union calculation

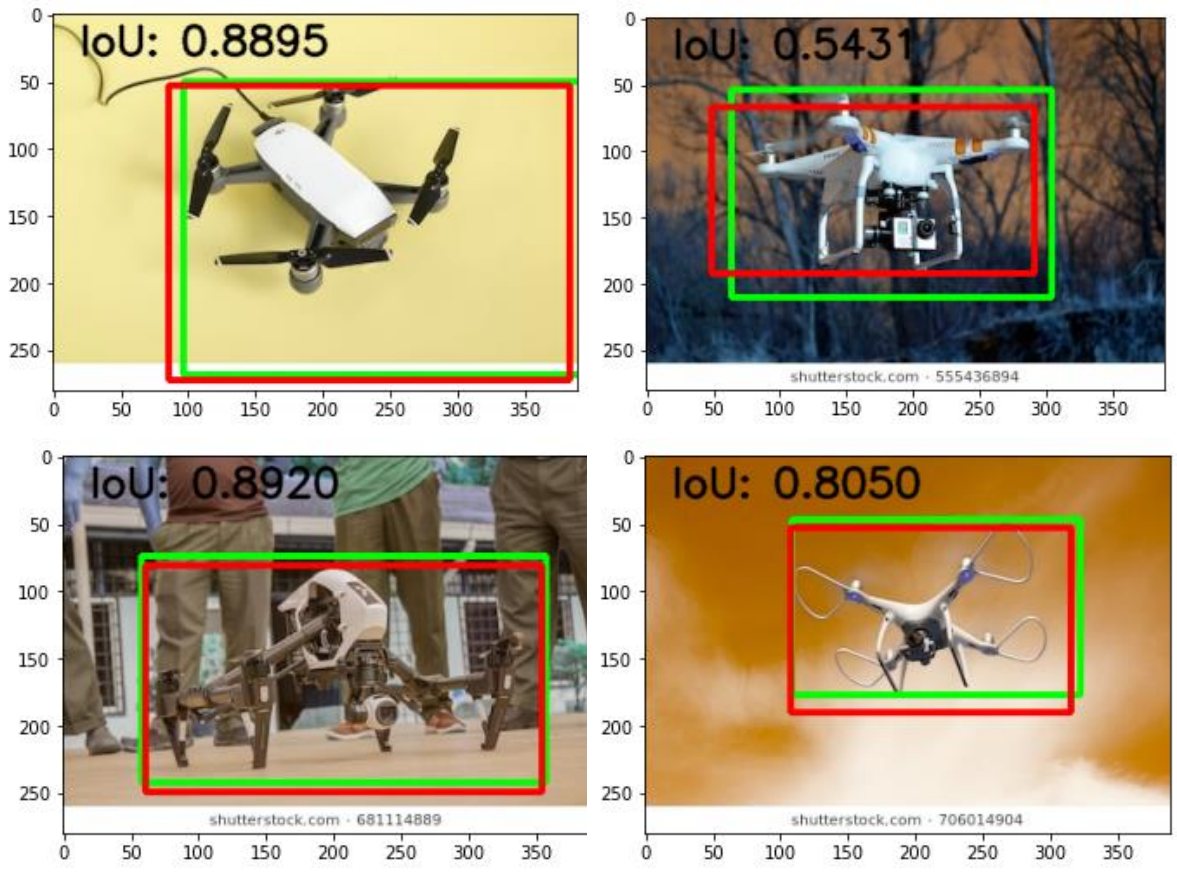

IoU: 0.8895
IoU: 0.5431
IoU: 0.8920
IoU: 0.8050
shutterstock.com · 555436894
shutterstock.com · 681114889
shutterstock.com · 706014904

Figure 31: Examples of predictions with metrics

For some images, the metric is displayed next to the prediction (*Figure 31*). As it is observable on the examples, the metric is very sensitive. Even when the predicted bounding box seems to be very good, the IoU can't achieve 1. At best, it is at 0.9 such as the third picture. Therefore, a IoU higher than 0.5 can reveal itself to be quite good.

# 4. Conclusion

In the first task, a well-known classifier for handwritten digits recognition has been performed and acquired very good results without a so deep neural network.

In the second task, most complex parameters setting, and models needed to be used since it was a tougher task. By training a second time a pre-trained model VGG16 with a given dataset, the network learnt to predict localization of an UAV on a picture by regression. To do that, many parameters need to be managed such as the size of the image controlled to have a working system able to predict adequate results.

In both, it is essential to understand at first the objective and how to achieve it (classification or regression), in order to adapt the parameters and the functions used (activation, loss function…) accordingly. More, analysing the data at first step in a pre-processing is as important as the model setting by itself, since without convenient data, the model couldn't reach good predictions.

# 5. Contents of figures

*Figure 1: Image example from the MNIST Dataset* .......... *4*
*Figure 2: Code detailed for Model A* .......... *5*
*Figure 3: Code detailed for Model B* .......... *5*
*Figure 4: Training curve for the accuracy (Model A)* .......... *7*
*Figure 5: Training curve for the loss (Model A)* .......... *7*
*Figure 6: Visualization of the ideal complexity* .......... *7*
*Figure 7: Learning curves for the accuracy (Model A)* .......... *8*
*Figure 8: Learning curves for the loss (Model A)* .......... *8*
*Figure 9: Prediction example for a digit image picked on the net* .......... *9*
*Figure 10: Training curve for the accuracy (Model B)* .......... *9*
*Figure 11: Training curve for the loss (Model B)* .......... *9*
*Figure 12: Learning curves for the accuracy (Model B)* .......... *9*
*Figure 13: Learning curves for the loss (Model B)* .......... *9*
*Figure 14: Image examples from the given dataset* .......... *11*
*Figure 15: Image examples with ground truth bounding boxes plotted* .......... *12*
*Figure 16: Image resize visualization* .......... *12*
*Figure 17: Pipeline used* .......... *13*
*Figure 18: Transfer learning principle* .......... *13*
*Figure 19: VGG16 architecture* .......... *14*
*Figure 20: VGG16 architecture with transfer learning considerations* .......... *14*
*Figure 21: VGG16 layers in details* .......... *15*
*Figure 22: Code detailed for the transfer learning part* .......... *15*
*Figure 23: Pipeline applied for the prediction* .......... *16*
*Figure 24: Examples of predictions* .......... *17*
*Figure 25: Accuracy and loss report for the model* .......... *17*
*Figure 26: Training curve for the accuracy* .......... *17*
*Figure 27: Training curve for the loss* .......... *17*
*Figure 28: Learning curves for the accuracy* .......... *18*
*Figure 29: Learning curves for the loss* .......... *18*
*Figure 30: Intersection Over Union calculation* .......... *19*
*Figure 31: Examples of predictions with metrics* .......... *19*

# 6. Bibliography

**_Contents_**

[1] *Machine Learning Mastery*, accessed 22 January 2022, https://machinelearningmastery.com/

[2] *Research Code*, accessed 23 January 2022, https://researchcode.com/

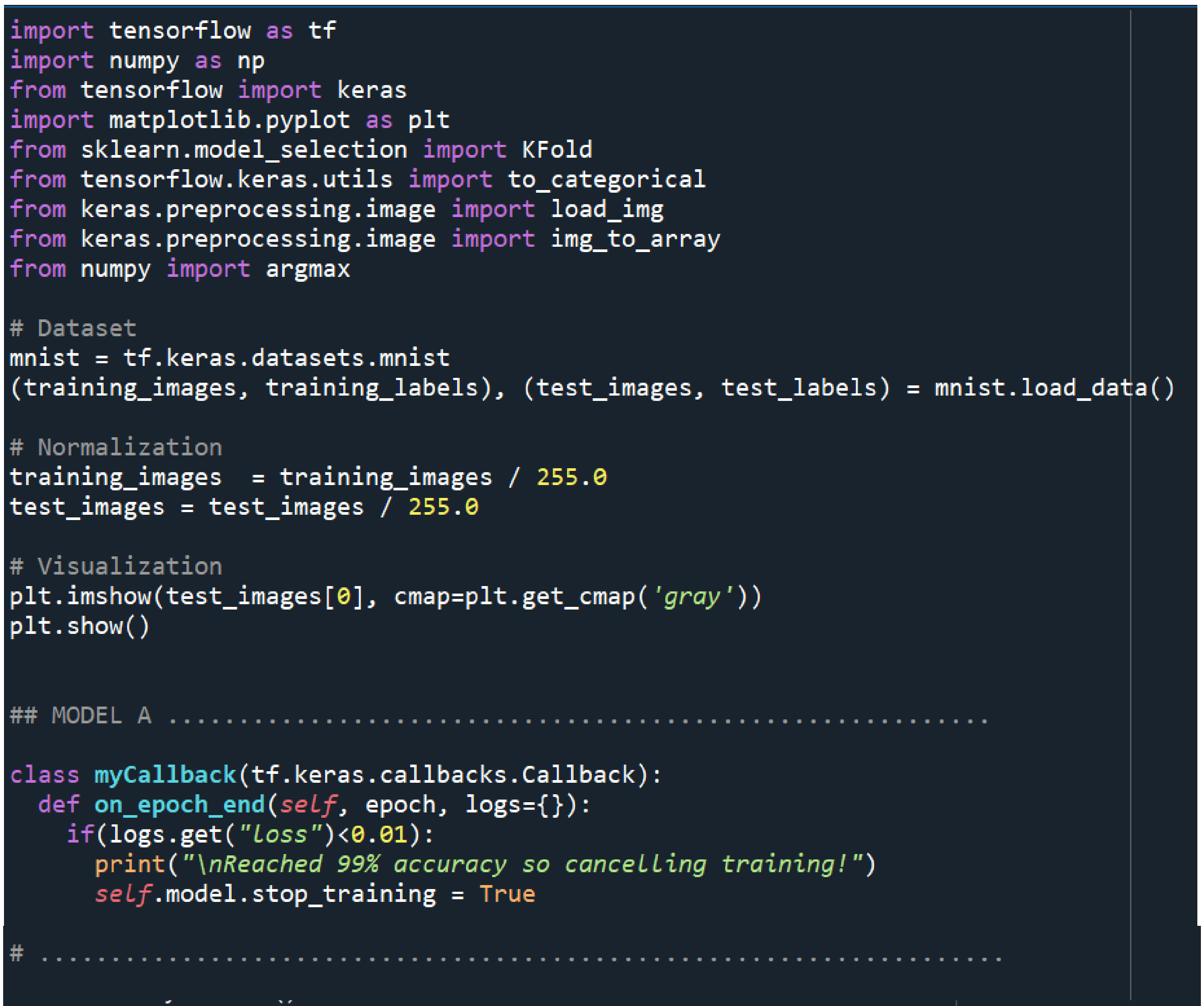
[3] *Github*, accessed 25 January 2022, https://github.com/

