Drone Detection Using Neural Networks

Statistics and Data Science for Engineers

Professor: Gabriel Gomez

GSI: Joohwan Seo

Team Number: 15

Team Members: Xiushi Shen, Xiaofeng Zhao, Lingxiao Pan, Shuhao Lu

Submission Date: *12/13/2022*

Problem Statement

Drones are becoming increasingly popular recreational toys and cinematography tools in consumer and industrial settings. The drone market share has been increasing in a steady paste. Normally, we can assume that only one drone is operating in a specific area at a time. However, this generalization is no longer true. Multiple drones can be flown in the same area and communicating between different drones is a difficult challenge to solve. When there are two or more drones operating in the same area, we must coordinate their location information to prevent costly mid-air collisions. The technique and application need to be generalizable so even if the two drones are different models operating on different systems, they can keep a safe distance without directly communicating with each other. Moreover, the solution needs to be cheap and use existing onboard sensors to prevent additional cost.

Solution Method

One costly method of solving the problem is to add additional communication equipment on the drones to coordinate between each other. However, a much simpler approach can be used. Machine learning, more extensively convolutional neural network (CNN) has been used extensively to detect objects using computer vision. It is a great method for detecting objects and identifying an object's location within an image. The method is used on drones for landmark detection and tracking, as such, redeploying the same system into drone detection and collision prevention seems like a reasonable option. When coordinating multiple drones operating at the same time, computer vision techniques can be used to detect drones and find the other drone's location within the image or video feed. In this project, we used extensive data with more than ten thousand images with drones and trained the model using python TensorFlow packages to detect and classify the location of the drone. A bounding box with four coordinates is used to generate the location within the image. 90% of the data set is used for training and 10% of the data set is used for testing.

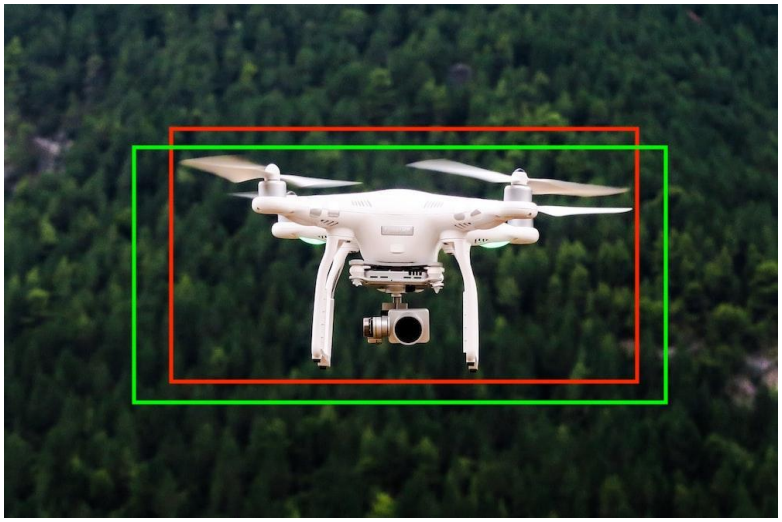


Figure 1: Drone image example with bounding box, red is input bounding box, green is the bounding box generated by CNN

We used Keras from TensorFlow to build our neural networks model. Here is how we designed the neural network which has convolutional layers, pooling layers, followed by a flatten layer to flatten the input. After that, we have four fully-connected layers.

