



DETECTION OF AIRCRAFT USING CONVOLUTIONAL NEURAL NETWORK

B. Tech Project-1 Report



BY- UJJWAL KUMAR (160040086)

SUPERVISOR- PROF. J. INDU

DEPARTMENT OF CIVIL ENGINEERING

IIT BOMBAY

CONTENTS:

ABSTRACT	2
INTRODUCTION	2
METHODOLOGY	4
TEST DATASETS USED	10
RESULTS	11
LIMITATIONS AND ASSUMPTIONS	12
CONCLUSIONS.....	13
SCOPE FOR FUTURE WORK	14
References	15

ABSTRACT:

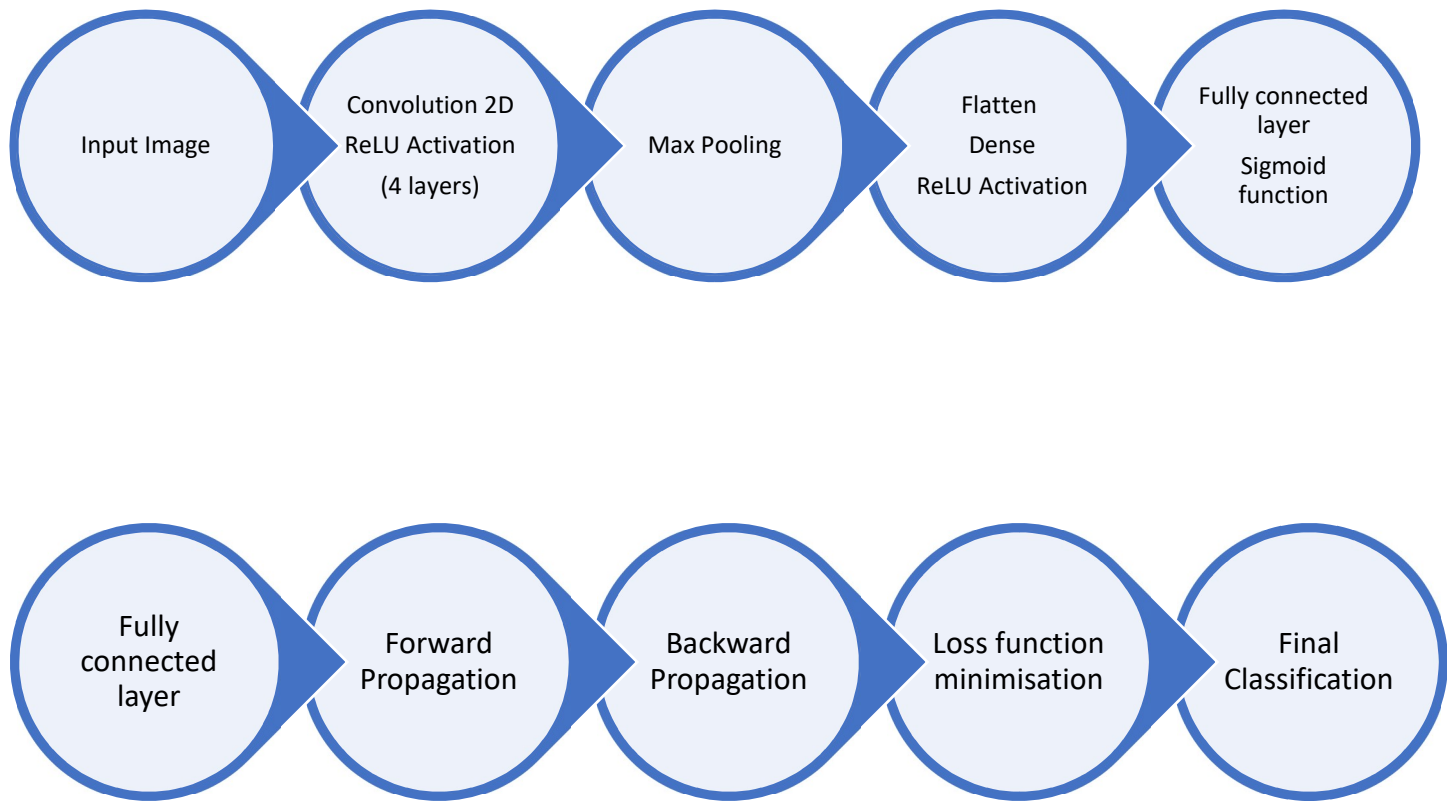
To address the issue of poor accuracy and slow execution time in detection of aircraft encountered in traditional methods, this project is based on using a CNN structure for training and detecting objects (aircraft in this case). The main challenge for detecting aircraft images is the background which is similar to aircraft and the altitude from which images are captured which makes aircrafts look very small. So, to cater to these challenges we use multifeatured fusion convolutional neural network. In this the shallow and deep layer features are fused at same scale after sampling to overcome the problems of low dimensionality in deep layers and the inadequate details of small objects. To increase the speed of training and detection for validating, we preprocess data with mean normalization ($x_i = \frac{x_i - \mu}{\text{range of } x_i}$, μ is mean of x_i) and feature scaling ($x_i = \frac{x_i}{\text{range of } x_i}$). Additionally, the training of network was carried out on data sets with different spatial resolutions and also images captured from different altitude.