[4] *Medium*, accessed 24 January 2022, https://medium.com/

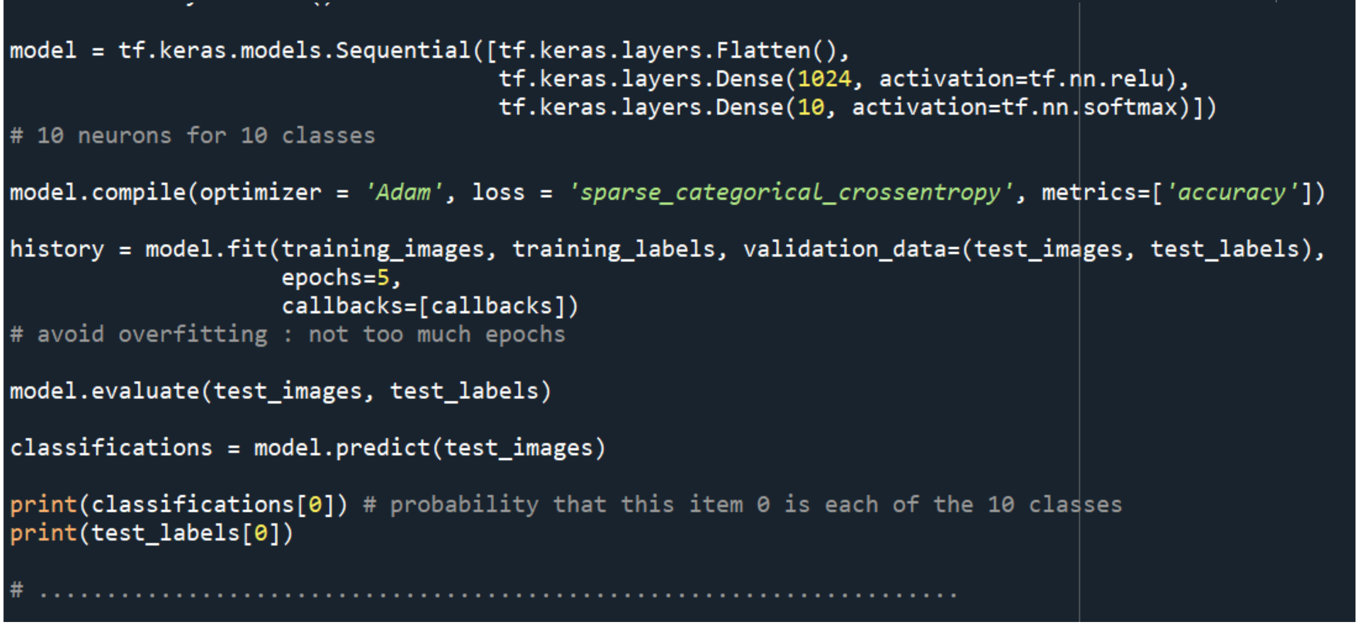
[5] *Towards Data Science*, accessed 23 January 2022, https://towardsdatascience.com/

