

CNN_on_CIFAR

December 6, 2021

0.1 Define a class to handle all functions

```
[1]: class cifar():
    def __init__(self,filepath):
        """ This initialises the filepath to the directory where the files are
        ↪present """
        self.filepath=filepath
        self.batch1=None
        self.batch2=None
        self.map=dict()
        self.map_labels()
        self.n_labels=len(self.map)
        self.model=None

    def load_best_model(self):
        """ The function will load the model from the folder thus making it
        ↪easier to do predictions and testing
        """
        import pickle
        self.model=pickle.load(open('cnnModel81.sav','rb'))

    def test_prediction(self,array):
        """ The function takes as input an image, does the prediction
        and the prints the image along with the predicted label thus helping to
        ↪make testing easier

        If the model has to be changed to a custom model, do
        cf.model=<newmodel>
        to restore the best model do
        cf.load_best_model()
        """
        self.plot(array,cf.label_decode(np.argmax(self.model.predict(array.
        ↪reshape(1,32,32,3)))))

    def map_labels(self):
        """ The function reads the batches.meta file and converts the lists and
        ↪labels to a dictionary for
```

```

        easy access later """
        content=self.unpickle('batches.meta')
        content=content[b'label_names']
        for i,number in enumerate(content):
            self.map[i]=number.decode('utf-8')

    def label_decode(self,id):
        """ Returns the Label name of the ID which is being passed into the
        ↪function"""
        return self.map[id]

    def unpickle(self,filename):
        """ The function takes a filename and returns the dictionary of data
        ↪inside the file.
            The data is pickled and stored in the files and hence this function
        ↪helps to unpack
            the data.

            Usage : data=unpickle('batches.meta')
            Return : Dictionary of data
            Data inside the dictionary can be explored using data.keys and
        ↪other dictionary functions """

        filepath=self.filepath+filename
        import pickle
        with open(filepath,'rb') as foo:
            dict=pickle.load(foo,encoding='bytes')
        return dict

    def convert_to_rgb(self,array):
        """ The function takes a flattened array as input and converts it into
        ↪a RGB Image.
            It normalises the image by dividing by 255 and then stacks it depth wise
        ↪to get an RGB Image. """

        import numpy as np
        r=array[0:1024].reshape(32,32)/255
        g=array[1024:2048].reshape(32,32)/255
        b=array[2048:].reshape(32,32)/255
        im=np.dstack((r,g,b))
        return im

    def read_batch_data(self,id):
        """ Function takes as input the number of the batch file to be read

```

```

        It then reads the data and corresponding label of the data from the
→file
        and then converts it into RGB images.
        The function return the images and labels which can later be used
→for training
        or validation.
        """
        import numpy as np
        filename='data_batch_'+str(id)
        data=self.unpickle(filename)
        y=data[b'labels']
        x=data[b'data']
        images=[]
        for image in x:
            images.append(self.convert_to_rgb(image))
        return np.array(images),np.array(y)

    def read_test_batch(self):
        """ The function already has a set of testing data file. Calling this
→function would
        read the corresponding testing file and would then convert the data
→into images and labels
        in the same way as in the read-batch-data function.
        The function then returns the images and its associated labels which
→are later used as test data
        during the process
        """
        import numpy as np
        filename='test_batch'
        data=self.unpickle(filename)
        y=data[b'labels']
        x=data[b'data']
        images=[]
        for image in x:
            images.append(self.convert_to_rgb(image))
        return np.array(images),np.array(y)

    def OHE(self,list_of_values):
        """ Since the output columns are label encoded, invoking the OHE
→function
        would convert the label encoded outputs to One Hot Encoded Outputs and
→returns the same
        The function takes as input a list of the labels
        """
        import numpy as np

```

```

        result=np.zeros((len(list_of_values),self.n_labels))
        for i in range(len(list_of_values)):
            result[i][list_of_values[i]]=1
        return result

    def plot(self,array,label):
        """ A helper function to plot an image """
        import matplotlib.pyplot as plt
        print(label)
        plt.imshow(array)

```

```

[5]: from google.colab import drive
      drive.mount('/gdrive')
      %cd /gdrive