INTRODUCTION:

Detection of objects in an image has been area of research for over many decades and among these, detection of aircraft still remains a challenging task because of the complex background, illumination change and variation of aircraft kind. The detection still needs to be carried out for military and security purpose. Various different approach has been carried out for aircraft detection starting from using filter with support vector or coarse-to-fine edge detection, using directional gradient and so on. But these methods use low-level features and thus has high false alarm. With the introduction of neural networks works shifted towards using supervised training approach and deep learning methods. Running deep networks on sliding windows for object proposal proved to be accurate than previous approaches but was very slow in execution. To increase the speed, detection based on spatial pyramid pooling method was adopted but the issue of slow candidate region proposal network still remained. Later the concept of using convolutional neural network along with region proposal network was introduced which proved to be very accurate and also reducing the runtime of the execution. The key requirements of using a convolutional structure is that as it involves so many weight parameters so the training data set needs to be very accurate and present in large number. To compensate for large training data set, data augmentation becomes a crucial aspect in this method. Also, the hidden structure of CNN doesn't follow a fixed pattern so finding the appropriate CNN hidden layers varies from the targeted purpose and the kind of dataset that we have. In this project, the method of implementing an entire CNN structure for detecting aircraft is being carried out. In this project a 4 layered fully convolutional network was implemented, trained and tested on 32000 datasets taken from Kaggle.

METHODOLOGY:

Overall structure of CNN Network



The dimension of each image is $20 \times 20 \times 3$ and a total of 32000 images were used. First, we input each image and store the output (aircraft present or not) which in this case is 0(not present) or 1(aircraft present). The naming convention of each image was first character being 0/1 followed by name where 0/1 represented the output of the image. In this the number of filters was increased after each layer. In the first step, 4 successive convolution & ReLU activation layer was computed with depth of layers being (20,44,68,92) and kernel size being fixed as 3×3 . The dimension of next layer is computed using $\frac{N-F+2P}{S} + 1$ for each X, Y direction where N-dimension size, F-kernel size, P-padding size, S-stride. In this case stride value was kept 1 and no padding was done. The dimension only changes after each convolution layer due to kernel filter applied, activation function just computes value of given input and output remains of same dimension. Activation function is applied to avoid overfitting and, in this case, ReLU computes $\max(0, x)$. Then *maxpooling* of dimension 2×2 filter layer is applied to downsample the data along row, column dimension. Then we flatten the layer (converting it to

a column vector) and then apply dense function (which connects all the convolution weights directly to given number of classes) to a 120 layered output, apply ReLU and finally apply dense to 1 class output and apply sigmoid activation function ($\frac{1}{1+e^{-\theta^T x}}$). If the value is >0.5 we classify the image as containing aircraft (value is set to 1) else a non-aircraft image is found.

```

jupyter BTPCNN Last Checkpoint: 10/07/2019 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [15]: def cnn(size, n_layers):
    MIN_NEURONS = 20
    MAX_NEURONS = 120
    KERNEL = (3, 3)

    # Determine the # of neurons in each convolutional layer
    steps = np.floor(MAX_NEURONS / (n_layers + 1))
    nuerons = np.arange(MIN_NEURONS, MAX_NEURONS, steps)
    nuerons = nuerons.astype(np.int32)

    model = Sequential()

    for i in range(0, n_layers):
        if i == 0:
            shape = (size[0], size[1], size[2])
            model.add(Conv2D(nuerons[i], KERNEL, input_shape=shape))
        else:
            model.add(Conv2D(nuerons[i], KERNEL))

        model.add(Activation('relu'))

    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(MAX_NEURONS))
    model.add(Activation('relu'))

    model.add(Dense(1))
    model.add(Activation('sigmoid'))

    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    model.summary()

    return model