# 7. Code

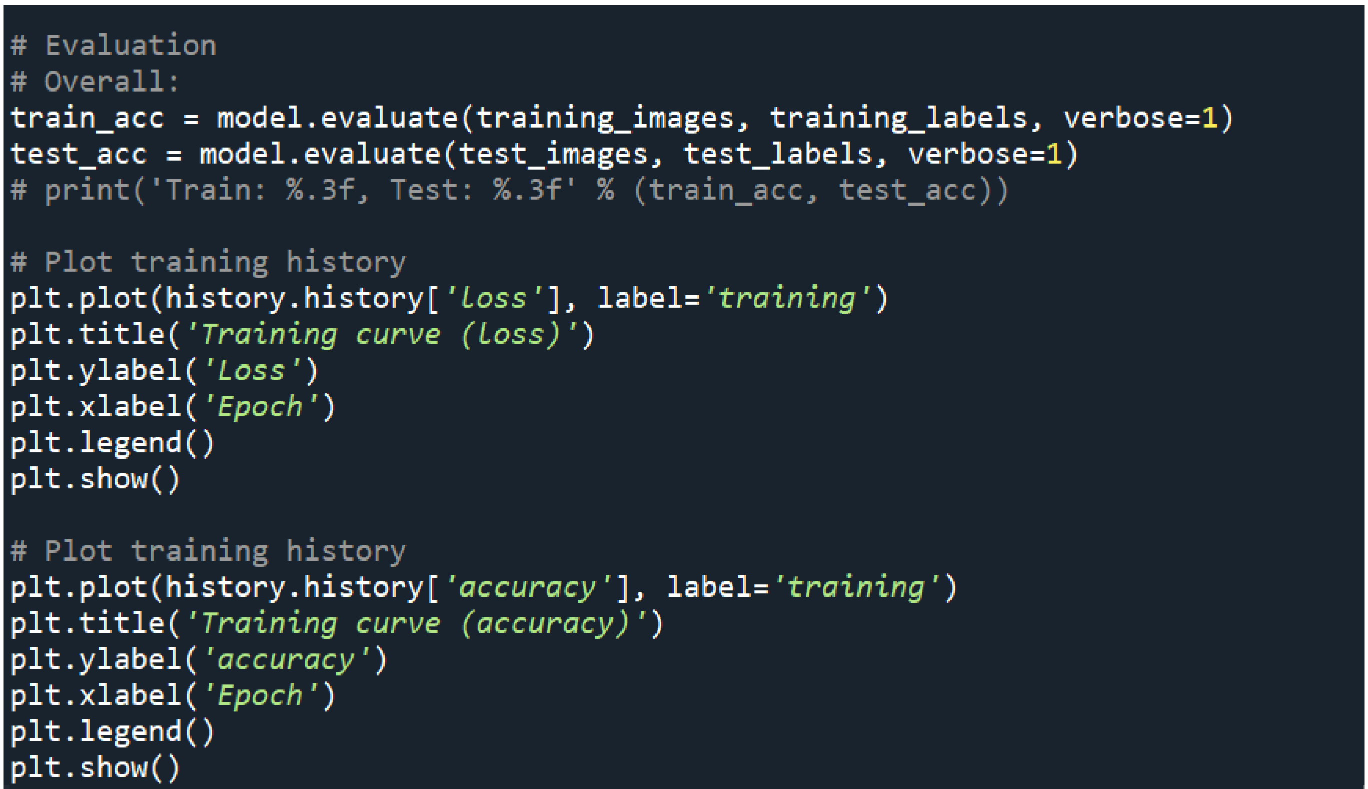
### Task 1



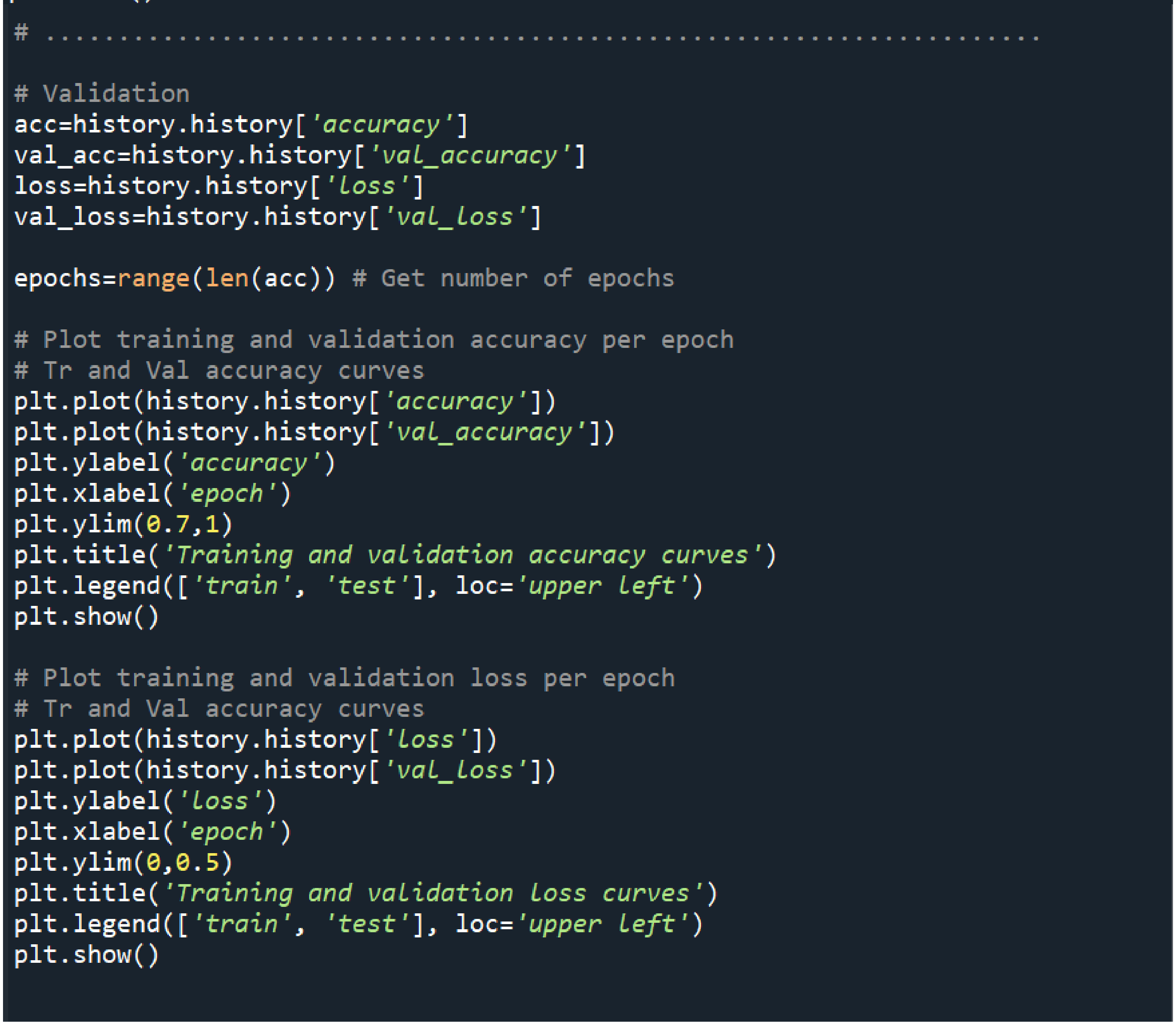
```
import tensorflow as tf
import numpy as np
from tensorflow import keras
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from tensorflow.keras.utils import to_categorical
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from numpy import argmax

# Dataset
mnist = tf.keras.datasets.mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()

# Normalization
training_images  = training_images / 255.0
test_images = test_images / 255.0

# Visualization
plt.imshow(test_images[0], cmap=plt.get_cmap('gray'))
plt.show()


## MODEL A ..........................................................

class myCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if(logs.get("loss")<0.01):
      print("\nReached 99% accuracy so cancelling training!")
      self.model.stop_training = True

# ....................................................................
```



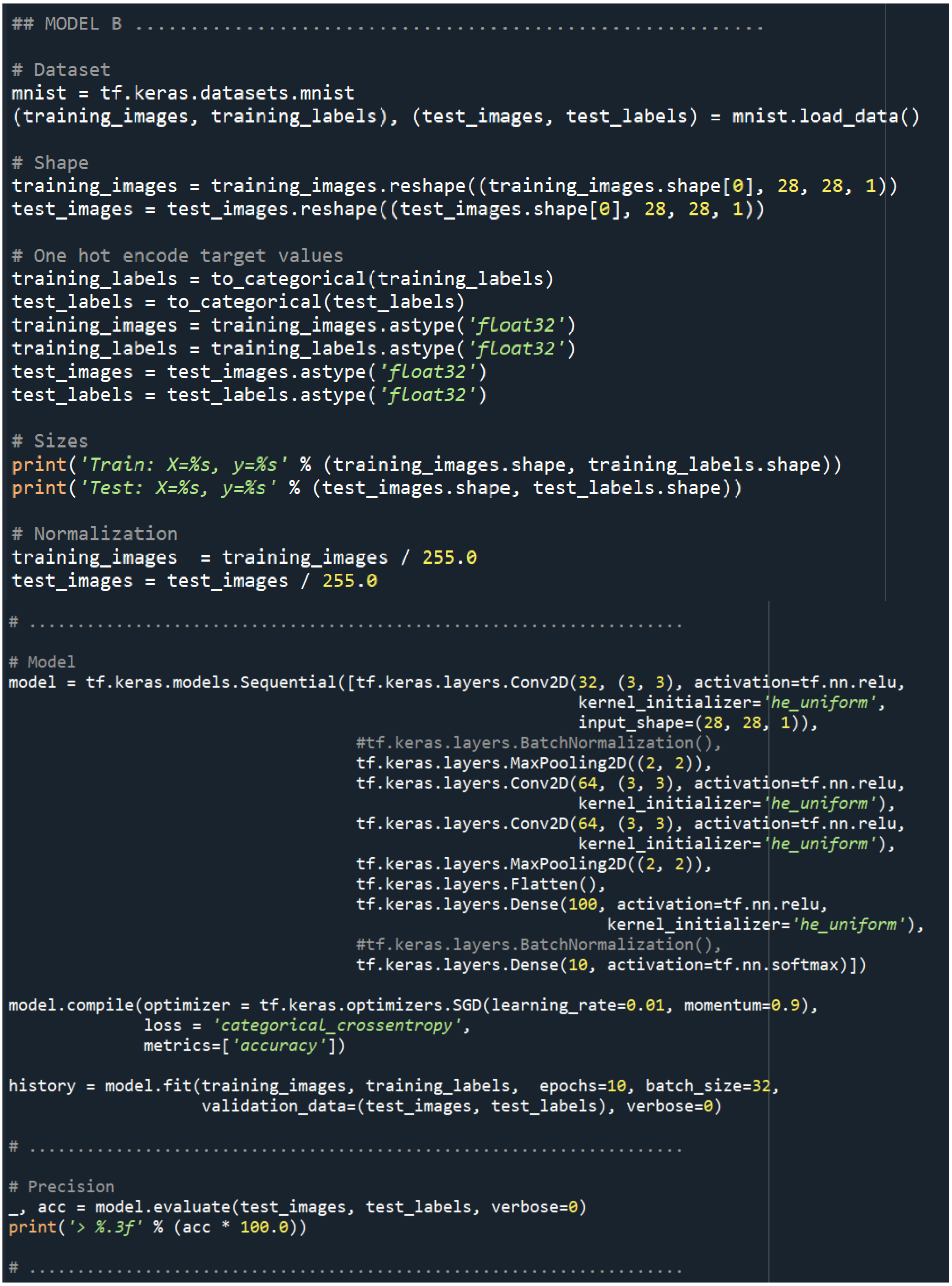
```
model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(1024, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
# 10 neurons for 10 classes

model.compile(optimizer = 'Adam', loss = 'sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(training_images, training_labels, validation_data=(test_images, test_labels),
                    epochs=5,
                    callbacks=[callbacks])
# avoid overfitting : not too much epochs

model.evaluate(test_images, test_labels)

classifications = model.predict(test_images)

print(classifications[0]) # probability that this item 0 is each of the 10 classes
print(test_labels[0])

# ....................................................................
```