In the beginning of the model, we utilized a pre-trained convolutional neural network model called VGG 16 which contains a bunch of layer sets of two convolutional layers and one pooling layer. After several iterations of convolution and pooling, we implemented a flatten layer to convert the data into a 1-dimensional array for inputting it to the next layer. Last, we also introduced four fully-connected layers which made sure all possible connections layer-to-layer were present, meaning every input of the input vector influences every output of the output vector. You can see the summary below that the activation function of the first three fully-connected layers is ReLU, which has 128, 64, 32 neurons respectively. In the last layer of our model, we have a fully-connected layer consisting of 4 neurons and with an activation function of Linear.

```
In [11]: # Import tensorflow.keras model to build neural networks.
         from tensorflow.keras.layers import Dense,Flatten
         from tensorflow.keras import Model,Input

         # Deep Convolutional Networks for Large-Scale Image Recognition.
         def build_model():
             # Instantiates the VGG16 pretrained model.
             # Set value for the pre-trained weights,input_tensors.
             # Include_top : we do not have 3 fully-connected layers at the top of the network.
             vgg = tf.keras.applications.vgg16.VGG16(weights="imagenet", include_top=False, input_tensor=Input(shape=(258, 258, 3)))

             vgg.trainable = True
             # vgg output : Flatten the high dimension.
             flatten = vgg.output
             flatten = Flatten()(flatten)
             # Build the 4 nerual networks layers.
             # Build the first layer with 128 Neurons, the activation function is relu.
             # Build the second layer with 64 Neurons, the activation function is relu.
             # Build the third layer with 32 Neurons, the activation function is relu.
             # Build the last layer with 4 Neurons, the activation function is linear.
             # Output would be 4 indexs of bounding box.
             bboxHead = Dense(128, activation="relu")(flatten)
             bboxHead = Dense(64, activation="relu")(bboxHead)
             bboxHead = Dense(32, activation="relu")(bboxHead)
             bboxHead = Dense(4, activation="linear")(bboxHead)
             # Generate Model
             model = Model(inputs=vgg.input, outputs=bboxHead)

             return model
```

Figure 2: Codes for model building

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 258, 258, 3)]     0

 block1_conv1 (Conv2D)       (None, 258, 258, 64)      1792

 block1_conv2 (Conv2D)       (None, 258, 258, 64)      36928

 block1_pool (MaxPooling2D)  (None, 129, 129, 64)      0

 block2_conv1 (Conv2D)       (None, 129, 129, 128)     73856

 block2_conv2 (Conv2D)       (None, 129, 129, 128)     147584

 block2_pool (MaxPooling2D)  (None, 64, 64, 128)       0

 block3_conv1 (Conv2D)       (None, 64, 64, 256)       295168

 block3_conv2 (Conv2D)       (None, 64, 64, 256)       590080

 block3_conv3 (Conv2D)       (None, 64, 64, 256)       590080

 block3_pool (MaxPooling2D)  (None, 32, 32, 256)       0

 block4_conv1 (Conv2D)       (None, 32, 32, 512)       1180160

 block4_conv2 (Conv2D)       (None, 32, 32, 512)       2359808

 block4_conv3 (Conv2D)       (None, 32, 32, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 16, 16, 512)       0

 block5_conv1 (Conv2D)       (None, 16, 16, 512)       2359808

 block5_conv2 (Conv2D)       (None, 16, 16, 512)       2359808

 block5_conv3 (Conv2D)       (None, 16, 16, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 8, 8, 512)         0

 flatten (Flatten)           (None, 32768)             0

 dense (Dense)               (None, 128)               4194432

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 4)                 132

=================================================================
Total params: 18,919,588
Trainable params: 18,919,588
Non-trainable params: 0
```

Figure 3: Model Summary

The final step in creating our neural network model is compilation. After completing this step, we can move on to the training phase. During this procedure, we defined the loss function, the optimizer, and the metrics. We would like to talk about each defined parameter separately. For optimizer, we implemented a default Adam algorithm which is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. For loss function, we chose mean squared error (MSE) as our lost function. Using this loss function, we could measure how close a regression line is to a set of data points. It is a risk function corresponding to the expected value of the squared error loss. Mean square error is calculated by taking the average, specifically the mean, of errors squared from data as it relates to a function. We still have to implement a metrics function to judge the performance of our model. Due to the nature of our model, we used Accuracy metrics which could calculate how often predictions equal labels. Finally, we defined a callback API to save the Keras model or model weights at some frequency. We set “val_loss” to monitor the model’s total loss, verbose to 1 to display messages when the callback takes an action. We also set “save_best_only” to 1 which means that it only saves when the model is considered

“best” and the latest best model according to the quantity monitored will not be overwritten. This call back function would be used in conjunction with training using “model.fit()” later.

```
In [14]: # Compile the model with optimizer.

         # Set learning_rate, loss function, accuracy metrices.

         model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001), loss='mse', metrics=['accuracy'])

         # Define the CallBack API to save the Keras model or model weights.

         # Set the "val_loss" to monitor the model's total loss.

         # Only saves when the model is considered the "best" and the latest best model according to the quantity monitored
         # will not be overwritten.