```

Drive already mounted at /gdrive; to attempt to forcibly remount, call drive.mount("/gdrive", force_remount=True).
/gdrive

```

[6]: cd '/gdrive/MyDrive/Colab Notebooks/Object Detection with CIFAR'

```

/gdrive/MyDrive/Colab Notebooks/Object Detection with CIFAR

```

[7]: # Creating an object of the class

      cf=cifar('cifar-10-batches-py/') # The location of data directory is given

```

0.2 Understanding the data

```

[8]: # Understanding the way classes have been labelled
      cf.map

```

```

[8]: {0: 'airplane',
      1: 'automobile',
      2: 'bird',
      3: 'cat',
      4: 'deer',
      5: 'dog',
      6: 'frog',
      7: 'horse',
      8: 'ship',
      9: 'truck'}

```

0.2.1 Read the batch data

```
[9]: # Call the class function to read the file and to convert it into images and
      ↪ labels
x_train1,y_train1=cf.read_batch_data(1)
x_train2,y_train2=cf.read_batch_data(2)
x_train3,y_train3=cf.read_batch_data(3)
x_train4,y_train4=cf.read_batch_data(4)
x_train5,y_train5=cf.read_batch_data(5)
```

0.3 Print images and verify from each batch

```
[10]: import matplotlib.pyplot as plt
plt.figure(figsize=(20,5))

#Printing image from batch 1
plt.subplot(1,5,1)
plt.title(cf.label_decode(y_train1[0]))
plt.imshow(x_train1[0])

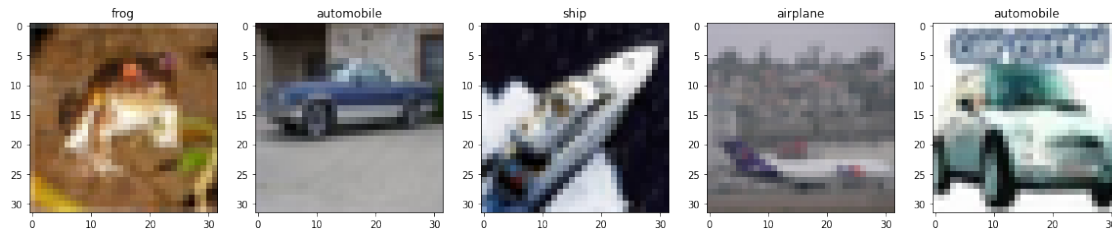
# Printing image from batch 2
plt.subplot(1,5,2)
plt.title(cf.label_decode(y_train2[0]))
plt.imshow(x_train2[0])

#Printing image from batch 3
plt.subplot(1,5,3)
plt.title(cf.label_decode(y_train3[0]))
plt.imshow(x_train3[0])

#Printing image from batch 4
plt.subplot(1,5,4)
plt.title(cf.label_decode(y_train4[0]))
plt.imshow(x_train4[0])

#Printing image from batch 5
plt.subplot(1,5,5)
plt.title(cf.label_decode(y_train5[0]))
plt.imshow(x_train5[0])
```

```
[10]: <matplotlib.image.AxesImage at 0x7fa07b9b6510>
```



0.4 Reading the test batch of images

```
[11]: x_test,y_test=cf.read_test_batch()
```

0.5 Creating a training, Validation and testing data

```
[12]: import numpy as np
x_train=np.vstack((x_train1,x_train2,x_train3,x_train4))
y_train=np.hstack((y_train1,y_train2,y_train3,y_train4))

x_val=x_train5
y_val=y_train5
```

0.6 One Hot Encoding the output

```
[13]: y_test=cf.OHE(y_test)
y_val=cf.OHE(y_val)
y_train=cf.OHE(y_train)
```

0.7 Designing a neural network

```
[14]: from keras import Sequential
from tensorflow.keras.layers import Conv2D,MaxPooling2D,Flatten,Dense,BatchNormalization,Dropout
model=Sequential()

model.add(Conv2D(filters=32,kernel_size=(3,3),padding='same',input_shape=(32,32,3),activation='relu'))
model.add(Conv2D(filters=64,kernel_size=(3,3),padding='same',activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(filters=128,kernel_size=(3,3),padding='same',activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(filters=256,kernel_size=(3,3),padding='same',activation='relu'))
model.add(MaxPooling2D(2,2))
```

```

model.
↳add(Conv2D(filters=512,kernel_size=(3,3),padding='same',activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Flatten())
model.add(Dense(1024,activation='relu'))
model.add(Dense(256,activation='relu'))
model.add(Dense(128,activation='relu'))
model.add(Dense(32,activation='relu'))
model.add(Dense(10,activation='softmax'))