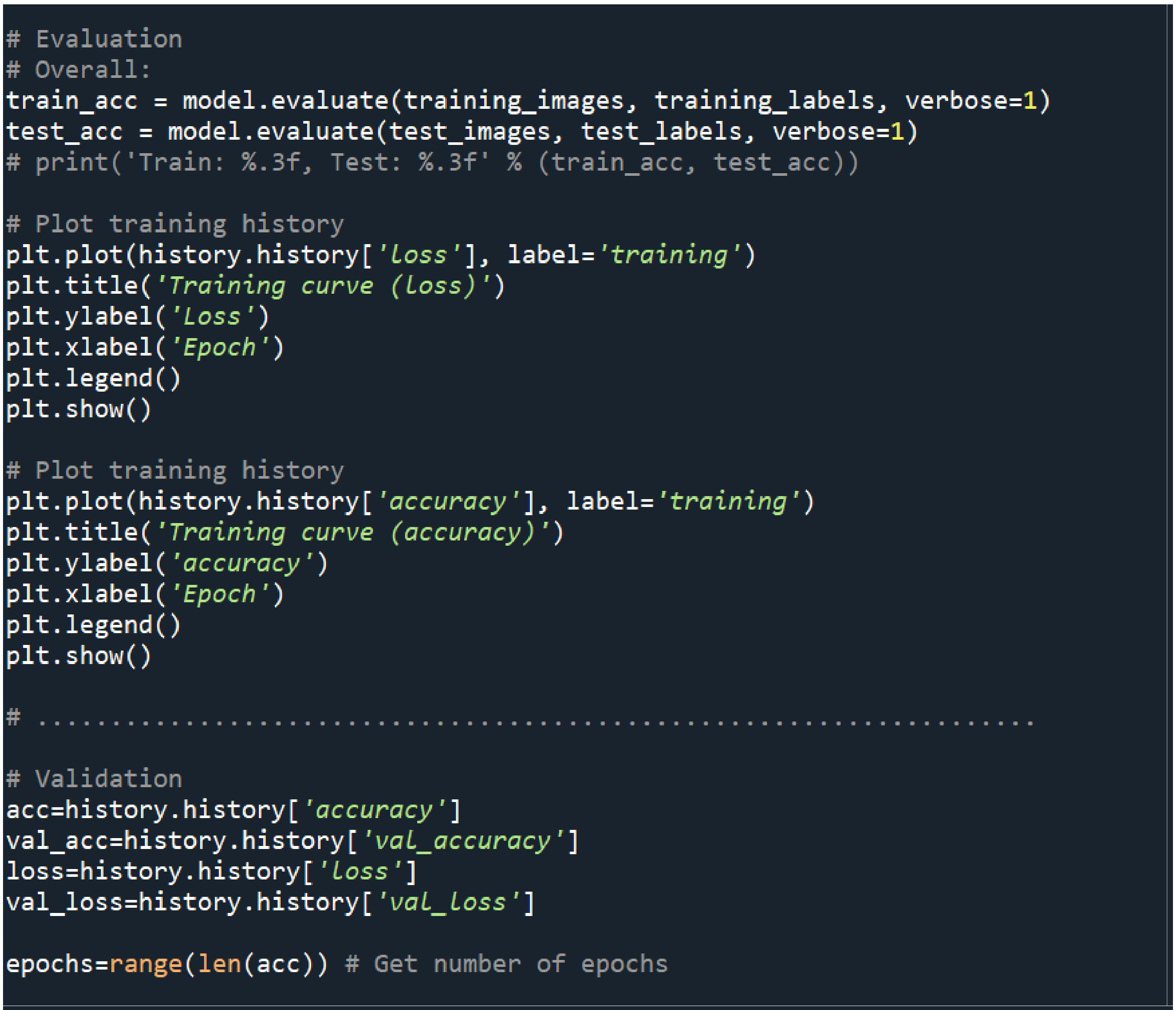
```
# Evaluation
# Overall:
train_acc = model.evaluate(training_images, training_labels, verbose=1)
test_acc = model.evaluate(test_images, test_labels, verbose=1)
# print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))

# Plot training history
plt.plot(history.history['loss'], label='training')
plt.title('Training curve (loss)')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

# Plot training history
plt.plot(history.history['accuracy'], label='training')
plt.title('Training curve (accuracy)')
plt.ylabel('accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```



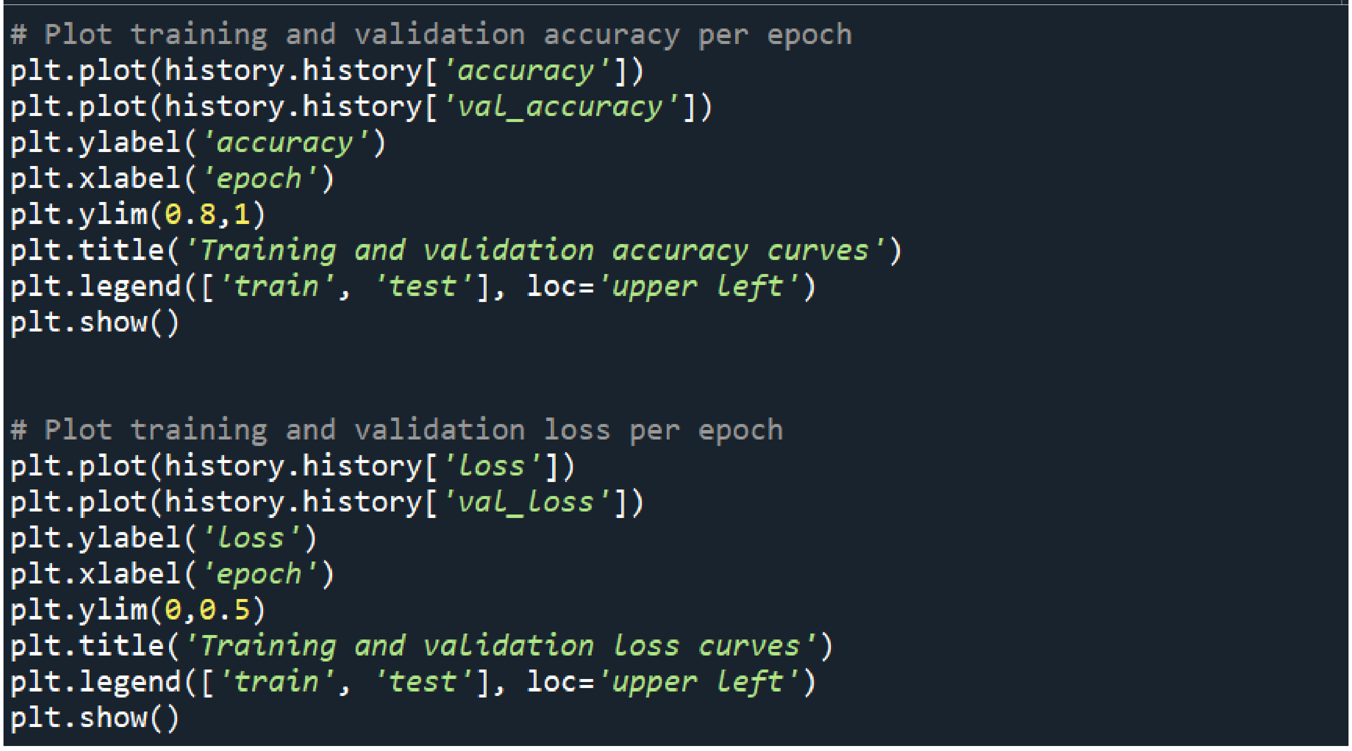
```
# .................................................................

# Validation
acc=history.history['accuracy']
val_acc=history.history['val_accuracy']
loss=history.history['loss']
val_loss=history.history['val_loss']

epochs=range(len(acc)) # Get number of epochs

# Plot training and validation accuracy per epoch
# Tr and Val accuracy curves
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.ylim(0.7,1)
plt.title('Training and validation accuracy curves')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# Plot training and validation loss per epoch
# Tr and Val accuracy curves
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.ylim(0,0.5)
plt.title('Training and validation loss curves')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