         # Verbose:Verbosity mode 1 displays messages when the callback takes an action.

         save_best = tf.keras.callbacks.ModelCheckpoint("Model.h5",monitor='val_loss',save_best_only=True, verbose=1)
```

Figure 4: Model Compilation

After all the parameters have been set, we can start training our model. We used train images and train targets to train for a fixed number of epochs. To make the training result better, we assigned 20% training data to be used as validation data by setting “validation_spilt = 0.2”. We also set the “batch_size = 16” which means that there are 16 samples processed before the model is updated. Also, we chose “epochs = 50” to make sure we can go through our training set 50 times. As we also want to see an animated progress bar, so we set the “verbose = 1”. “CallBack” made sure we only saved the best results.

```
In [18]: # Training Model

         # Validation_split was set to 0.2 to shuffles the data into 80% tarining data and 20% validation data.

         # Callback set as the save_best we defined before.

         # Set batch_size to 16 for each round.

         # Set epoches equals to 50 training round.
         model.fit(trainImages, trainTargets, validation_split=0.2, batch_size= 16, epochs=50, verbose=1, callbacks=[save_best])
```

```
62/62 [==============================] - ETA: 0s - loss: 0.0106 - accuracy: 0.8108
Epoch 41: val_loss did not improve from 0.01841
62/62 [==============================] - 412s 7s/step - loss: 0.0106 - accuracy: 0.8108 - val_loss: 0.0200 - val_accuracy:
0.7102
Epoch 42/50
62/62 [==============================] - ETA: 0s - loss: 0.0108 - accuracy: 0.8088
Epoch 42: val_loss improved from 0.01841 to 0.01811, saving model to Model.h5
62/62 [==============================] - 414s 7s/step - loss: 0.0108 - accuracy: 0.8088 - val_loss: 0.0181 - val_accuracy:
0.7102
Epoch 43/50
62/62 [==============================] - ETA: 0s - loss: 0.0096 - accuracy: 0.8170
Epoch 43: val_loss did not improve from 0.01811
62/62 [==============================] - 414s 7s/step - loss: 0.0096 - accuracy: 0.8170 - val_loss: 0.0198 - val_accuracy:
0.7020
Epoch 44/50
62/62 [==============================] - ETA: 0s - loss: 0.0093 - accuracy: 0.8313
Epoch 44: val_loss did not improve from 0.01811
62/62 [==============================] - 412s 7s/step - loss: 0.0093 - accuracy: 0.8313 - val_loss: 0.0187 - val_accuracy:
0.7224
Epoch 45/50
```

Figure 5: Training Model

In our Model we used the VGG16 pretrained Convolutional Network model for data classification and detection. The following diagram shows the hig level Architecture for VGG16 Model.

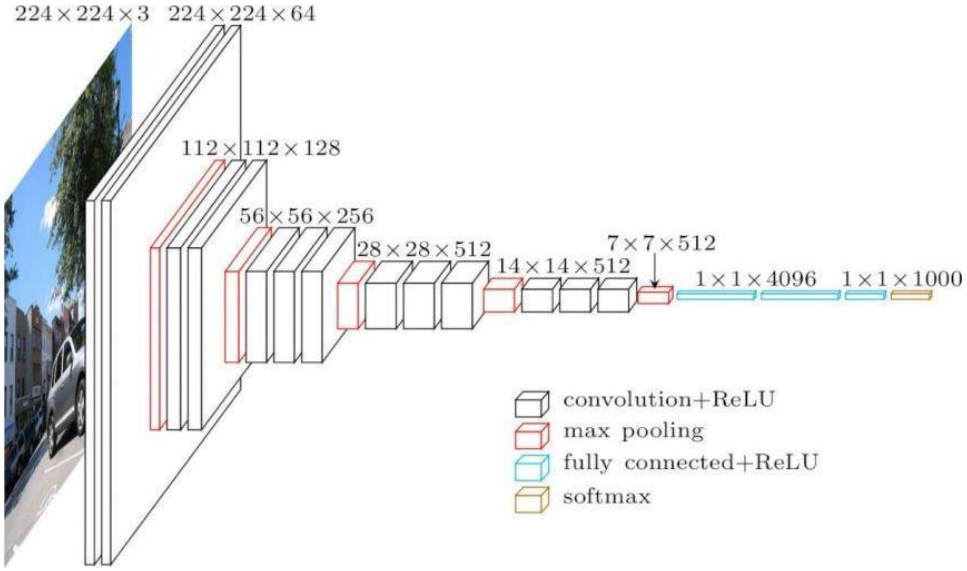


Figure 6: High level architecture of VGG16

The VGG16 framework consists of a volume base layer, a pooling layer, and a fully connected layer. The input to cov1 layer is of fixed size 224 x 224 RGB image (In our project, the Image shape is (258,258,3)). The image is passed through a stack of convolutional (conv.) layers, where the filters were used with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations, it also utilizes 1×1 convolution filters, which can be seen as a linear transformation of the input channels. The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling layer is performed over a 2×2-pixel window, with stride 2. The output for the VGG16 model based on the TensorFlow framework is (7,7,512) shape matrix which is the feature of the input picture. The Fully Connected layers (FC) need to be designed by ourselves. Before importing into FC layers, we need to flatten the output data from VGG 16 into 1 dimension array. The following shows how Max pooling layers treat features collection and dimensionality reduction:

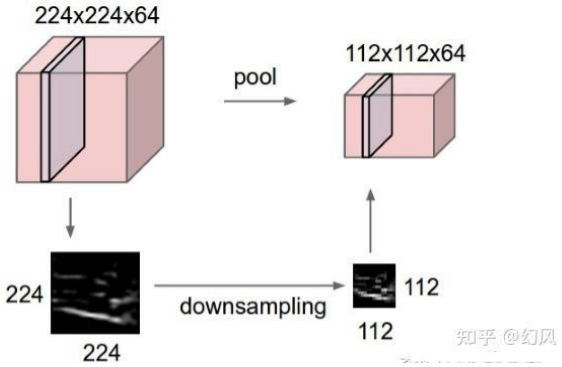


Figure 7: Pooling layers and dimensionality reduction

The Fully Connected layers follow a stack of convolutional layers (which we already flattened): In the fully connected layer all neurons are connected with weights, and usually the fully connected layer is at the tail of the convolutional neural network. Once the previous convolutional layer has captured enough features to be used to recognize the image, the next step is how to perform the classification. Usually, the end of the convolutional network will flatten the rectangle obtained at the end into a long vector and send it to the fully-connected layer with the output layer for classification.

The input drone dataset we used contains 11,920 jpg files and txt file pairs. Each picture with drone corresponds to a txt file, which included the true value for four bounding box indexes for the drone within this picture. The details of the bounding box have been defined above. The shape of each input image is (258,258,3) and our goal is to use VGG16 model to predicted and detect where the drone is by predicting four indexes for the bounding box.

Results

The predicted data generated by the data are the XY coordinates of the bounding box. The results are compared to the actual coordinates using the L2 norm method as shown in the equation $Accuracy = \sqrt{(x_{pred} - x_{actual})^2 + (y_{pred} - y_{actual})^2}$. The default model consists of the convolutional neural network of VGG, four fully connected layers with hidden layers of ReLu and the linear output layer, and a learning rate of 1e-3. While the default model has an accuracy of 48.35%, the team explored possible methods that could improve the accuracy in theory.

Numbers of layers

A model that learns the training dataset too well or insufficiently performs poorly on new data. The problem of overfitting and underfitting can be addressed by changing the number of layers. Therefore, the team investigated the number of layers in a fully connected network to make a good fit for the model. In the experiment, the number of layers ranges from 2-5 layers, with the result as shown in the table. Higher accuracy means that the model is fitted better. So, it is concluded that the optimal number of layers is three. A higher number of layers than three results in overfitting, and lower numbers of layers are underfitting.

Table 1: Accuracy versus number of layers

Number of Layers	2	3	4	5
Accuracy	47.42%	50.3%	48.35%	39.08%

Learning Rate

The learning rate is a parameter that controls how much to change the model in response to the estimated error each time the model weights are updated. While maintaining the other variables' same values, a too small learning rate results in the inability to reach the converge