```

[15]: model.summary()

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
conv2d_1 (Conv2D)	(None, 32, 32, 64)	18496
max_pooling2d (MaxPooling2D)	(None, 16, 16, 64)	0
conv2d_2 (Conv2D)	(None, 16, 16, 128)	73856
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 128)	0
conv2d_3 (Conv2D)	(None, 8, 8, 256)	295168
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 256)	0
conv2d_4 (Conv2D)	(None, 4, 4, 512)	1180160
max_pooling2d_3 (MaxPooling2D)	(None, 2, 2, 512)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 1024)	2098176
dense_1 (Dense)	(None, 256)	262400
dense_2 (Dense)	(None, 128)	32896
dense_3 (Dense)	(None, 32)	4128

dense_4 (Dense) (None, 10) 330

```
=====
Total params: 3,966,506
Trainable params: 3,966,506
Non-trainable params: 0
-----
```

```
[16]: model.
      ↪ compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
[23]: model.
      ↪ fit(x_train, y_train, batch_size=200, epochs=10, validation_data=(x_val, y_val), verbose=1)
```

```
Epoch 1/7
200/200 [=====] - 492s 2s/step - loss: 1.8147 -
accuracy: 0.3170 - val_loss: 1.4675 - val_accuracy: 0.4535
Epoch 2/7
200/200 [=====] - 490s 2s/step - loss: 1.2611 -
accuracy: 0.5423 - val_loss: 1.0709 - val_accuracy: 0.6178
Epoch 3/7
200/200 [=====] - 490s 2s/step - loss: 0.9597 -
accuracy: 0.6579 - val_loss: 0.9379 - val_accuracy: 0.6743
Epoch 4/7
200/200 [=====] - 491s 2s/step - loss: 0.7469 -
accuracy: 0.7364 - val_loss: 0.7780 - val_accuracy: 0.7293
Epoch 5/7
200/200 [=====] - 490s 2s/step - loss: 0.5845 -
accuracy: 0.7957 - val_loss: 0.7078 - val_accuracy: 0.7593
Epoch 6/7
200/200 [=====] - 490s 2s/step - loss: 0.4461 -
accuracy: 0.8442 - val_loss: 0.7203 - val_accuracy: 0.7664
Epoch 7/7
200/200 [=====] - 490s 2s/step - loss: 0.3372 -
accuracy: 0.8811 - val_loss: 0.8189 - val_accuracy: 0.7527
```

```
[23]: <keras.callbacks.History at 0x7f1110a9f3d0>
```

```
[24]: import pickle
      pickle.dump(model, open('model.sav', 'wb'))
```

```
INFO:tensorflow:Assets written to:
ram://98cb39c6-fba9-4e34-aa0b-e679f08fd3b3/assets
```

The model has 75 % on data. However, on training for more than 20 epochs, it was seen that the training loss was decreasing, but the validation loss was increasing. Hence, in order to avoid overfitting the number of epochs were reduced to 10. We

can try to reduce overfitting by adding Dropout as it will avoid a few percentage of neurons during the process