```

Figure 1 - Code building CNN architecture

```

In [12]: # Number of positive and negative examples to show
N_TO_VISUALIZE = 10

positive_example_indices = (y_train == 1)
positive_examples = x_train[positive_example_indices, :, :]
positive_examples = positive_examples[0:N_TO_VISUALIZE, :, :]

negative_example_indices = (y_train == 0)
negative_examples = x_train[negative_example_indices, :, :]
negative_examples = negative_examples[0:N_TO_VISUALIZE, :, :]

visualize_data(positive_examples, negative_examples)

```

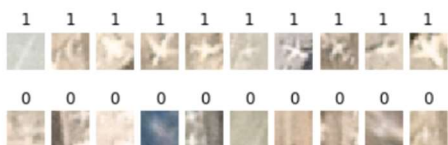


Figure 2 - Visualising the data set

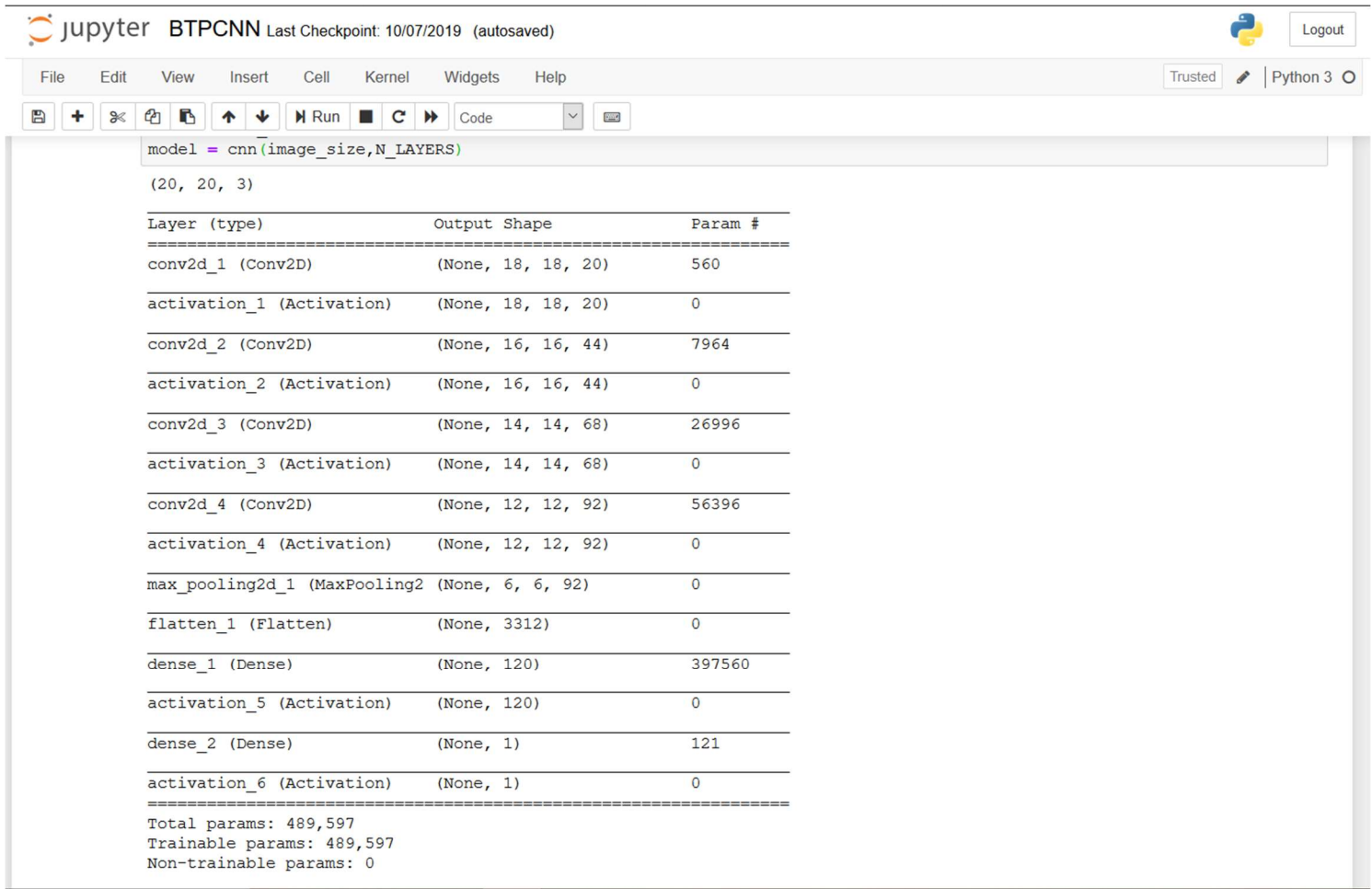


Figure 3 - Parametric variation in CNN architecture

Mathematical Computation from Fully Connected Layer to Final Output Layer:

It has majorly 3 components: Forward computation, Backward computation and defining loss function and minimizing it. These steps are exactly same as done in neural networks just that the input layer in this is the convolved fully connected output layer.

Pseudo code for backpropagation:

initialize network weights (with some random small values)

do

for Each training example named ex

prediction = neural network output (network, ex)

actual = known output (ex)

compute error (prediction - actual) at the output units

compute {Delta $w_{\{h\}}$ } for all weights from hidden layer to output layer //backward pass

compute {Delta $w_{\{i\}}$ } for all weights from input layer to hidden layer //backward pass

update network weights // except the input layer

until all examples classified correctly or any custom defined stopping criteria

return the network

The Loss function J is generally taken as Euclidean distance function and accordingly the weight parameters are updated. There are various ways to minimise the loss function. In case of CNN the normal gradient based approach doesn't seem to work because the loss function may get stuck in local minima, so to avoid this we generally used Stochastic Gradient Descent with Momentum term optimisation algorithm or some higher end algorithm such as Adagrad, RMS Prop or Adams. In this Adams was used because rest other optimisation algorithm contains a hyperparameter (similar to learning rate) and finding the optimised value of that hyperparameter again either becomes another optimisation problem or using hit and trial method becomes computationally expensive.

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^m (h(x)_i - y_i)^2 \text{ where } h(x)_i \text{ is predicted \& } y_i \text{ is known value}$$

Layer wise computation:

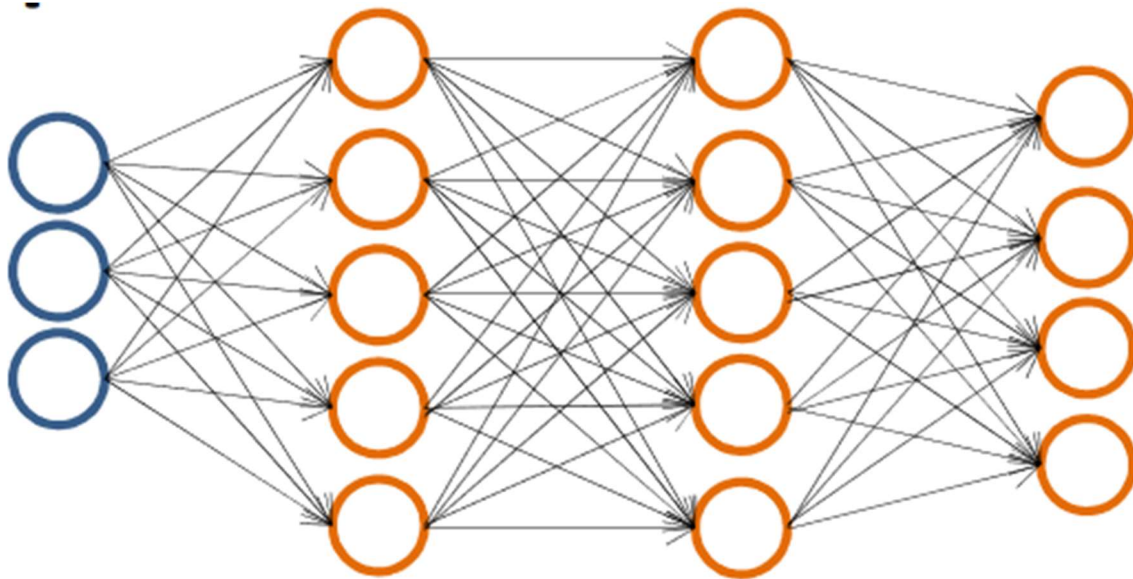


Figure 4 - Assumed architecture for mathematical computation

Assuming each layer is named $a^{(1)}, a^{(2)}, a^{(3)}, a^{(4)}$ where $a^{(1)}$ is input layer $a^{(4)}$ is output layer and rest are hidden layers connected with each other; θ^n connects layer n to $n+1$

Forward propagation equations:

$$a^{(1)} = x \text{ where } x \text{ is input layer}$$

$$z^{(2)} = \theta^{(1)} * a^{(1)} \Rightarrow a^{(2)} = g(z^{(2)}) \left[\text{add } a_0^{(2)} \text{ to } a^{(2)} \text{ as bias unit} \right] \rightarrow g(x) - \text{sigmoid of } x$$

$$z^{(3)} = \theta^{(2)} * a^{(2)} \Rightarrow a^{(3)} = g(z^{(3)}) \left[\text{add } a_0^{(3)} \text{ to } a^{(3)} \text{ as bias unit} \right]$$

$$z^{(4)} = \theta^{(3)} * a^{(3)} \Rightarrow a^{(4)} = g(z^{(4)}) = h_{\theta}(x)$$

Backpropagation equations:

$\delta_j^{(l)} \rightarrow$ error of node j in layer l

$$\delta^{(4)} = a^{(4)} - y$$

$$\delta^{(3)} = (\theta^{(3)})^T * \delta^{(4)} * g'(z^{(3)}) \quad . * \text{ is element wise multiplication}$$

$$\delta^{(2)} = (\theta^{(2)})^T * \delta^{(3)} * g'(z^{(2)}) \quad g'(z^{(n)}) = a^{(n)} * (1 - a^{(n)})$$

No $\delta^{(1)}$ term as input layer remains the same

Updating parameter equation:

for $i = 1:m \leftarrow (x^i, y^i)$

$$a^{(1)} = x^i$$

forward computation to compute a^l for $l = 2, 3, 4 \dots L$

using y^i compute $\delta^{(L)} = a^{(L)} - y^i$

compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots \delta^{(2)}$

$$\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

end

$$D_{ij}^{(l)} = \frac{1}{m} D_{ij}^{(l)} + \frac{\lambda}{m} \theta_{ij}^{(l)} \quad \text{if } j \neq 0$$

$$D_{ij}^{(l)} = \frac{1}{m} D_{ij}^{(l)} \quad \text{if } j = 0$$

Finally, minimization of loss function $J(\theta)$ was done using 'Adams' optimization algorithm and loss was calculated in terms of binary cross entropy loss.

$$J(\theta) = -\frac{1}{m} [\sum_{i=1}^m y^i * \log(h_{\theta}(x^i)) + (1 - y^i) * \log(1 - (h_{\theta}(x^i)))]$$

where $(h_{\theta}(x^i))$ is predicted value of i_{th} dataset and y^i is the known value of i_{th} dataset

Training the network:

For training the entire dataset is split into 70:30 where 70% of dataset is used for training and rest for validating. 70% of dataset is selected using random permutation. Epoch (maximum iteration) of 150 was selected and Batch size of 200. To avoid overfitting termination condition was set using 'patience' function. For no change in delta for 10 iteration would lead to termination of loop. The training ended after 53 iterations and for each iteration it took about 1 minute. The training dataset went up to a maximum of 99.82% while the validation accuracy achieved was about 75%. The training was performed on only CPU computation mode as GPU computational supported device was not available.


```
localhost:8888/notebooks/Desktop/CNN/BTPCNN.ipynb
Jupyter BTPCNN Last Checkpoint: 10/07/2019 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

In [28]: EPOCHS = 150
         BATCH_SIZE = 200

In [29]: PATIENCE = 10
         early_stopping = EarlyStopping(monitor='loss', min_delta=0, patience=PATIENCE, verbose=0, mode='auto')

In [30]: LOG_DIRECTORY_ROOT = ''
         now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
         log_dir = "{}{}run-{}".format(LOG_DIRECTORY_ROOT, now)
         tensorboard = TensorBoard(log_dir=log_dir, write_graph=True, write_images=True)

In [31]: callbacks = [early_stopping, tensorboard]

In [23]: # Train the model
         model.fit(x_train, y_train, epochs=EPOCHS, batch_size=BATCH_SIZE, callbacks=callbacks, verbose=1)
Epoch 45/150
22400/22400 [=====] - 54s 2ms/step - loss: 0.0067 - acc: 0.9977
Epoch 46/150
22400/22400 [=====] - 56s 2ms/step - loss: 0.0080 - acc: 0.9971
Epoch 47/150
22400/22400 [=====] - 51s 2ms/step - loss: 0.0058 - acc: 0.9980
Epoch 48/150
22400/22400 [=====] - 51s 2ms/step - loss: 0.0078 - acc: 0.9972
Epoch 49/150
22400/22400 [=====] - 51s 2ms/step - loss: 0.0062 - acc: 0.9982
Epoch 50/150
22400/22400 [=====] - 52s 2ms/step - loss: 0.0047 - acc: 0.9981
Epoch 51/150
22400/22400 [=====] - 55s 2ms/step - loss: 0.0074 - acc: 0.9975
Epoch 52/150
22400/22400 [=====] - 52s 2ms/step - loss: 0.0090 - acc: 0.9970
Epoch 53/150
22400/22400 [=====] - 51s 2ms/step - loss: 0.0080 - acc: 0.9972

Out[23]: <keras.callbacks.History at 0x18aef01fd0>
```

Figure 5 - Training dataset code with early call back function

RESULT:

The overall accuracy on the validation data set obtained was around 98.5% while the accuracy on training data set was around 75.32%. The plots of accuracy and loss versus epochs for both training data and validation data set was plotted to visualise the sections where the model may require improvements to increase accuracy.

```
In [64]: # Make a prediction on the test set
         test_predictions = model.predict(x_test)
         test_predictions = np.round(test_predictions)

In [65]: # Report the accuracy
         accuracy = accuracy_score(y_test, test_predictions)
         print("Accuracy: " + str(accuracy))

         Accuracy: 0.7530208333333334

In [66]: model.save("BTPCNN_Save")
```

Figure 6 - Predicting and obtaining accuracy

```
In [44]: # Plot training & validation loss values
plt.plot(history.history['loss'])
#plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

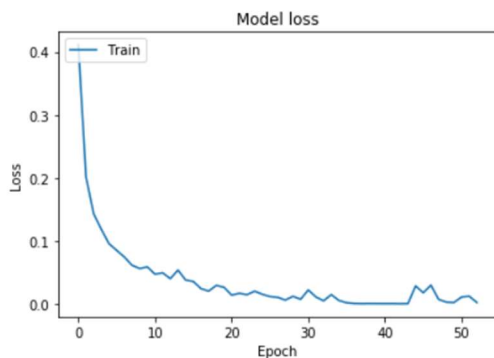


Figure 7 - Plot of training loss vs Iteration

We see that as the iteration increases the overall loss goes on increasing. This is due to the fact that initially the weight network is randomly initialised but with increasing data set it gets updated and eventually converges towards global minima value.

```
In [45]: # Plot training & validation accuracy values
plt.plot(history.history['acc'])
#plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

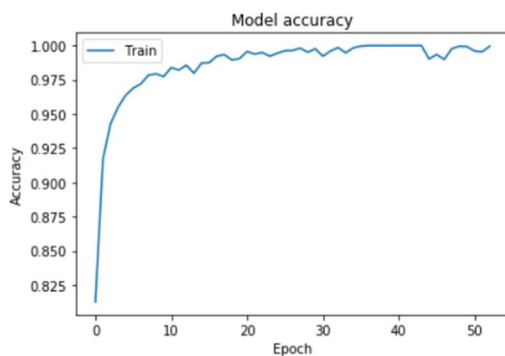


Figure 8 - Plot of training accuracy vs Iteration

The interpretation of accuracy becomes obvious after the loss function trend. With the decreasing loss the overall training accuracy goes on increasing.

It is seen that the overall accuracy of training dataset is very high while it is not the case with validation dataset. This could imply that the network is overfitting the training dataset and we might require a more complex network architecture.

LIMITATIONS & FUTURE IMPROVEMENTS:

The overall training accuracy is about 99.4% while the predicting accuracy obtained in this is around 75% but it can definitely be increased further. There are two major areas which could increase predicting accuracy significantly, 1st being the network structure itself is quite simple and small as it involves only around 0.5M parameters while networks like VGGNet has around 150M parameters. So, increasing the complexity and depth of network should definitely increase the predicting accuracy. Second reason is the dataset itself. Though the number of dataset is around 32000 but only around 20% of it contains images of aircraft so the network generalisation fails due to this and also the resolution of each dataset is only about 20*20*3 which is too small. A better generalised dataset with higher resolution and data augmentation applied on the data set should give a better trained model which would give a significantly higher accuracy.

As this model only predicts whether the image contains aircraft or not but this can be extended further to draw a bounding box around the location where image has the aircraft and also predict whether image contains multiple aircraft images or not. This can be achieved by using sliding window approach, where once it is identified that the image contains aircraft then a window of varying dimension could slide across the entire image generating probable regions where aircraft might be present and if the probability of occurrence of that region is greater than some threshold value then a box could be highlighted across those coordinates signifying the position of aircraft. Another possible approach for this could be using a dataset which has image and coordinates of the box around the image and using these types of data the network can learn to identify the aircraft as well as where to draw the bounding box.

CONCLUSION:

CNN architecture does provide a more accurate solution for identification of objects in images and in this case aircrafts. The biggest challenge is the variation in types of images for an aircraft that is variation in terms of altitude from which image is taken, variation of resolution of image and the background involvement. Although training the model does take some time but it's time can be decreased by using GPU computation. Post training, the prediction takes very little time. This method is definitely quite accurate, fast and the model once trained its weight can be saved and a predicting software GUI can be made out of it.

BIBLIOGRAPHY:

1. X. Li, S. Wang, B. Jiang and X. Chan, "Airplane detection using convolutional neural networks in a coarse-to-fine manner," 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Chengdu, 2017, pp. 235-239. doi:10.1109/ITNEC.2017.8284943
2. Xu, Yuelei et al. "Rapid Airplane Detection in Remote Sensing Images Based on Multilayer Feature Fusion in Fully Convolutional Neural Networks." *Sensors* (Basel, Switzerland) vol. 18,7 2335. 18 Jul. 2018, doi:10.3390/s18072335
3. Chen, Xueyun, Shiming Xiang, Cheng-Lin Liu and Chunhong Pan. "Aircraft Detection by Deep Convolutional Neural Networks." *IPSJ Trans. Computer Vision and Applications* 7 (2014): 10-17.
4. Li, Yang; Fu, Kun; Sun, Hao; Sun, Xian. 2018. "An Aircraft Detection Framework Based on Reinforcement Learning and Convolutional Neural Networks in Remote Sensing Images." *Remote Sens.* 10, no. 2: 243.
5. Xu T.B., Cheng G.L., Yang J. Fast Aircraft Detection Using End-to-End Fully Convolutional Network; Proceedings of the 2016 IEEE International Conference on Digital Signal Processing (DSP); Beijing, China. 16–18 October 2016
6. Zhang W., Sun X., Fu K., Wang C., Wang H. Object detection in high-resolution remote sensing images using rotation invariant parts based model. *IEEE Trans. Geosci. Remote Sens.* 2014;11:74–78. doi: 10.1109/LGRS.2013.2246538.
7. Ren S., He K., Girshick R.B. Object Detection Networks on Convolutional Feature Maps. *IEEE Trans. Pattern Anal. Mach. Intell.* 2017;39:1476–1481. doi: 10.1109/TPAMI.2016.2601099.
8. Dataset obtained from Kaggle - <https://www.kaggle.com/rhammell/planesnet/data>