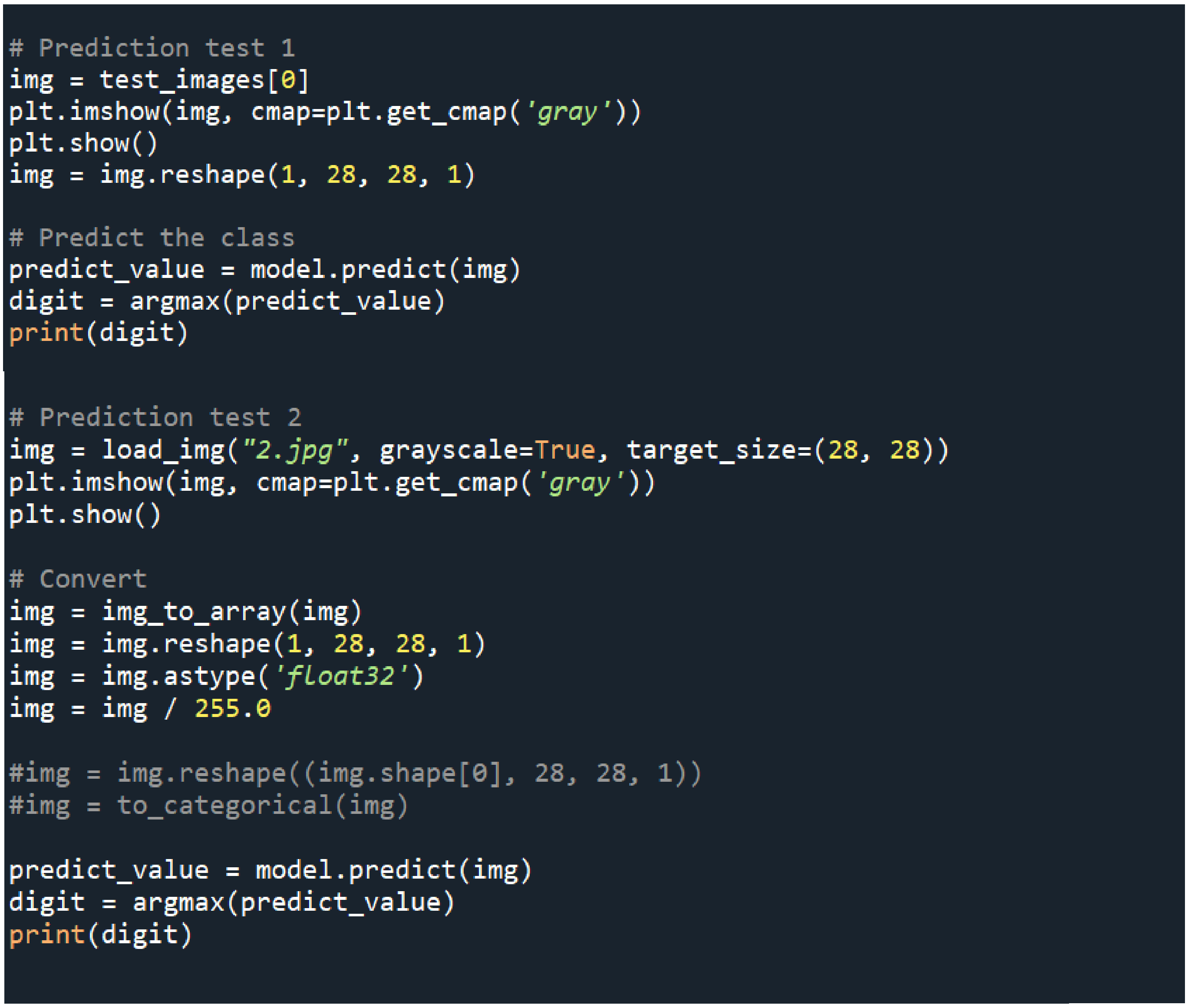
```
## MODEL B ..........................................................

# Dataset
mnist = tf.keras.datasets.mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()

# Shape
training_images = training_images.reshape((training_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))

# One hot encode target values
training_labels = to_categorical(training_labels)
test_labels = to_categorical(test_labels)
training_images = training_images.astype('float32')
training_labels = training_labels.astype('float32')
test_images = test_images.astype('float32')
test_labels = test_labels.astype('float32')

# Sizes
print('Train: X=%s, y=%s' % (training_images.shape, training_labels.shape))
print('Test: X=%s, y=%s' % (test_images.shape, test_labels.shape))

# Normalization
training_images  = training_images / 255.0
test_images = test_images / 255.0

# ...................................................................

# Model
model = tf.keras.models.Sequential([tf.keras.layers.Conv2D(32, (3, 3), activation=tf.nn.relu,
                                                           kernel_initializer='he_uniform',
                                                           input_shape=(28, 28, 1)),
                                    #tf.keras.layers.BatchNormalization(),
                                    tf.keras.layers.MaxPooling2D((2, 2)),
                                    tf.keras.layers.Conv2D(64, (3, 3), activation=tf.nn.relu,
                                                           kernel_initializer='he_uniform'),
                                    tf.keras.layers.Conv2D(64, (3, 3), activation=tf.nn.relu,
                                                           kernel_initializer='he_uniform'),
                                    tf.keras.layers.MaxPooling2D((2, 2)),
                                    tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(100, activation=tf.nn.relu,
                                                          kernel_initializer='he_uniform'),
                                    #tf.keras.layers.BatchNormalization(),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(training_images, training_labels,  epochs=10, batch_size=32,
                    validation_data=(test_images, test_labels), verbose=0)

# ...................................................................

# Precision
_, acc = model.evaluate(test_images, test_labels, verbose=0)
print('> %.3f' % (acc * 100.0))

# ...................................................................
```



```
# Evaluation
# Overall:
train_acc = model.evaluate(training_images, training_labels, verbose=1)
test_acc = model.evaluate(test_images, test_labels, verbose=1)
# print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))

# Plot training history
plt.plot(history.history['loss'], label='training')
plt.title('Training curve (loss)')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

# Plot training history
plt.plot(history.history['accuracy'], label='training')
plt.title('Training curve (accuracy)')
plt.ylabel('accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.show()

# ...................................................................

# Validation
acc=history.history['accuracy']
val_acc=history.history['val_accuracy']
loss=history.history['loss']
val_loss=history.history['val_loss']

epochs=range(len(acc)) # Get number of epochs
```

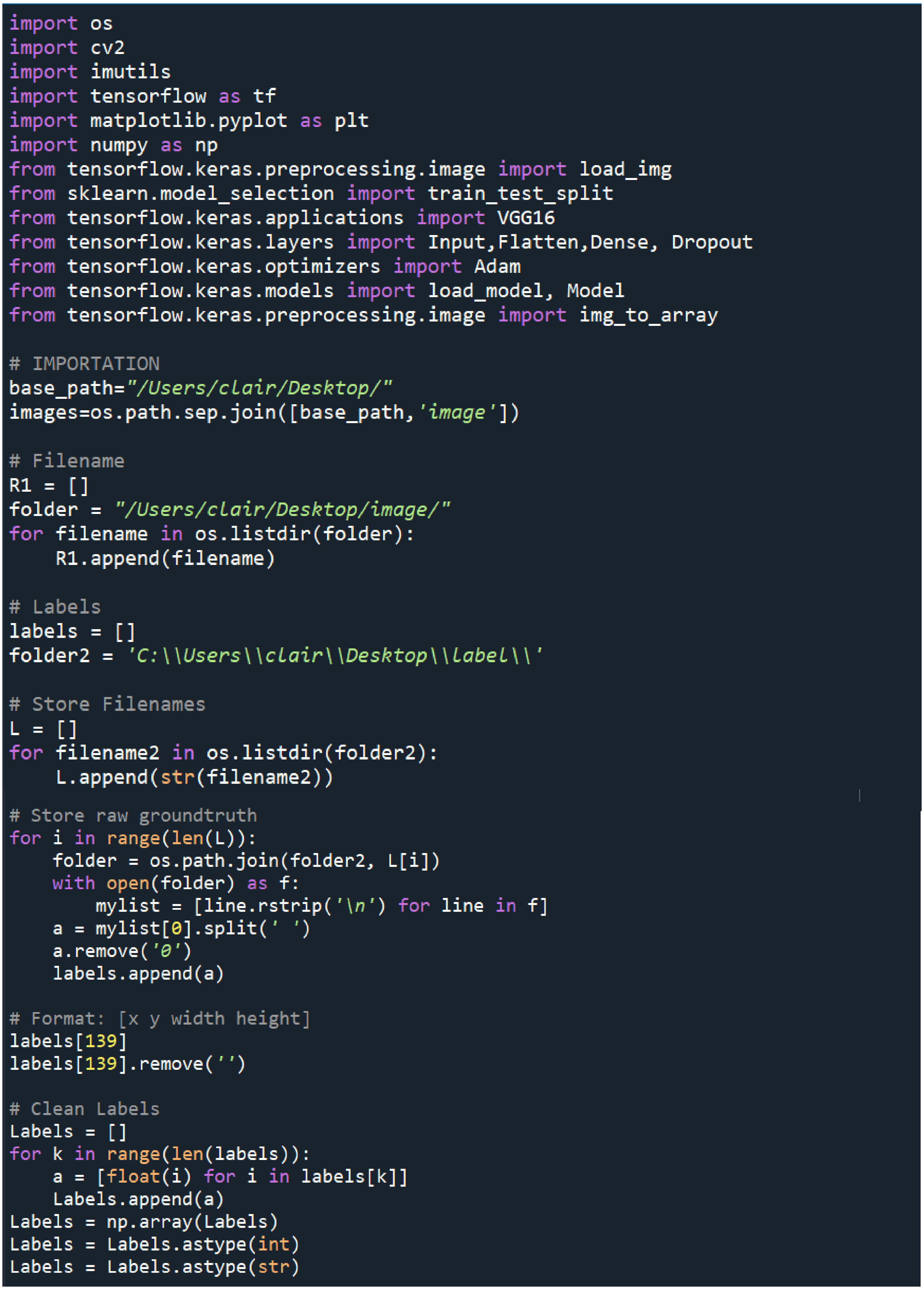


```
# Plot training and validation accuracy per epoch
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.ylim(0.8,1)
plt.title('Training and validation accuracy curves')
plt.legend(['train', 'test'], loc='upper left')
plt.show()


# Plot training and validation loss per epoch
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.ylim(0,0.5)
plt.title('Training and validation loss curves')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

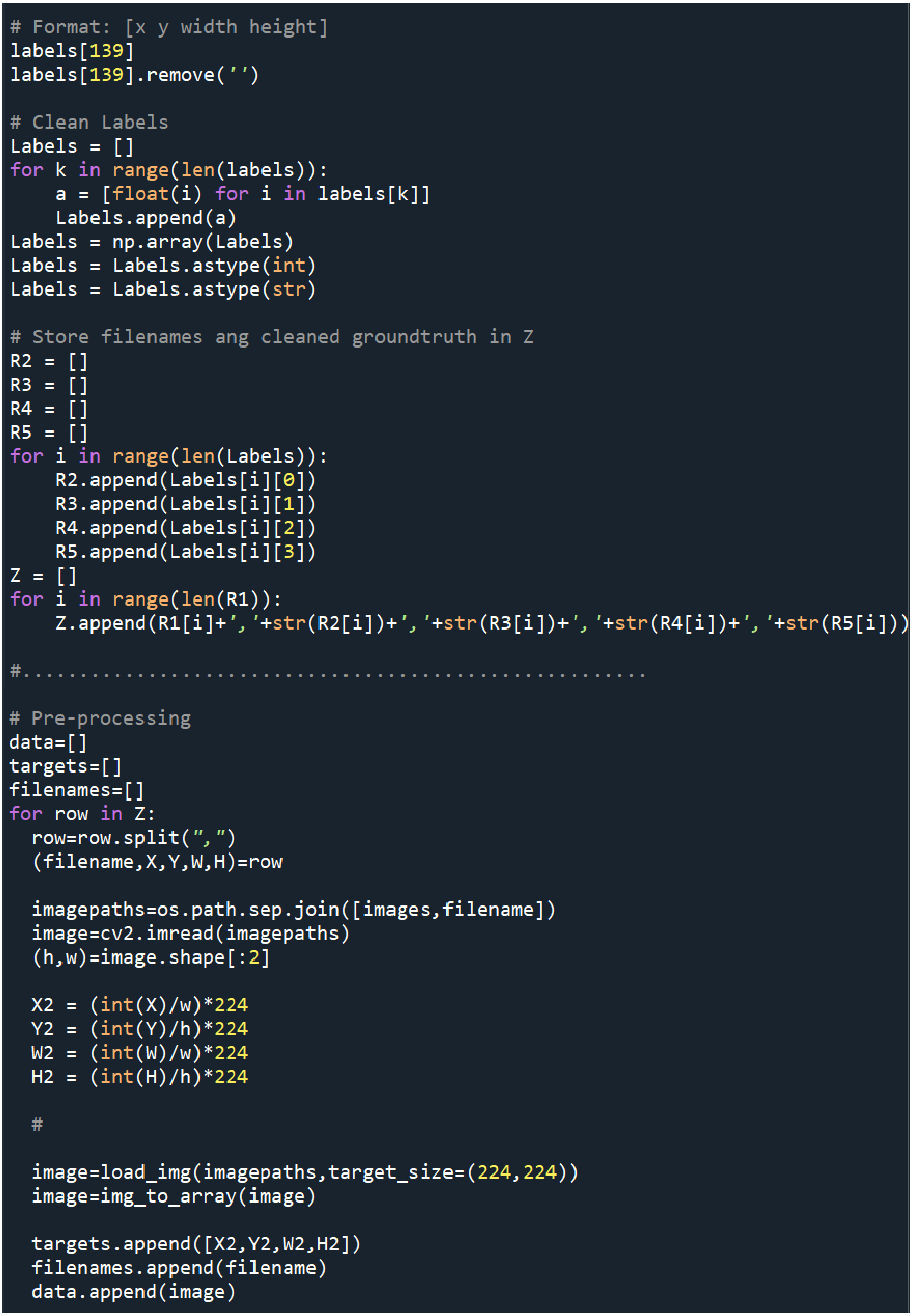