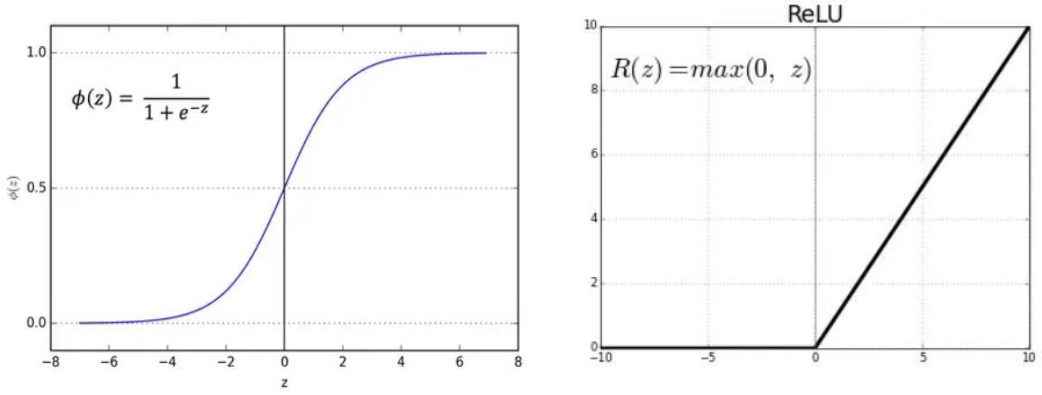
point, and a too large learning rate results in a sub-optimal set of weights due to the unstable training process. Therefore, the team investigated the learning rate running from 1e-5 to 1e-3, as shown in the table. The optimal learning rate is 1e-4.

Table 2: Accuracy versus learning rate

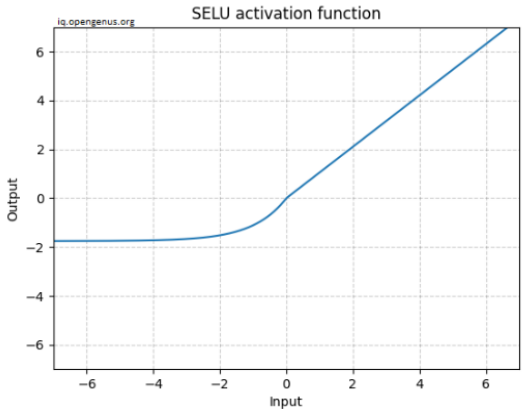
Learning rate	1e-3	1e-4	1e-5
Accuracy	48.35%	70.3%	39.08%

Hidden layer activation function

An activation function determines whether the neuron's input to the network is important or not using mathematical operations. The primary role of hidden layers is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output. Some existing activation functions include ReLU, sigmoid, and SeLu. ReLU replaces negative values with zero and leaves positive values unchanged, as shown in Figure 8(a). Sigmoid is a nonlinear function and can be used to define nonlinear boundaries, as shown in Figure 8(b). SELU solved the "Dying ReLU Problem" as it doesn't treat large and small negative numbers the same as shown in Figure 8(c). The table shows the accuracy using different methods, and ReLU has the best accuracy. ReLu should be much better than sigmoid since sigmoid has the vanishing gradient problem, which has the output saturate at 1 with an extremely large input. Also, it is noted that simply replacing ReLU with SELU does not improve the accuracy since ReLU converges much faster than SeLU.



(a) **(b)**



(c)

Figure8: Activation Methods

Table 3: Accuracy versus activation functions

Activation Method	Sigmoid	ReLU	SELU
Accuracy	24.61%	48.45%	41.89%

Discussion

According to the experiment, the model consisted of the convolutional neural network of VGG, three fully connected layers with hidden layers of ReLU and the linear output layer, and a learning rate of 1e-4 should have the best performance. However, even though the accuracy result of 57.71% is better than that of the base model, it is worse than several previous models. Therefore, simply combining all features that work the best doesn't generate the best accuracy. More investigations need to be done to create the model with the best result.

References

Ollis, Nicholas. "Implementing Swish Activation Function in Keras." Big Nerd Ranch, October 19, 2021. https://bignerdranch.com/blog/implementing-swish-activation-function-in-keras/.

"Giving Activation in Dense Layer Keras with Code Examples." Programming and Tools Blog. Accessed December 6, 2022. https://www.folkstalk.com/tech/giving-activation-in-dense-layer-keras-with-code-examples/.

Rosebrock, Adrian. "Object Detection: Bounding Box Regression with Keras, Tensorflow, and Deep Learning." PyImageSearch, April 17, 2021. https://pyimagesearch.com/2020/10/05/object-detection-bounding-box-regression-with-keras-tensorflow-and-deep-learning/.

Hassan, Muneeb ul. "VGG16 - Convolutional Network for Classification and Detection." VGG16 - Convolutional Network for Classification and Detection, February 24, 2021. https://neurohive.io/en/popular-networks/vgg16/.

Dommaraju, Goutham. "Keras' Accuracy Metrics." Medium. Towards Data Science, May 20, 2020. https://towardsdatascience.com/keras-accuracy-metrics-8572eb479ec7.