0.7.1 Creating another model , same as above but with Dropout layers

```
[17]: from keras import Sequential
from tensorflow.keras.layers import
↳Conv2D,MaxPooling2D,Flatten,Dense,BatchNormalization,Dropout
model=Sequential()

model.
↳add(Conv2D(filters=32,kernel_size=(3,3),padding='same',input_shape=(32,32,3),activation='re
model.add(Conv2D(filters=64,kernel_size=(3,3),padding='same',activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Dropout(0.4))
model.
↳add(Conv2D(filters=128,kernel_size=(3,3),padding='same',activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Dropout(0.3))
model.
↳add(Conv2D(filters=256,kernel_size=(3,3),padding='same',activation='relu'))
model.add(MaxPooling2D(2,2))
model.
↳add(Conv2D(filters=512,kernel_size=(3,3),padding='same',activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(1024,activation='relu'))
model.add(Dense(256,activation='relu'))
model.add(Dense(128,activation='relu'))
model.add(Dense(32,activation='relu'))
model.add(Dense(10,activation='softmax'))

[18]: model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 32, 32, 32)	896
conv2d_6 (Conv2D)	(None, 32, 32, 64)	18496
max_pooling2d_4 (MaxPooling 2D)	(None, 16, 16, 64)	0
dropout (Dropout)	(None, 16, 16, 64)	0

conv2d_7 (Conv2D)	(None, 16, 16, 128)	73856
max_pooling2d_5 (MaxPooling 2D)	(None, 8, 8, 128)	0
dropout_1 (Dropout)	(None, 8, 8, 128)	0
conv2d_8 (Conv2D)	(None, 8, 8, 256)	295168
max_pooling2d_6 (MaxPooling 2D)	(None, 4, 4, 256)	0
conv2d_9 (Conv2D)	(None, 4, 4, 512)	1180160
max_pooling2d_7 (MaxPooling 2D)	(None, 2, 2, 512)	0
dropout_2 (Dropout)	(None, 2, 2, 512)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_5 (Dense)	(None, 1024)	2098176
dense_6 (Dense)	(None, 256)	262400
dense_7 (Dense)	(None, 128)	32896
dense_8 (Dense)	(None, 32)	4128
dense_9 (Dense)	(None, 10)	330

```
=====
Total params: 3,966,506
Trainable params: 3,966,506
Non-trainable params: 0
-----
```

```
[20]: from sklearn.model_selection import KFold

model.
    ↪ compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
kF=KFold(n_splits=5)
count=1
for train_index, test_index in kF.split(x_train):
    print('\n Fold '+str(count)+'\n')
```