```python
# Prediction test 1
img = test_images[0]
plt.imshow(img, cmap=plt.get_cmap('gray'))
plt.show()
img = img.reshape(1, 28, 28, 1)

# Predict the class
predict_value = model.predict(img)
digit = argmax(predict_value)
print(digit)

# Prediction test 2
img = load_img("2.jpg", grayscale=True, target_size=(28, 28))
plt.imshow(img, cmap=plt.get_cmap('gray'))
plt.show()

# Convert
img = img_to_array(img)
img = img.reshape(1, 28, 28, 1)
img = img.astype('float32')
img = img / 255.0

#img = img.reshape((img.shape[0], 28, 28, 1))
#img = to_categorical(img)

predict_value = model.predict(img)
digit = argmax(predict_value)
print(digit)
```

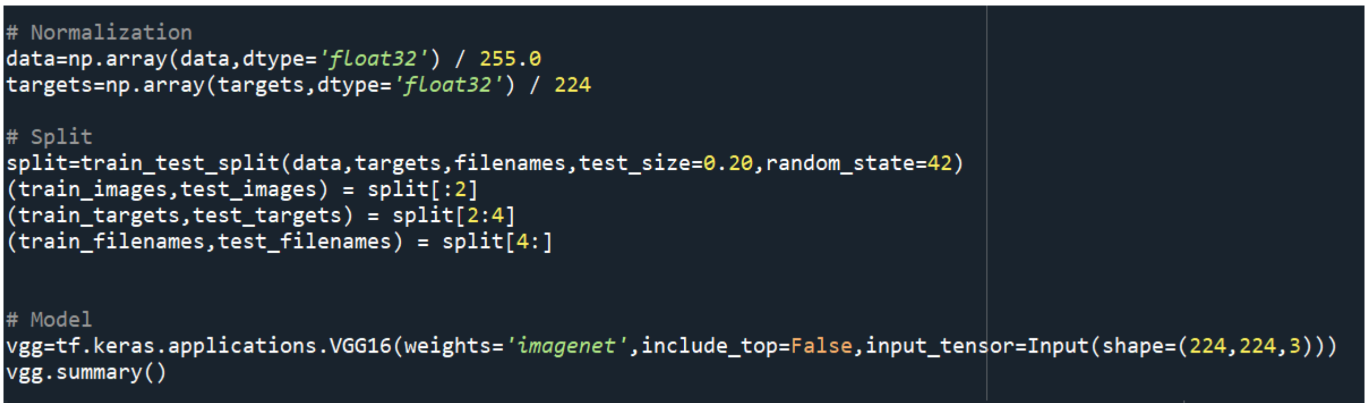
## Task 2



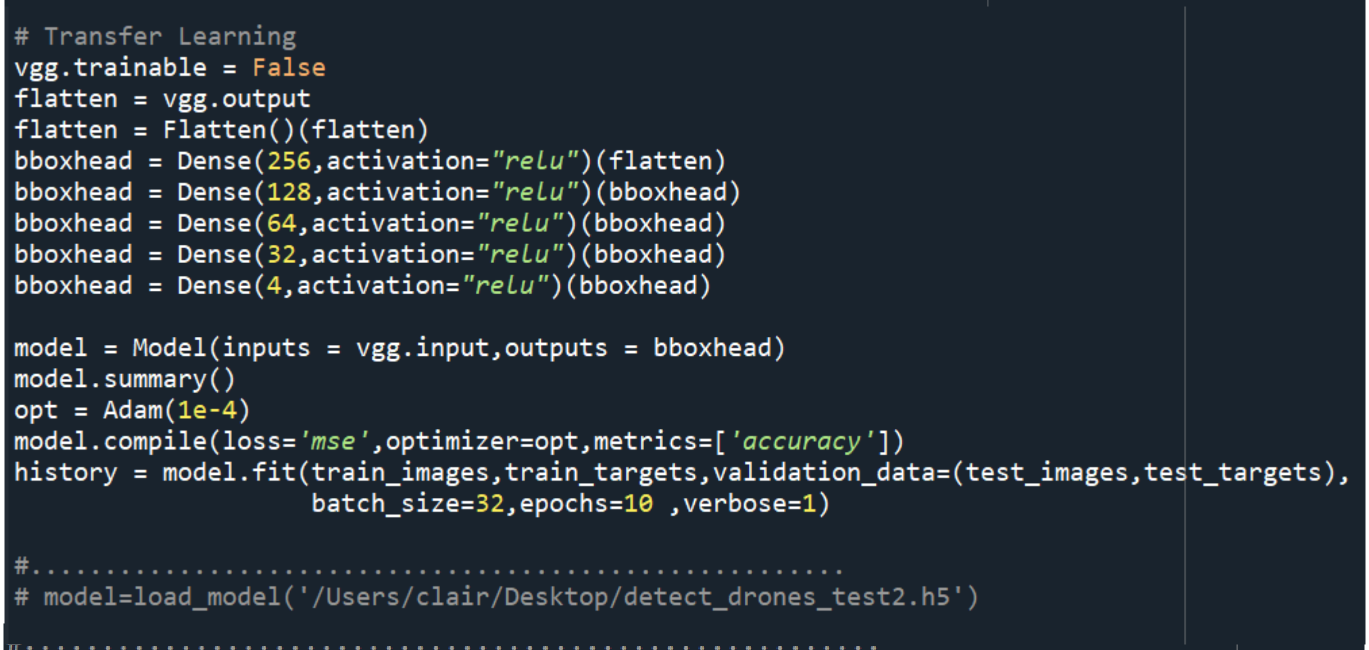
```
import os
import cv2
import imutils
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.preprocessing.image import load_img
from sklearn.model_selection import train_test_split
from tensorflow.keras.applications import VGG16
from tensorflow.keras.layers import Input,Flatten,Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import load_model, Model
from tensorflow.keras.preprocessing.image import img_to_array

# IMPORTATION
base_path="/Users/clair/Desktop/"
images=os.path.sep.join([base_path,'image'])

# Filename
R1 = []
folder = "/Users/clair/Desktop/image/"
for filename in os.listdir(folder):
    R1.append(filename)

# Labels
labels = []
folder2 = 'C:\\Users\\clair\\Desktop\\label\\'

# Store Filenames
L = []
for filename2 in os.listdir(folder2):
    L.append(str(filename2))

# Store raw groundtruth
for i in range(len(L)):
    folder = os.path.join(folder2, L[i])
    with open(folder) as f:
        mylist = [line.rstrip('\n') for line in f]
    a = mylist[0].split(' ')
    a.remove('0')
    labels.append(a)

# Format: [x y width height]
labels[139]
labels[139].remove('')

# Clean Labels
Labels = []
for k in range(len(labels)):
    a = [float(i) for i in labels[k]]
    Labels.append(a)
Labels = np.array(Labels)
Labels = Labels.astype(int)
Labels = Labels.astype(str)
```



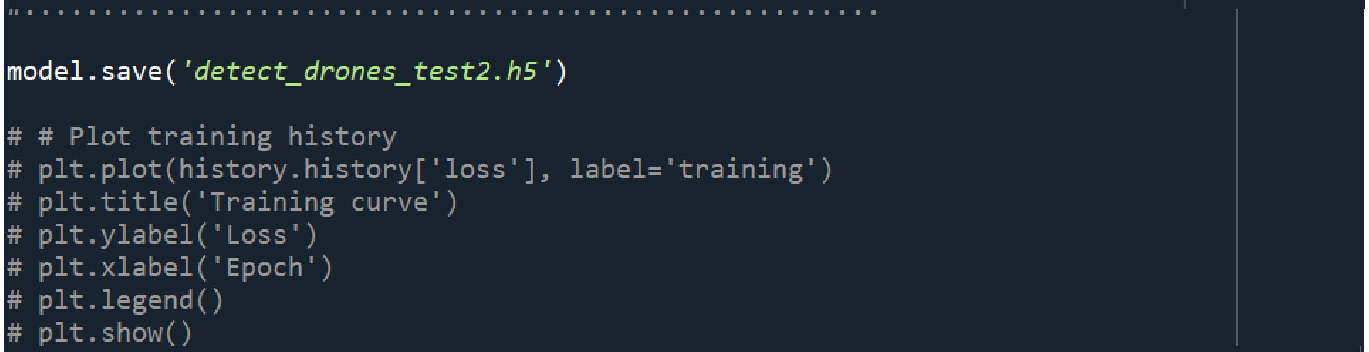
```
# Format: [x y width height]
labels[139]
labels[139].remove('')

# Clean Labels
Labels = []
for k in range(len(labels)):
    a = [float(i) for i in labels[k]]
    Labels.append(a)
Labels = np.array(Labels)
Labels = Labels.astype(int)
Labels = Labels.astype(str)

# Store filenames ang cleaned groundtruth in Z
R2 = []
R3 = []
R4 = []
R5 = []
for i in range(len(Labels)):
    R2.append(Labels[i][0])
    R3.append(Labels[i][1])
    R4.append(Labels[i][2])
    R5.append(Labels[i][3])
Z = []
for i in range(len(R1)):
    Z.append(R1[i]+','+str(R2[i])+','+str(R3[i])+','+str(R4[i])+','+str(R5[i]))

#.......................................................

# Pre-processing
data=[]
targets=[]
filenames=[]
for row in Z:
  row=row.split(",")
  (filename,X,Y,W,H)=row

  imagepaths=os.path.sep.join([images,filename])
  image=cv2.imread(imagepaths)
  (h,w)=image.shape[:2]

  X2 = (int(X)/w)*224
  Y2 = (int(Y)/h)*224
  W2 = (int(W)/w)*224
  H2 = (int(H)/h)*224

  #

  image=load_img(imagepaths,target_size=(224,224))
  image=img_to_array(image)

  targets.append([X2,Y2,W2,H2])
  filenames.append(filename)
  data.append(image)
```



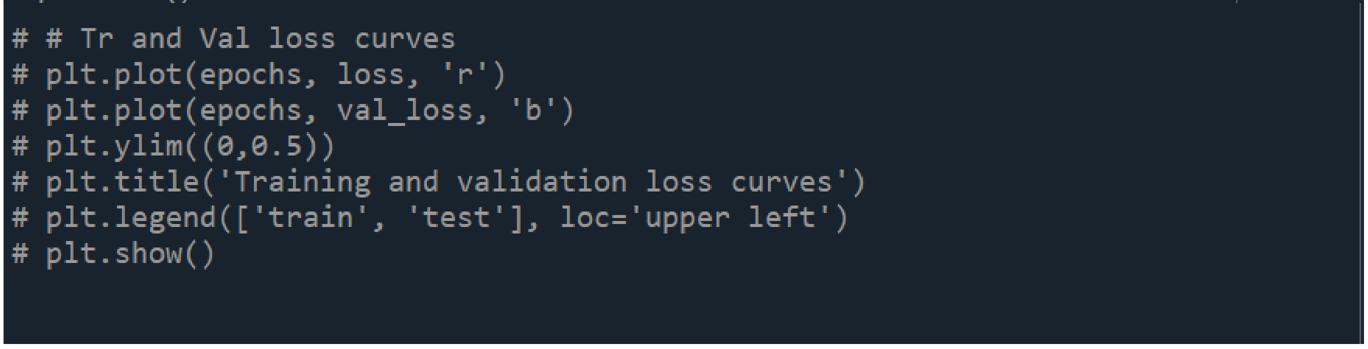
```
# Normalization
data=np.array(data,dtype='float32') / 255.0
targets=np.array(targets,dtype='float32') / 224

# Split
split=train_test_split(data,targets,filenames,test_size=0.20,random_state=42)
(train_images,test_images) = split[:2]
(train_targets,test_targets) = split[2:4]
(train_filenames,test_filenames) = split[4:]


# Model
vgg=tf.keras.applications.VGG16(weights='imagenet',include_top=False,input_tensor=Input(shape=(224,224,3)))
vgg.summary()
```



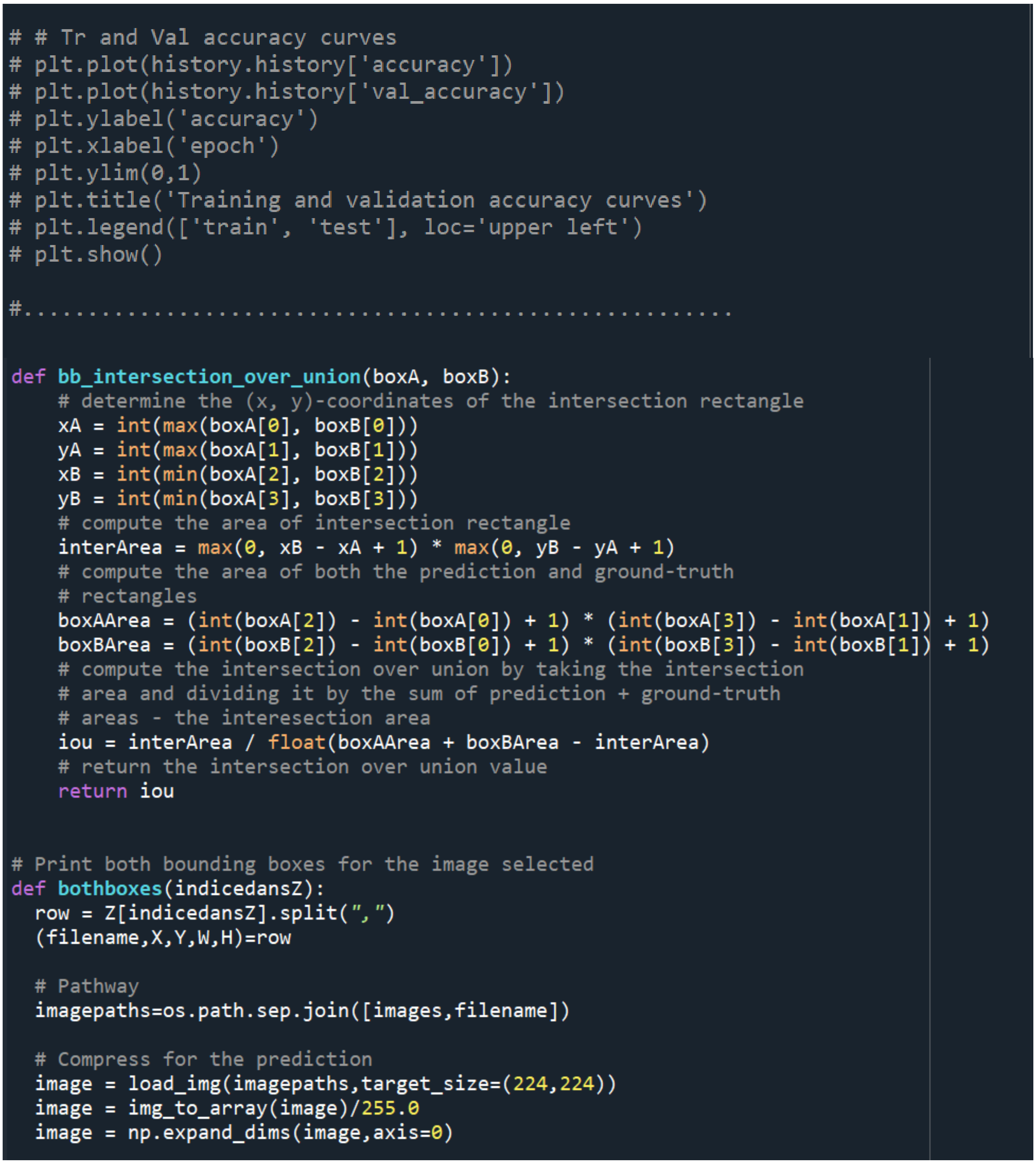
```
# Transfer Learning
vgg.trainable = False
flatten = vgg.output
flatten = Flatten()(flatten)
bboxhead = Dense(256,activation="relu")(flatten)
bboxhead = Dense(128,activation="relu")(bboxhead)
bboxhead = Dense(64,activation="relu")(bboxhead)
bboxhead = Dense(32,activation="relu")(bboxhead)
bboxhead = Dense(4,activation="relu")(bboxhead)

model = Model(inputs = vgg.input,outputs = bboxhead)
model.summary()
opt = Adam(1e-4)
model.compile(loss='mse',optimizer=opt,metrics=['accuracy'])
history = model.fit(train_images,train_targets,validation_data=(test_images,test_targets),
                    batch_size=32,epochs=10 ,verbose=1)