```

model.
→fit(x_train[train_index],y_train[train_index],batch_size=50,epochs=7,validation_data=(x_val,
model.evaluate(x_train[test_index],y_train[test_index]))
count+=1

```

Fold 1

Epoch 1/7

640/640 [=====] - 26s 39ms/step - loss: 0.4085 - accuracy: 0.8605 - val_loss: 0.6669 - val_accuracy: 0.7914

Epoch 2/7

640/640 [=====] - 24s 37ms/step - loss: 0.3659 - accuracy: 0.8728 - val_loss: 0.6818 - val_accuracy: 0.7874

Epoch 3/7

640/640 [=====] - 24s 38ms/step - loss: 0.3393 - accuracy: 0.8830 - val_loss: 0.6964 - val_accuracy: 0.7848

Epoch 4/7

640/640 [=====] - 25s 39ms/step - loss: 0.3361 - accuracy: 0.8833 - val_loss: 0.7004 - val_accuracy: 0.7833

Epoch 5/7

640/640 [=====] - 24s 38ms/step - loss: 0.3133 - accuracy: 0.8928 - val_loss: 0.7108 - val_accuracy: 0.7845

Epoch 6/7

640/640 [=====] - 24s 38ms/step - loss: 0.3047 - accuracy: 0.8957 - val_loss: 0.7110 - val_accuracy: 0.7904

Epoch 7/7

640/640 [=====] - 25s 39ms/step - loss: 0.2779 - accuracy: 0.9051 - val_loss: 0.7575 - val_accuracy: 0.7807

250/250 [=====] - 2s 9ms/step - loss: 0.2805 - accuracy: 0.9014

Fold 2

Epoch 1/7

640/640 [=====] - 24s 38ms/step - loss: 0.3293 - accuracy: 0.8870 - val_loss: 0.7191 - val_accuracy: 0.7872

Epoch 2/7

640/640 [=====] - 24s 38ms/step - loss: 0.2965 - accuracy: 0.8991 - val_loss: 0.7276 - val_accuracy: 0.7907

Epoch 3/7

640/640 [=====] - 24s 38ms/step - loss: 0.2879 - accuracy: 0.9028 - val_loss: 0.7812 - val_accuracy: 0.7813

Epoch 4/7

640/640 [=====] - 25s 39ms/step - loss: 0.2680 - accuracy: 0.9097 - val_loss: 0.7160 - val_accuracy: 0.7852

Epoch 5/7

640/640 [=====] - 24s 38ms/step - loss: 0.2535 -
accuracy: 0.9137 - val_loss: 0.7302 - val_accuracy: 0.7855
Epoch 6/7
640/640 [=====] - 25s 39ms/step - loss: 0.2509 -
accuracy: 0.9156 - val_loss: 0.7333 - val_accuracy: 0.7926
Epoch 7/7
640/640 [=====] - 24s 38ms/step - loss: 0.2402 -
accuracy: 0.9189 - val_loss: 0.7690 - val_accuracy: 0.7838
250/250 [=====] - 2s 9ms/step - loss: 0.2239 -
accuracy: 0.9234

Fold 3

Epoch 1/7
640/640 [=====] - 24s 38ms/step - loss: 0.2789 -
accuracy: 0.9071 - val_loss: 0.7244 - val_accuracy: 0.7949
Epoch 2/7
640/640 [=====] - 25s 39ms/step - loss: 0.2512 -
accuracy: 0.9168 - val_loss: 0.7463 - val_accuracy: 0.7985
Epoch 3/7
640/640 [=====] - 25s 39ms/step - loss: 0.2426 -
accuracy: 0.9183 - val_loss: 0.7322 - val_accuracy: 0.7885
Epoch 4/7
640/640 [=====] - 25s 39ms/step - loss: 0.2380 -
accuracy: 0.9214 - val_loss: 0.7527 - val_accuracy: 0.7925
Epoch 5/7
640/640 [=====] - 25s 39ms/step - loss: 0.2146 -
accuracy: 0.9274 - val_loss: 0.7936 - val_accuracy: 0.7935
Epoch 6/7
640/640 [=====] - 25s 39ms/step - loss: 0.2107 -
accuracy: 0.9298 - val_loss: 0.7984 - val_accuracy: 0.7886
Epoch 7/7
640/640 [=====] - 25s 39ms/step - loss: 0.2089 -
accuracy: 0.9308 - val_loss: 0.8102 - val_accuracy: 0.7859
250/250 [=====] - 2s 9ms/step - loss: 0.1436 -
accuracy: 0.9520

Fold 4

Epoch 1/7
640/640 [=====] - 25s 38ms/step - loss: 0.2406 -
accuracy: 0.9211 - val_loss: 0.7356 - val_accuracy: 0.7886
Epoch 2/7
640/640 [=====] - 25s 39ms/step - loss: 0.2208 -
accuracy: 0.9272 - val_loss: 0.7510 - val_accuracy: 0.7915
Epoch 3/7
640/640 [=====] - 25s 39ms/step - loss: 0.2083 -
accuracy: 0.9323 - val_loss: 0.8197 - val_accuracy: 0.7860

Epoch 4/7
 640/640 [=====] - 24s 38ms/step - loss: 0.2008 - accuracy: 0.9351 - val_loss: 0.8143 - val_accuracy: 0.7904
 Epoch 5/7
 640/640 [=====] - 24s 38ms/step - loss: 0.1919 - accuracy: 0.9370 - val_loss: 0.8469 - val_accuracy: 0.7822
 Epoch 6/7
 640/640 [=====] - 25s 39ms/step - loss: 0.1945 - accuracy: 0.9362 - val_loss: 0.8150 - val_accuracy: 0.7914
 Epoch 7/7
 640/640 [=====] - 25s 39ms/step - loss: 0.1831 - accuracy: 0.9391 - val_loss: 0.8217 - val_accuracy: 0.7908
 250/250 [=====] - 2s 9ms/step - loss: 0.1012 - accuracy: 0.9685

Fold 5

Epoch 1/7
 640/640 [=====] - 25s 39ms/step - loss: 0.2108 - accuracy: 0.9299 - val_loss: 0.7791 - val_accuracy: 0.7918
 Epoch 2/7
 640/640 [=====] - 25s 39ms/step - loss: 0.1842 - accuracy: 0.9389 - val_loss: 0.8220 - val_accuracy: 0.7885
 Epoch 3/7
 640/640 [=====] - 24s 38ms/step - loss: 0.1802 - accuracy: 0.9420 - val_loss: 0.7867 - val_accuracy: 0.7967
 Epoch 4/7
 640/640 [=====] - 25s 39ms/step - loss: 0.1663 - accuracy: 0.9458 - val_loss: 0.8159 - val_accuracy: 0.7880
 Epoch 5/7
 640/640 [=====] - 24s 38ms/step - loss: 0.1615 - accuracy: 0.9487 - val_loss: 0.7782 - val_accuracy: 0.7906
 Epoch 6/7
 640/640 [=====] - 25s 39ms/step - loss: 0.1624 - accuracy: 0.9480 - val_loss: 0.8624 - val_accuracy: 0.7930
 Epoch 7/7
 640/640 [=====] - 24s 38ms/step - loss: 0.1667 - accuracy: 0.9463 - val_loss: 0.8019 - val_accuracy: 0.7966
 250/250 [=====] - 2s 9ms/step - loss: 0.0853 - accuracy: 0.9761