#.....................................................
# model=load_model('/Users/clair/Desktop/detect_drones_test2.h5')
```



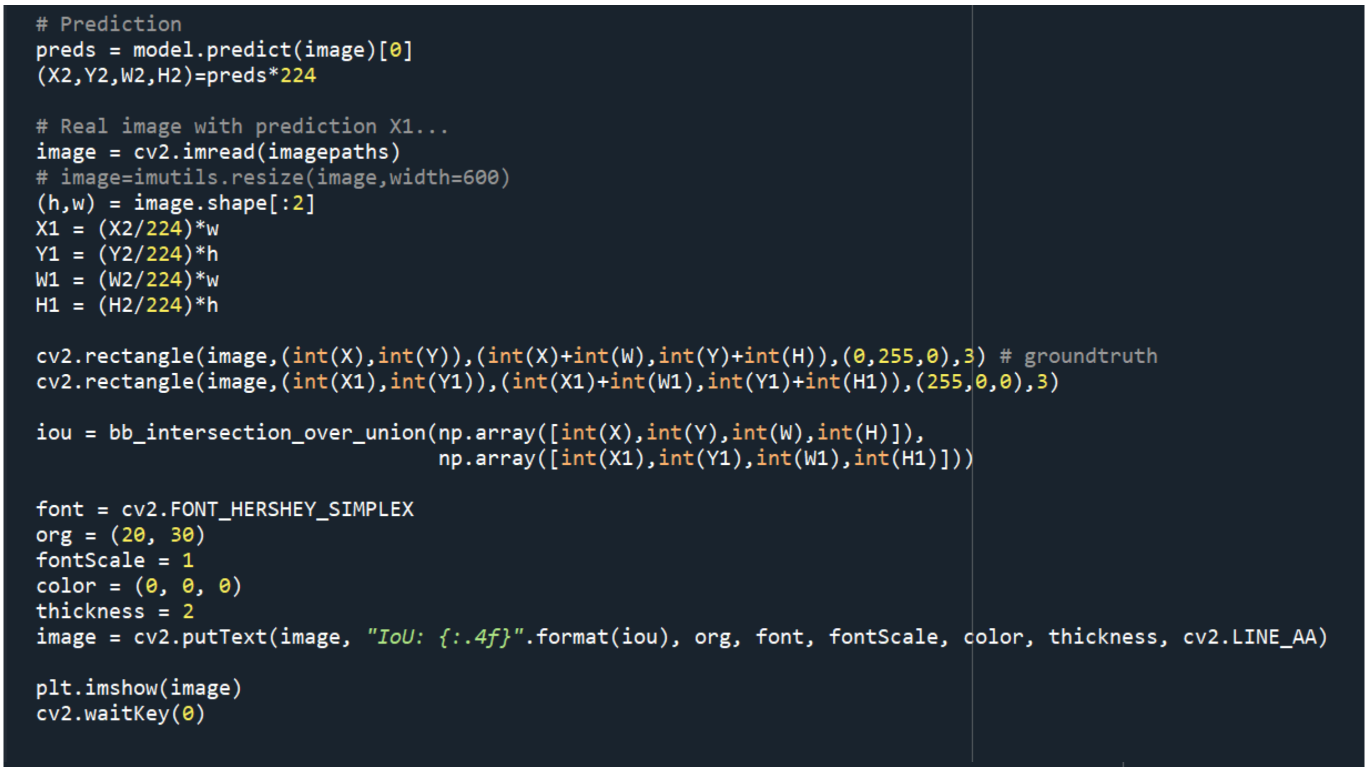
```
#.........................................................

model.save('detect_drones_test2.h5')

# # Plot training history
# plt.plot(history.history['loss'], label='training')
# plt.title('Training curve')
# plt.ylabel('Loss')
# plt.xlabel('Epoch')
# plt.legend()
# plt.show()
```



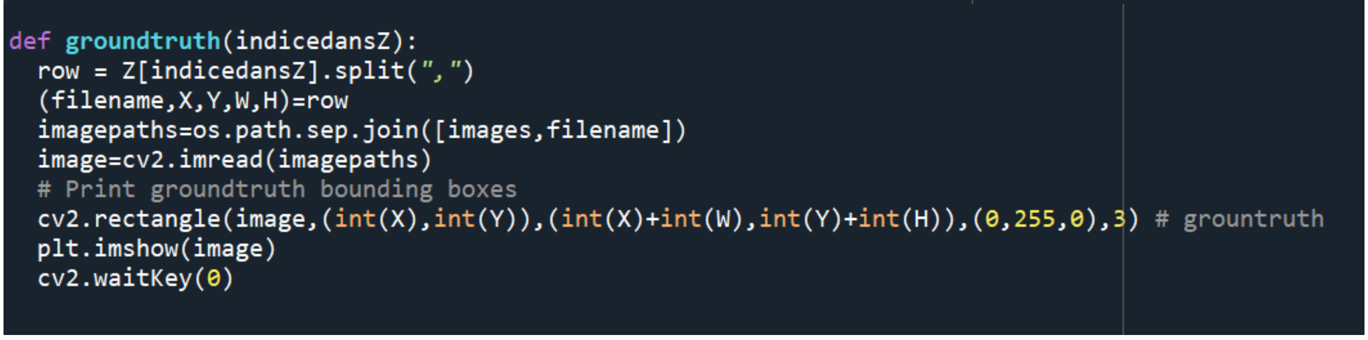
```
# # Tr and Val loss curves
# plt.plot(epochs, loss, 'r')
# plt.plot(epochs, val_loss, 'b')
# plt.ylim((0,0.5))
# plt.title('Training and validation loss curves')
# plt.legend(['train', 'test'], loc='upper left')
# plt.show()
```



```
# # Tr and Val accuracy curves
# plt.plot(history.history['accuracy'])
# plt.plot(history.history['val_accuracy'])
# plt.ylabel('accuracy')
# plt.xlabel('epoch')
# plt.ylim(0,1)
# plt.title('Training and validation accuracy curves')
# plt.legend(['train', 'test'], loc='upper left')
# plt.show()

#.............................................


def bb_intersection_over_union(boxA, boxB):
    # determine the (x, y)-coordinates of the intersection rectangle
    xA = int(max(boxA[0], boxB[0]))
    yA = int(max(boxA[1], boxB[1]))
    xB = int(min(boxA[2], boxB[2]))
    yB = int(min(boxA[3], boxB[3]))
    # compute the area of intersection rectangle
    interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)
    # compute the area of both the prediction and ground-truth
    # rectangles
    boxAArea = (int(boxA[2]) - int(boxA[0]) + 1) * (int(boxA[3]) - int(boxA[1]) + 1)
    boxBArea = (int(boxB[2]) - int(boxB[0]) + 1) * (int(boxB[3]) - int(boxB[1]) + 1)
    # compute the intersection over union by taking the intersection
    # area and dividing it by the sum of prediction + ground-truth
    # areas - the interesection area
    iou = interArea / float(boxAArea + boxBArea - interArea)
    # return the intersection over union value
    return iou


# Print both bounding boxes for the image selected
def bothboxes(indicedansZ):
  row = Z[indicedansZ].split(",")
  (filename,X,Y,W,H)=row

  # Pathway
  imagepaths=os.path.sep.join([images,filename])

  # Compress for the prediction
  image = load_img(imagepaths,target_size=(224,224))
  image = img_to_array(image)/255.0
  image = np.expand_dims(image,axis=0)
```



```
    # Prediction
    preds = model.predict(image)[0]
    (X2,Y2,W2,H2)=preds*224

    # Real image with prediction X1...
    image = cv2.imread(imagepaths)
    # image=imutils.resize(image,width=600)
    (h,w) = image.shape[:2]
    X1 = (X2/224)*w
    Y1 = (Y2/224)*h
    W1 = (W2/224)*w
    H1 = (H2/224)*h

    cv2.rectangle(image,(int(X),int(Y)),(int(X)+int(W),int(Y)+int(H)),(0,255,0),3) # groundtruth
    cv2.rectangle(image,(int(X1),int(Y1)),(int(X1)+int(W1),int(Y1)+int(H1)),(255,0,0),3)

    iou = bb_intersection_over_union(np.array([int(X),int(Y),int(W),int(H)]),
                                     np.array([int(X1),int(Y1),int(W1),int(H1)]))

    font = cv2.FONT_HERSHEY_SIMPLEX
    org = (20, 30)
    fontScale = 1
    color = (0, 0, 0)
    thickness = 2
    image = cv2.putText(image, "IoU: {:.4f}".format(iou), org, font, fontScale, color, thickness, cv2.LINE_AA)

    plt.imshow(image)
    cv2.waitKey(0)
```



```
def groundtruth(indicedansZ):
  row = Z[indicedansZ].split(",")
  (filename,X,Y,W,H)=row
  imagepaths=os.path.sep.join([images,filename])
  image=cv2.imread(imagepaths)
  # Print groundtruth bounding boxes
  cv2.rectangle(image,(int(X),int(Y)),(int(X)+int(W),int(Y)+int(H)),(0,255,0),3) # grountruth
  plt.imshow(image)
  cv2.waitKey(0)
```