```
[26]: model.  

      ↪ compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])  

      model.  

      ↪ fit(x_train, y_train, batch_size=50, epochs=20, validation_data=(x_val, y_val), verbose=1)
```

Epoch 1/20
 800/800 [=====] - 311s 388ms/step - loss: 1.6707 -

accuracy: 0.3834 - val_loss: 1.3837 - val_accuracy: 0.4981
 Epoch 2/20
 800/800 [=====] - 307s 384ms/step - loss: 1.2032 -
 accuracy: 0.5706 - val_loss: 0.9606 - val_accuracy: 0.6538
 Epoch 3/20
 800/800 [=====] - 307s 384ms/step - loss: 0.9874 -
 accuracy: 0.6521 - val_loss: 0.9107 - val_accuracy: 0.6762
 Epoch 4/20
 800/800 [=====] - 307s 384ms/step - loss: 0.8684 -
 accuracy: 0.6958 - val_loss: 0.7493 - val_accuracy: 0.7364
 Epoch 5/20
 800/800 [=====] - 307s 384ms/step - loss: 0.7812 -
 accuracy: 0.7312 - val_loss: 0.7172 - val_accuracy: 0.7527
 Epoch 6/20
 800/800 [=====] - 308s 385ms/step - loss: 0.7169 -
 accuracy: 0.7503 - val_loss: 0.6876 - val_accuracy: 0.7644
 Epoch 7/20
 800/800 [=====] - 307s 384ms/step - loss: 0.6649 -
 accuracy: 0.7660 - val_loss: 0.6687 - val_accuracy: 0.7718
 Epoch 8/20
 800/800 [=====] - 306s 383ms/step - loss: 0.6218 -
 accuracy: 0.7835 - val_loss: 0.6621 - val_accuracy: 0.7752
 Epoch 9/20
 800/800 [=====] - 307s 384ms/step - loss: 0.5826 -
 accuracy: 0.7958 - val_loss: 0.6244 - val_accuracy: 0.7854
 Epoch 10/20
 800/800 [=====] - 308s 385ms/step - loss: 0.5526 -
 accuracy: 0.8047 - val_loss: 0.6435 - val_accuracy: 0.7867
 Epoch 11/20
 800/800 [=====] - 306s 383ms/step - loss: 0.5223 -
 accuracy: 0.8147 - val_loss: 0.6036 - val_accuracy: 0.8005
 Epoch 12/20
 800/800 [=====] - 306s 383ms/step - loss: 0.5016 -
 accuracy: 0.8253 - val_loss: 0.6389 - val_accuracy: 0.7905
 Epoch 13/20
 800/800 [=====] - 306s 383ms/step - loss: 0.4789 -
 accuracy: 0.8321 - val_loss: 0.6128 - val_accuracy: 0.8018
 Epoch 14/20
 800/800 [=====] - 305s 382ms/step - loss: 0.4608 -
 accuracy: 0.8380 - val_loss: 0.6164 - val_accuracy: 0.8045
 Epoch 15/20
 800/800 [=====] - 305s 381ms/step - loss: 0.4466 -
 accuracy: 0.8415 - val_loss: 0.6331 - val_accuracy: 0.8030
 Epoch 16/20
 800/800 [=====] - 305s 381ms/step - loss: 0.4364 -
 accuracy: 0.8439 - val_loss: 0.6110 - val_accuracy: 0.8066
 Epoch 17/20
 800/800 [=====] - 307s 383ms/step - loss: 0.4148 -

```
accuracy: 0.8532 - val_loss: 0.6074 - val_accuracy: 0.8120
Epoch 18/20
800/800 [=====] - 307s 383ms/step - loss: 0.4070 -
accuracy: 0.8563 - val_loss: 0.5929 - val_accuracy: 0.8144
Epoch 19/20
800/800 [=====] - 305s 382ms/step - loss: 0.3916 -
accuracy: 0.8626 - val_loss: 0.6401 - val_accuracy: 0.7984
Epoch 20/20
800/800 [=====] - 305s 382ms/step - loss: 0.3788 -
accuracy: 0.8664 - val_loss: 0.6184 - val_accuracy: 0.8113
```

```
[26]: <keras.callbacks.History at 0x7f19ebdeb5d0>
```

```
[27]: model.evaluate(x_test,y_test)
```

```
313/313 [=====] - 19s 61ms/step - loss: 0.6527 -
accuracy: 0.7988
```

```
[27]: [0.6526737213134766, 0.798799991607666]
```

```
[35]: # Saving the model
```

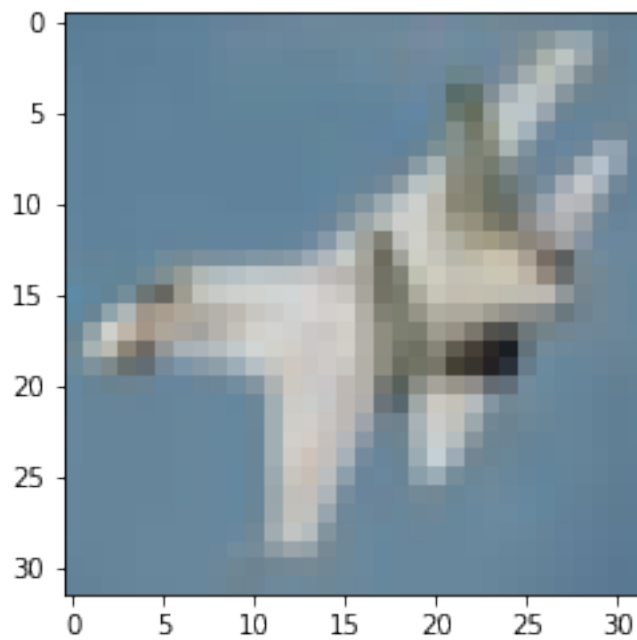
```
import pickle
pickle.dump(model,open('cnnModel81.sav','wb'))
```

```
INFO:tensorflow:Assets written to:
ram://9f446629-a858-46c9-946d-b8857f1310db/assets
```

0.8 Testing the working of the model

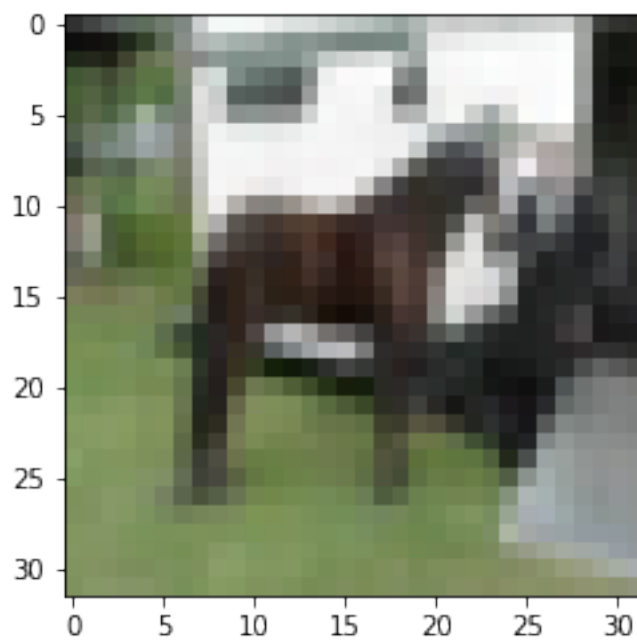
```
[108]: cf.test_prediction(x_test[10])
```

```
airplane
```



```
[109]: cf.test_prediction(x_test[20])
```

horse



0.9 We can see that the model is predicting properly

[]: