Classifying Images with and without Texts using Pre-Trained Models

- Name: Akshara Shukla
- Class: AI7
- Pre-Trained CNN Exercise

Introduction

This project consists of experimenting with a pre-trained model VGG16 for classing images with and without texts. This will be an interesting assignment as the classification of images with images can be a helpful tool for the industries that work closely with papers. For instance in finance, for classifying the bills paper into the textual data with the numbers and important details and leaving the not required part of the paper. This notebook consists of the exploratory data analysis along with the model fine-tuning, training, evaluation and testing it on new images.

Firstly, lets import the required libraries.

Setup

In [46]:

```
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns

from IPython.display import Image
import splitfolders

import os #Operating Systems for fetching image data from folders/sub-folders on my pc
import glob #global for getting filenames matching a specific pattern
import collections as cl #For storing filenames in a container that will hold the count of each of the elements present i

import keras                          #For working our neural network which provides us various functions for building ne
from keras import models, layers
from keras.preprocessing import image
```

```
import sklearn.metrics
from sklearn.metrics import classification_report, confusion_matrix
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img, img_to_array,array_to_img #for loading all the images from a particular d
from keras.preprocessing.image import ImageDataGenerator #for rescaling and importing our image files from our pc's direc
from keras.optimizers import Adam,RMSprop  #the optimizer for minimising loss


#gradcam
from keras import backend as K
import tensorflow as tf
tf.compat.v1.disable_eager_execution()
import cv2

#pre-trained models to be compared
from tensorflow.keras.applications.vgg16 import VGG16 #pretrained model for image classification

import cv2 as cv
#Importing the different callback API (functions) we will be using while training our model.
from keras.callbacks import EarlyStopping
from keras.callbacks import CSVLogger
from keras.callbacks import ReduceLROnPlateau
from keras.callbacks import ModelCheckpoint

%matplotlib inline

print('numpy version:', np.__version__)
print('pandas version:', pd.__version__)
print('matplotlib version:', matplotlib.__version__)
print('seaborn version', sns.__version__)
```

```
numpy version: 1.18.5
pandas version: 1.1.4
matplotlib version: 3.3.4
seaborn version 0.11.1
```

Data Preparation

The dataset that is used is taken from kaggle. You can access the database from this link. The dataset consits of textual and non-textual images from a 12th grade Biology textbook. I have locally downloaded and saved the images on my local laptop. By using the 'splitfolder' library, I performed the train and test split of the images with the ratio of 80:20.

The training images have been loaded in the train_dir variable and the images for testing in the test_dir variable.

In [2]:
```
splitfolders.ratio(r'C:\Users\PC\Evernote\Logs\text_nontext/', output = 'output', seed = 42,ratio=(.8,.2))
```

```
Copying files: 916 files [00:01, 479.84 files/s]
```

In [3]:
```
#Defining paths to our train and test folders
train_dir = 'output/train/'
test_dir = 'output/test/'
```

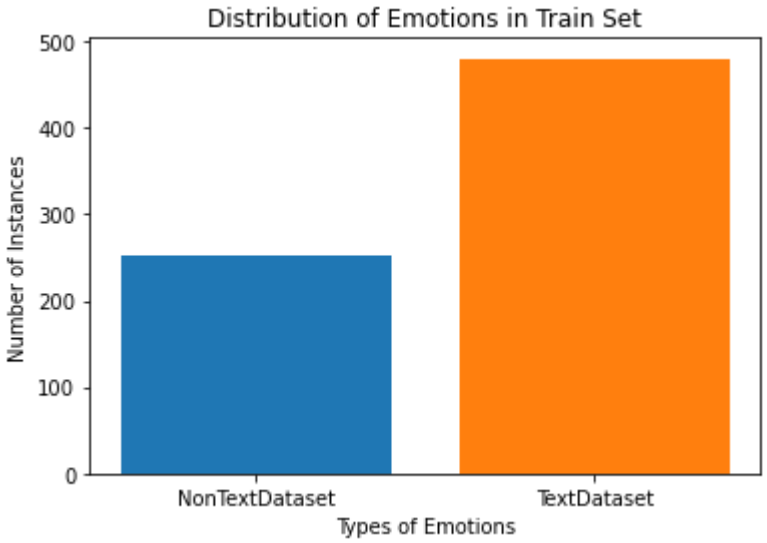
In [4]:
```
#Finding folders in our training directory containing images
print(os.listdir(train_dir))
print(range(len(os.listdir(train_dir))))
```

```
['NonTextDataset', 'TextDataset']
range(0, 2)
```

The above results depict that our train_dir folder includes two folders containing images for the Nontextdata and Textdata.

In [5]:
```
#Starting total_img from 0 because the images will be adding up after every run
total_img = 0
#Going in each training emotion folder and calculating the number of files by using len().
for emotions in os.listdir(train_dir):
    img_count = len(os.listdir(train_dir + emotions))
    total_img += img_count
    print(f'{emotions} has {img_count} images available')
    plt.bar(emotions,img_count)

plt.xlabel('Types of Emotions')
plt.ylabel('Number of Instances')
plt.title('Distribution of Emotions in Train Set')
print("-------------------------------------")
print(f'The total images in our train directory are: {total_img}')
```

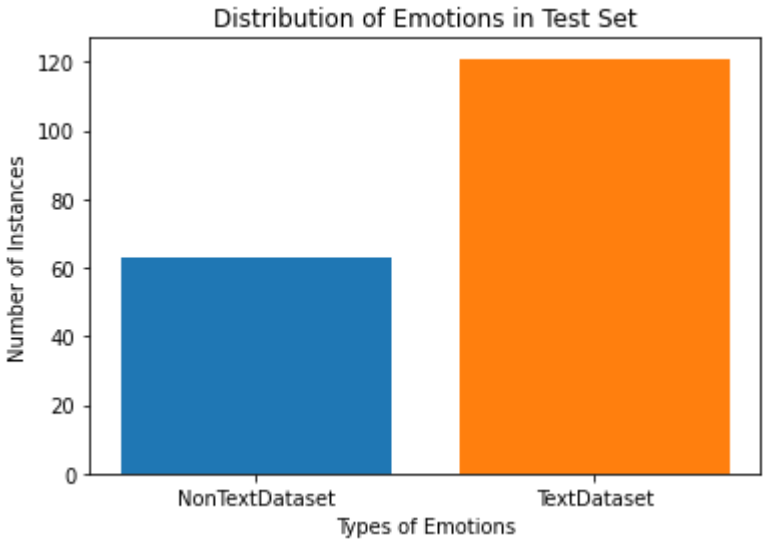
```
NonTextDataset has 252 images available
TextDataset has 480 images available
-------------------------------------
The total images in our train directory are: 732
```



In [6]:

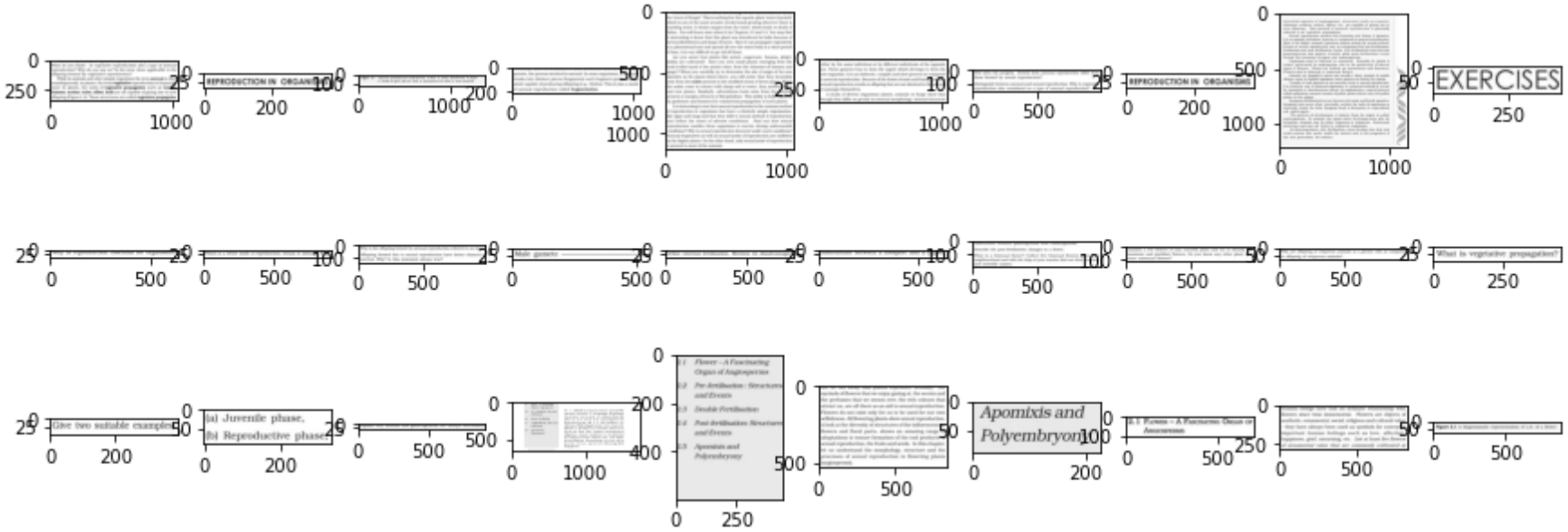
```
#Starting total_img from 0 because the images will be adding up after every run
total_img = 0
#Going in each training emotion folder and calculating the number of files by using len().
for emotions in os.listdir(test_dir):
    img_count = len(os.listdir(test_dir + emotions))
    total_img += img_count
    print(f'{emotions} has {img_count} images available')
    plt.bar(emotions,img_count)

plt.xlabel('Types of Emotions')
plt.ylabel('Number of Instances')
plt.title('Distribution of Emotions in Test Set')
print("-------------------------------------")
print(f'The total images in our test directory are: {total_img}')
```

```
NonTextDataset has 63 images available
TextDataset has 121 images available
-------------------------------------
The total images in our test directory are: 184
```



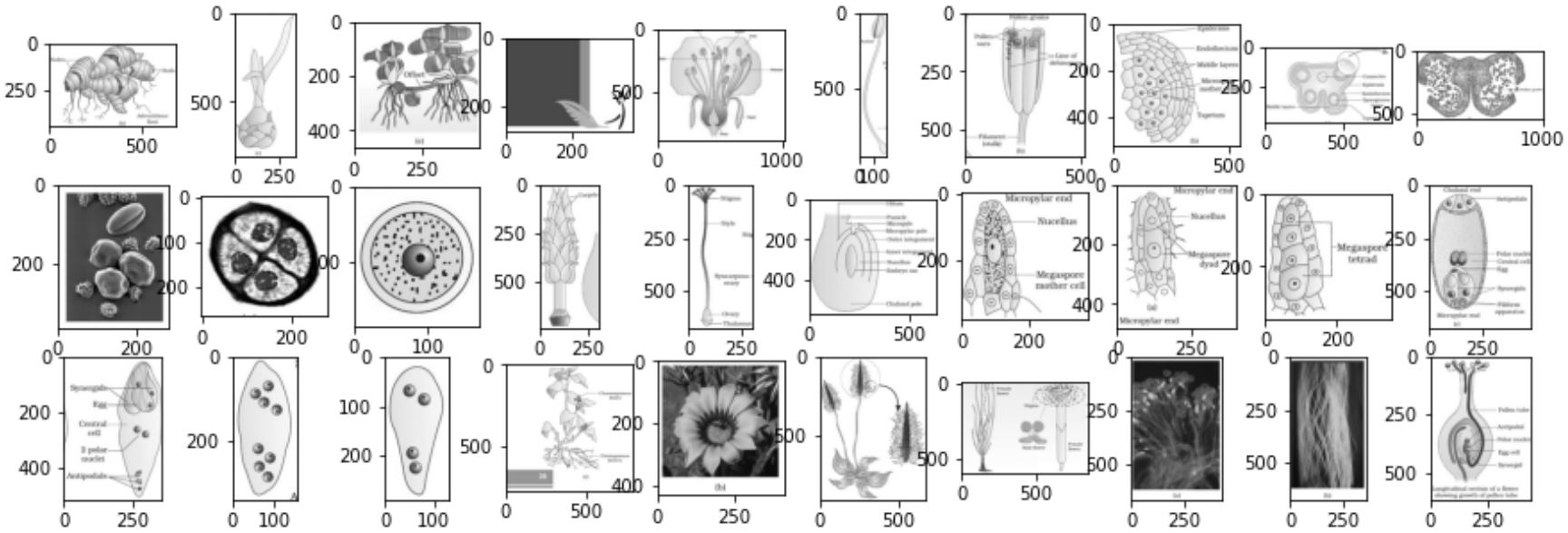
The results above illustrate the distribution of the images in both of our train and test sets. We can see that there are more data points for the images with the text. In total there are a total of 732 images for training and 184 for testing. The difference between the two categories isn't much but it would be interesting into how the classification model will understand and learn from them.

In [7]:
```
#Plotting the first 30 images from our Textdataset directory
fig=plt.figure(figsize=(15, 5))
for i in range(1,31):
    img = load_img(train_dir + 'TextDataset/' + os.listdir(train_dir + 'TextDataset/')[i])
    fig.add_subplot(3,10,i)
    plt.imshow(img, cmap = "gray")
```



The above results illustrate the images with the textual data on them. All of the images only include some description and examples of textual images.

In [8]:
```
#Plotting the first 30 images from our NonText directory
fig=plt.figure(figsize=(15, 5))
for i in range(1,31):
    img = load_img(train_dir + 'NonTextDataset/' + os.listdir(train_dir + 'NonTextDataset/')[i])
    fig.add_subplot(3,10,i)
    plt.imshow(img, cmap = "gray")
```



The above results illustrate all of the images with the non-textual data values. Most of the images show some biological diagram such as the cell or flower structure. What's interesting is that, due to the presence of diagrams there's some text also present in them.

Image Augementation and Preparation

Now that we have an understanding of the data values, we can start with the data pre-processing. This step will include looking at the shape of the images, resizing them if they are unlike and normalizing their pixel values. These metrics are important because for passing images, they need to be in the same size for the model to not differentiate or biased towards an image which is for instance bigger in size.

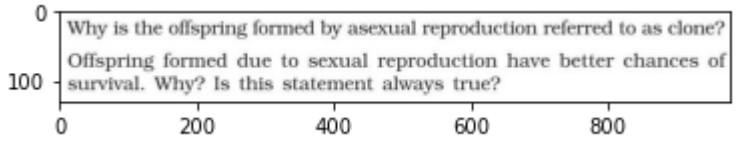
In [9]:

```
# Read image
img = load_img(train_dir + 'TextDataset/' + os.listdir(train_dir + 'TextDataset/')[13])

# Changing image to an array and checking the dimensions of image
img = np.array(img)
dimensions = img.shape

# height, width, number of channels in image
height = img.shape[0]
width = img.shape[1]
channels = img.shape[2]

print('Image Dimension    : ',dimensions)
```

```
print('Image Height       : ',height)
print('Image Width        : ',width)
print('Number of Channels : ',channels)
plt.imshow(img)
```

```
Image Dimension    :  (130, 978, 3)
Image Height       :  130
Image Width        :  978
Number of Channels :  3
```

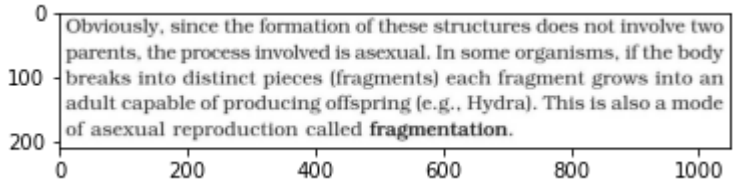
Out[9]:
```
<matplotlib.image.AxesImage at 0x19ce7ebbb20>
```



In [10]:
```
# Read image
img = load_img(train_dir + 'TextDataset/' + os.listdir(train_dir + 'TextDataset/')[4])

# Changing image to an array and checking the dimensions of image
img = np.array(img)
dimensions = img.shape

# height, width, number of channels in image
height = img.shape[0]
width = img.shape[1]
channels = img.shape[2]

print('Image Dimension    : ',dimensions)
print('Image Height       : ',height)
print('Image Width        : ',width)
print('Number of Channels : ',channels)
plt.imshow(img)
```

```
Image Dimension    :  (210, 1050, 3)
Image Height       :  210
Image Width        :  1050
Number of Channels :  3
```

Out[10]:
```
<matplotlib.image.AxesImage at 0x19d02598670>
```



It can be infered that the images aren't of the same image dimension since by printing the shape of two different images from the same directory, they both have different shape.

Performing some scaling and image augmentation to our images by zooming on them, shearing them, and flipping etc, the **ImageDataGenerator** helps us augment and generate more images from the images already exisiting in our df. So, initially rescaling the pixel values and rotating our images in a range 40 with zooming and shearing on 20% and flipping the images horizontally for the model to learn from different angles. This helped the model to better understand the images and find key similarities and patterns.

In [11]:
```
#Performing important image augmenting techniques such as twisting and turning our images and generating new outputs on o
train_path = ImageDataGenerator(rescale = 1./255.,
                                rotation_range = 40,
                                shear_range = 0.2,
                                zoom_range = 0.2,
                                horizontal_flip = True)

#Augmenting and generating new images in our test images.
test_path = ImageDataGenerator(rescale = 1/255.0)

#Creating training set of our model consisting pixel values and emotion indices.
#Using batch size as 32 for train and test to begin experimenting
train_set = train_path.flow_from_directory(train_dir, target_size = (48,48), batch_size = 32, class_mode = 'categorical')

##Creating test set of our model which will be used to test our model's accuracy.
test_set = test_path.flow_from_directory(test_dir, target_size = (48,48), batch_size = 32, class_mode = 'categorical')
```

```
Found 732 images belonging to 2 classes.
Found 184 images belonging to 2 classes.
```

In [12]:
```
train_set.class_indices
```

Out[12]:
```
{'NonTextDataset': 0, 'TextDataset': 1}
```

In [13]:
```
test_set.class_indices
```

Out[13]:
```
{'NonTextDataset': 0, 'TextDataset': 1}
```

Adding CallBack functions for Model Evaluation

A **callback function** is a function that helps in performing various tasks while the training is happening.

a) We will be using **EarlyStop** which is great for solving overfitting problems as it stops the training process if our monitored value *val_loss* has stopped improving after certain epochs which in our case is defined as 4 *patience*.

c) **ReduceLRPlateau** which reduces the learning rate once a metric has stopped improving or rather when it has fully learnt.

In [14]:
```
#EarlyStop Callback
earlystop = EarlyStopping(monitor = 'val_loss',
                          min_delta = 0,    #training process will be stopped if the absolute change of val_loss is less t
                          patience = 4,
                          verbose = 1,
                          restore_best_weights=True)

#ReduceLRPlateau Callback
reduce_lr = ReduceLROnPlateau(monitor = 'val_loss',
                              factor = 0.1,   #value by which learning rate will be reduced (new_lr = lr*factor)
                              patience = 3,   #epochs after which the learning rate would reduce is same learning rate is
                              min_lr = 0.001) #lower bound on the learning rate
```

Determining the **batch_size** our our training and testing sets. The batch_size determines the number of samples of images which will be trained in each mini batch. Our batch size is 32 which means out of the total samples of our training 732, there are 32 small subset sizes which are used for training the model while learning. Each batch trains network in a successive order, taking into account the updated weights coming from the appliance of the previous batch.

Additionally, calculating the **steps_per_epoch** which is the number of batch iterations which will be trained before a training epoch is considered finished. Our steps per epoch is 22 for training set which implies there will be 22 images passed through layers and trained in our model in each epoch.

In [15]:
```
steps_per_epoch = train_set.n // train_set.batch_size #double slash division returns rounded int
print(f'The train set steps per epoch: {steps_per_epoch}')
print(f'The train set has: {train_set.n}')
print(f'Train Set batch has: {train_set.batch_size} elements')
print("----------------------------------------")
validation_steps = test_set.n //test_set.batch_size
print(f'The test set steps per epoch: {validation_steps}')
```

```
print(f'The test set has: {test_set.n}')
print(f'Test Set batch has:  {test_set.batch_size} elements')
```

```
The train set steps per epoch: 22
The train set has: 732
Train Set batch has: 32 elements
---------------------------------------
The test set steps per epoch: 5
The test set has: 184
Test Set batch has:  32 elements
```

Implementing VGG16 on our dataset

After conducting research on the usage of CNN's in image recognition, I came across some pretrained models one of which is Vgg16 which is known for producing great results on datasets. It has outperformed the other models such as ResNet in the largest object detection dataset i.e., imagenet. The VGG architecture consists of blocks, where each block is composed of 2D Convolution and Max Pooling layers. Therefore, to test if building our product on vgg16 would give us a more accurate result, we assigned our input shape with our default image scaled as follows. One parameter to understand here, is the weights is set to 'imagenet' which is the original project which was used to build this architecture, therefore the same weights are added by default.

I'll be inputting the shape of the images as 48x48 because smaller the images because I want to test if the smaller pixel values perform better or not.

In [16]:
```
#Instantiating a VGG16 model and assigning it to variable base_model.
base_model = VGG16(input_shape = (48, 48, 3), # Shape of our images
                   include_top = False, # Only having the first half of the model because we will be defining the last fu
                   weights = 'imagenet') #Setting weights as they were assigned in the original architecture
```

In [17]:
```
#Taking all the layers in 'base model', and assigning them as False for trainable because we want to train our model with
for layer in base_model.layers:
    layer.trainable = False
```

In [18]:
```
#Following the default values of implementing vgg16
# Flattening the output layer to 1 dimension for them to be taken into hidden layers
x = layers.Flatten()(base_model.output)

# Adding a fully connected layer with 126 hidden units and ReLU activation
x = layers.Dense(126, activation='relu')(x)
```

```
# Add a dropout rate of 0.7
x = layers.Dropout(0.7)(x)

# Assigning 2 as our number of output classes with activation softmax for the last layer
x = layers.Dense(2, activation='softmax')(x)

# Lastly, assining the fully made model to modelvgg containing the layers and parameters according to our dataset
modelvgg = keras.models.Model(base_model.input, x)

# Compiling the vgg16 model with adam optimizer and Experimenting the default loss function.
modelvgg.compile(optimizer = 'adam', loss = 'binary_crossentropy',metrics = ['accuracy'])
```

In [19]:

```
vgghist = modelvgg.fit(train_set,
                       validation_data = test_set,
                       steps_per_epoch = steps_per_epoch,
                       epochs = 11,
                       callbacks = [earlystop, reduce_lr])
```

```
Epoch 1/11
22/22 [==============================] - ETA: 0s - batch: 10.5000 - size: 31.8182 - loss: 0.5974 - accuracy: 0.7257
C:\Users\PC\anaconda3\lib\site-packages\tensorflow\python\keras\engine\training.py:2325: UserWarning: `Model.state_update
s` will be removed in a future version. This property should not be used in TensorFlow 2.0, as `updates` are applied auto
matically.
  warnings.warn('`Model.state_updates` will be removed in a future version. '
22/22 [==============================] - 6s 275ms/step - batch: 10.5000 - size: 31.8182 - loss: 0.5974 - accuracy: 0.7257
- val_loss: 0.3692 - val_accuracy: 0.9185
Epoch 2/11
22/22 [==============================] - 6s 278ms/step - batch: 10.5000 - size: 31.8182 - loss: 0.3460 - accuracy: 0.8971
- val_loss: 0.2575 - val_accuracy: 0.9511
Epoch 3/11
22/22 [==============================] - 6s 276ms/step - batch: 10.5000 - size: 32.0000 - loss: 0.2647 - accuracy: 0.9119
- val_loss: 0.1890 - val_accuracy: 0.9565
Epoch 4/11
22/22 [==============================] - 6s 281ms/step - batch: 10.5000 - size: 31.6364 - loss: 0.2366 - accuracy: 0.9296
- val_loss: 0.1595 - val_accuracy: 0.9565
Epoch 5/11
22/22 [==============================] - 6s 268ms/step - batch: 10.5000 - size: 31.8182 - loss: 0.2194 - accuracy: 0.9300
- val_loss: 0.1372 - val_accuracy: 0.9565
Epoch 6/11
22/22 [==============================] - 6s 270ms/step - batch: 10.5000 - size: 32.0000 - loss: 0.1740 - accuracy: 0.9517
- val_loss: 0.1238 - val_accuracy: 0.9565
Epoch 7/11
```

```
22/22 [==============================] - 6s 264ms/step - batch: 10.5000 - size: 31.8182 - loss: 0.1823 - accuracy: 0.9414
- val_loss: 0.1255 - val_accuracy: 0.9565
Epoch 8/11
22/22 [==============================] - 6s 261ms/step - batch: 10.5000 - size: 31.8182 - loss: 0.1557 - accuracy: 0.9514
- val_loss: 0.1055 - val_accuracy: 0.9565
Epoch 9/11
22/22 [==============================] - 6s 256ms/step - batch: 10.5000 - size: 31.6364 - loss: 0.1379 - accuracy: 0.9583
- val_loss: 0.1061 - val_accuracy: 0.9565
Epoch 10/11
22/22 [==============================] - 6s 259ms/step - batch: 10.5000 - size: 31.8182 - loss: 0.1543 - accuracy: 0.9529
- val_loss: 0.0961 - val_accuracy: 0.9565
Epoch 11/11
22/22 [==============================] - 6s 258ms/step - batch: 10.5000 - size: 32.0000 - loss: 0.1300 - accuracy: 0.9645
- val_loss: 0.0842 - val_accuracy: 0.9674
```

VGG16 Evaluation

For the model's evaluation, I'll be plotting the loss and accuracy plots, confusion matrix, and classification report.

In [20]:

```
train_loss, train_acc = modelvgg.evaluate(train_set)
test_loss, test_acc = modelvgg.evaluate(test_set)
print("Training set Accuracy: {:.2f}" .format(train_acc))
print("Test set Accuracy: {:.2f}" .format(test_acc))
```

```
Training set Accuracy: 0.95
Test set Accuracy: 0.97
```

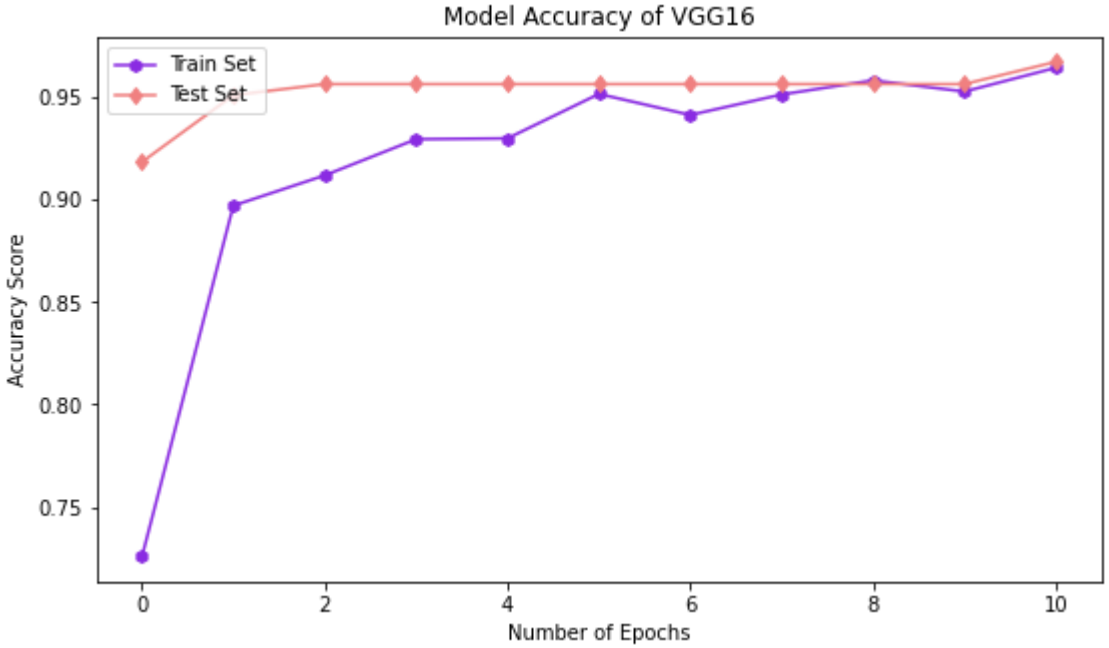
The model.evaluate() provides the loss and the metrics (accuracy) of the model over the epochs that it has been trained as explained on the tensorflow documentation. Since, our training and test accuracy are approximately similar but by training again the difference is of 2%, this can indicate that the model has understood the images as a starting point in a good manner. According to this link which describes how to interpret when the value is same for both the sets.

The high number of the accuracy is due to the process of transfer learning. The pre-trained models are already trained on large datasets and have been fine-tuned as well.

In [21]:

```
#Model Accuracy
plt.figure(figsize=(9,5))
plt.plot(vgghist.history['accuracy'],color = 'blueviolet',marker = "h", label = "Train Set")
plt.plot(vgghist.history['val_accuracy'],color = 'lightcoral', marker = "d", label = "Test Set")

plt.title('Model Accuracy of VGG16')
```

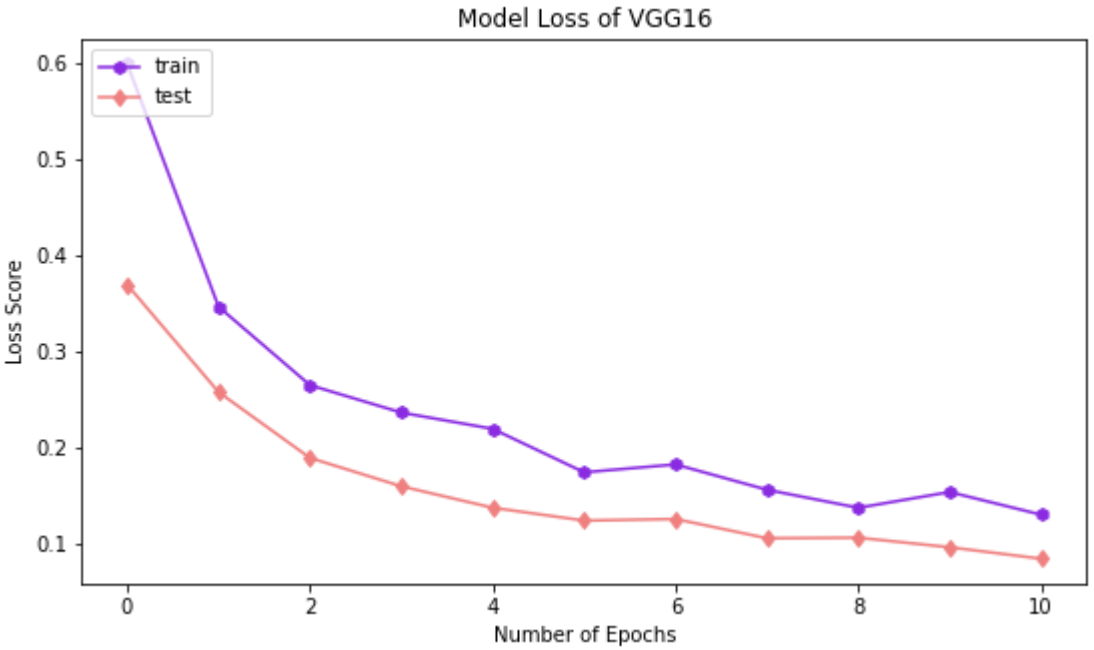
```
plt.xlabel('Number of Epochs')
plt.ylabel('Accuracy Score')
plt.legend(loc='upper left')
plt.show()
```



The above plot illustrates the accuracy of the model over the epochs. The understanding that can be collected is that throughout our epochs, both the sets have learned with a similar accuracy rate with a bit of difference. This curve is quite close to the ideal curve but there can be more work done to improve the train set's accuracy.

In [22]:

```
#Model Loss
plt.figure(figsize=(9,5))
plt.plot(vgghist.history['loss'],color = 'blueviolet', marker = "h")
plt.plot(vgghist.history['val_loss'],color = 'lightcoral', marker = "d")

plt.title('Model Loss of VGG16')
plt.xlabel('Number of Epochs')
plt.ylabel('Loss Score')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



The loss graph above illustrates the loss value throughout the epochs for train and test set. Loss values are the average of the total amount of error that can be expected from the model. The curve is declining for both the sets that means the model has leart well.

In [23]:
```
y_predvgg = modelvgg.predict(test_set)
y_predvgg = np.argmax(y_predvgg, axis=1) #Assigning, axis = 1, to get the highest common values and for classification me
y_predvgg
```

Out[23]:
```
array([1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1,
       1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0,
       1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1,
       0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
       0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0,
       1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1,
       1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1,
       0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 1], dtype=int64)
```

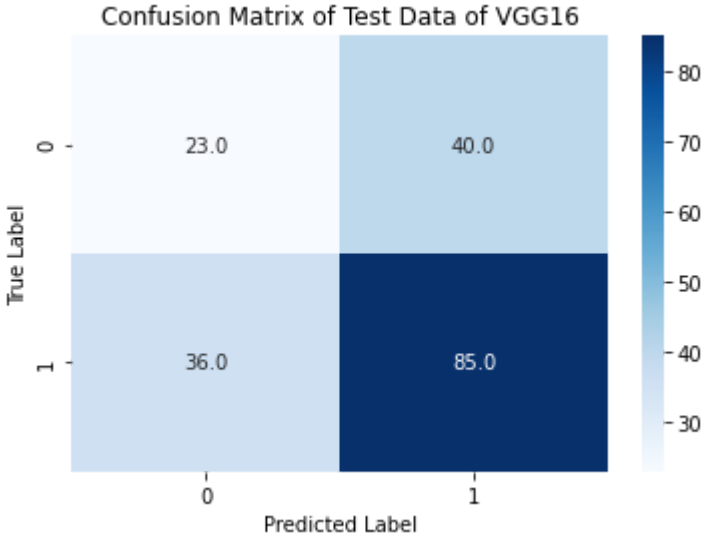
In [24]:
```
y_true = test_set
```

In [25]:
```
true_classes = test_set.classes
class_labels = list(test_set.class_indices.keys())
```

```
class_labels
```

Out[25]:
```
['NonTextDataset', 'TextDataset']
```

In [26]:
```
test_cm = sklearn.metrics.confusion_matrix(true_classes, y_predvgg)
sns.heatmap(test_cm, annot = True, cmap = plt.cm.Blues,fmt = ".1f" ) #Using fmt as .1f to turn the hexadecimal values to

plt.title("Confusion Matrix of Test Data of VGG16")
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```



The confusion matrix plotted above showes that the category of 1 i.e., (TextData) has been classified pretty well without 85.0 being correctly classified and half of it being miss-classified. While for the category 0 i.e.,(NonTextualData) has classified majority of the images as the opposite label.

In [27]:
```
clf_test = classification_report(test_set.classes, y_predvgg, target_names = class_labels)
print(clf_test)
```

```
                precision    recall  f1-score   support

NonTextDataset       0.39      0.37      0.38        63
   TextDataset       0.68      0.70      0.69       121
```

```
              precision    recall  f1-score   support

    accuracy                         0.59       184
   macro avg       0.53      0.53      0.53       184
weighted avg       0.58      0.59      0.58       184
```

The classification report is an helpful metric for getting an inside view of the data points and their performance. The f1-score is the most important metric as it's the combination of both the precision and recall. It's known as the harmonic mean of the precision and recall values. According to this link, the value of 0.69 tells us that this category has been understood by the model is a good manner. While 0.38 value with only about 63 support data points tells us that the data points were less for the model to fully differentiate. By adding more examples for nontext data maybe this can be improved.

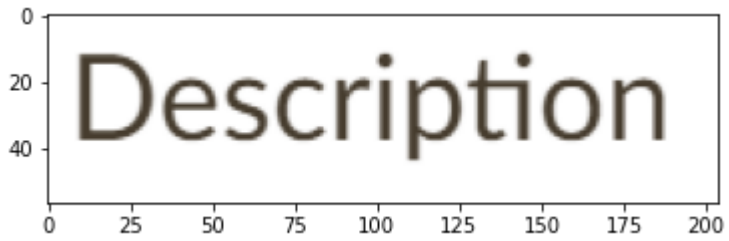
Predicting on Unseen Data

Now, that we have evaluated the model, it's time for predicting new images by our trained model. For this, the function `predict` takes in the image in a jpg format, preprocesses it similar to our step above in the script. And, finally predicts the image's category by our model.

In [28]:
```
def predict(img):
    img_read = cv.imread(img)
    image = load_img(img, target_size = (48,48))
    new_img = img_to_array(image)

    #array to be reshaped 1, and with the shape of our img.shape
    new_img = new_img.reshape((1,) + new_img.shape)
    #normalizing its pixel values
    new_img /= 255.
    #plt.imshow(img)

    #predicting
    labels = test_set.class_indices
    labels = {v: k for k, v in labels.items()}

    #Predict the inputs on the model
    yhat = modelvgg.predict(new_img)
    yhat = np.argmax(yhat)

    return plt.imshow(img_read), labels[yhat]
```

In [29]:
```
predict(img = '2022-09-24 16_10_25-CNN - Exercise.png')
```

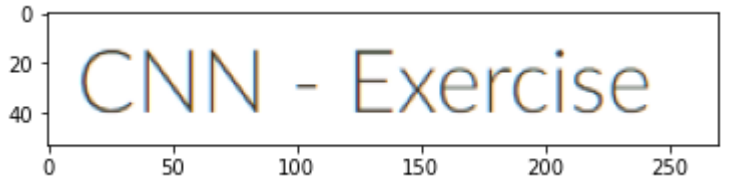
Out[29]: (<matplotlib.image.AxesImage at 0x19d0345fac0>, 'TextDataset')



The above image has been taken from the canvas page for testing. The model's prediction that this image consists of text is infact correct.

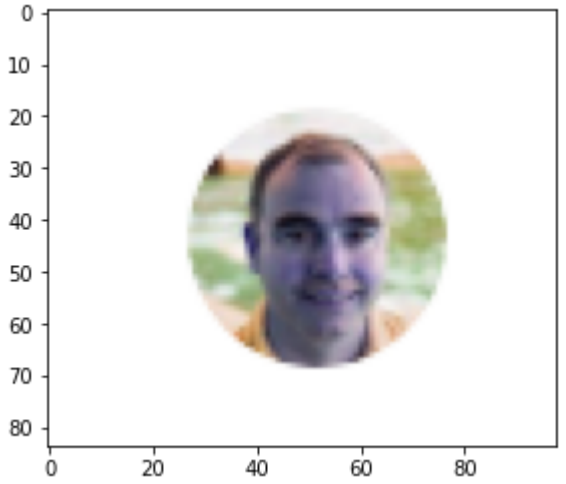
In [30]:
```
predict(img = '2022-09-24 19_35_36-CNN - Exercise.png')
```

Out[30]: (<matplotlib.image.AxesImage at 0x19d034bc5e0>, 'TextDataset')



In [31]:
```
predict(img = '2022-09-24 21_40_48-AI-A-RB-CMK.png')
```

Out[31]: (<matplotlib.image.AxesImage at 0x19d03510280>, 'NonTextDataset')



The above image is of my technical tutor, the model classified it correctly as a nontextdata.

Images that were misclassified

Lastly, I'll be looking into the images from the test set that have been misclassified. For this, again I'll be predicting the values and comparing them to the actual label.

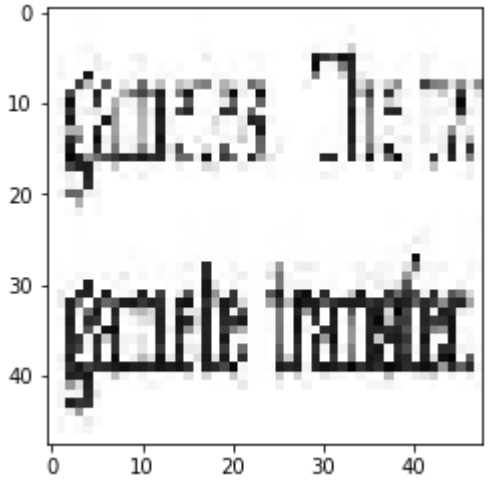
In [133…

```
#creating list of the images that are misclassified by the model
images = []
labels = []
predicted_val = []

#creating list for the images that are correctly classified and storing their labels
correct_images = []
correct_labels = []
correct_pred_val=[]


classlabels = test_set.class_indices
classlabels = {v: k for k, v in classlabels.items()}

for i in range(14):
    img, label = next(test_set) #returning the next item in the iteration
    img = img_to_array(img[i])
    img = np.expand_dims(img, axis=0)
    result = modelvgg.predict(img)
    result = np.argmax(result)
    result = classlabels[result]
    label = np.argmax(label)
    label = classlabels[label]
    if result != label:
        images.append(img)
        labels.append(label)
        predicted_val.append(result)
    if result == label:
        correct_images.append(img)
        correct_labels.append(label)
        correct_pred_val.append(result)
```

In [33]:

```
plt.imshow(np.squeeze(images[0]))
print(f'True Value: {labels[0]}\nPredicted Value: {predicted_val[0]}')
```

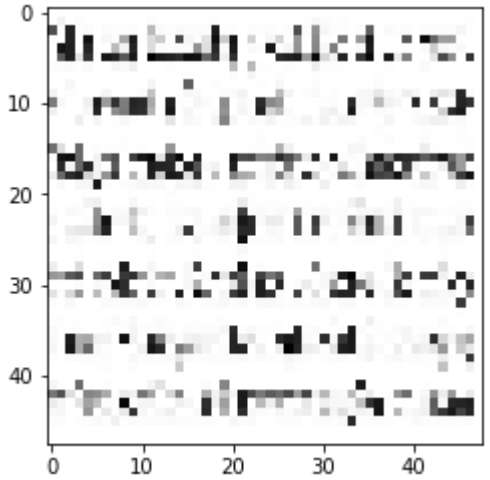
```
True Value: NonTextDataset
Predicted Value: TextDataset
```



The above image is a sample from the test dataset which has been classified wrongly. During the exploratory data analysis, it was visible that there were some images with labelled diagrams. Similarly, in this image, there are some textual words written but also include diagrams. They have been blurred due to the reduction in shape and normalizing the pixel values. This is an interesting data sample, as by human eyes, I would also classify this as a textual image.

In [34]:
```
plt.imshow(np.squeeze(images[1]))
print(f'True Value: {labels[1]}\nPredicted Value: {predicted_val[1]}')
```

```
True Value: NonTextDataset
Predicted Value: TextDataset
```

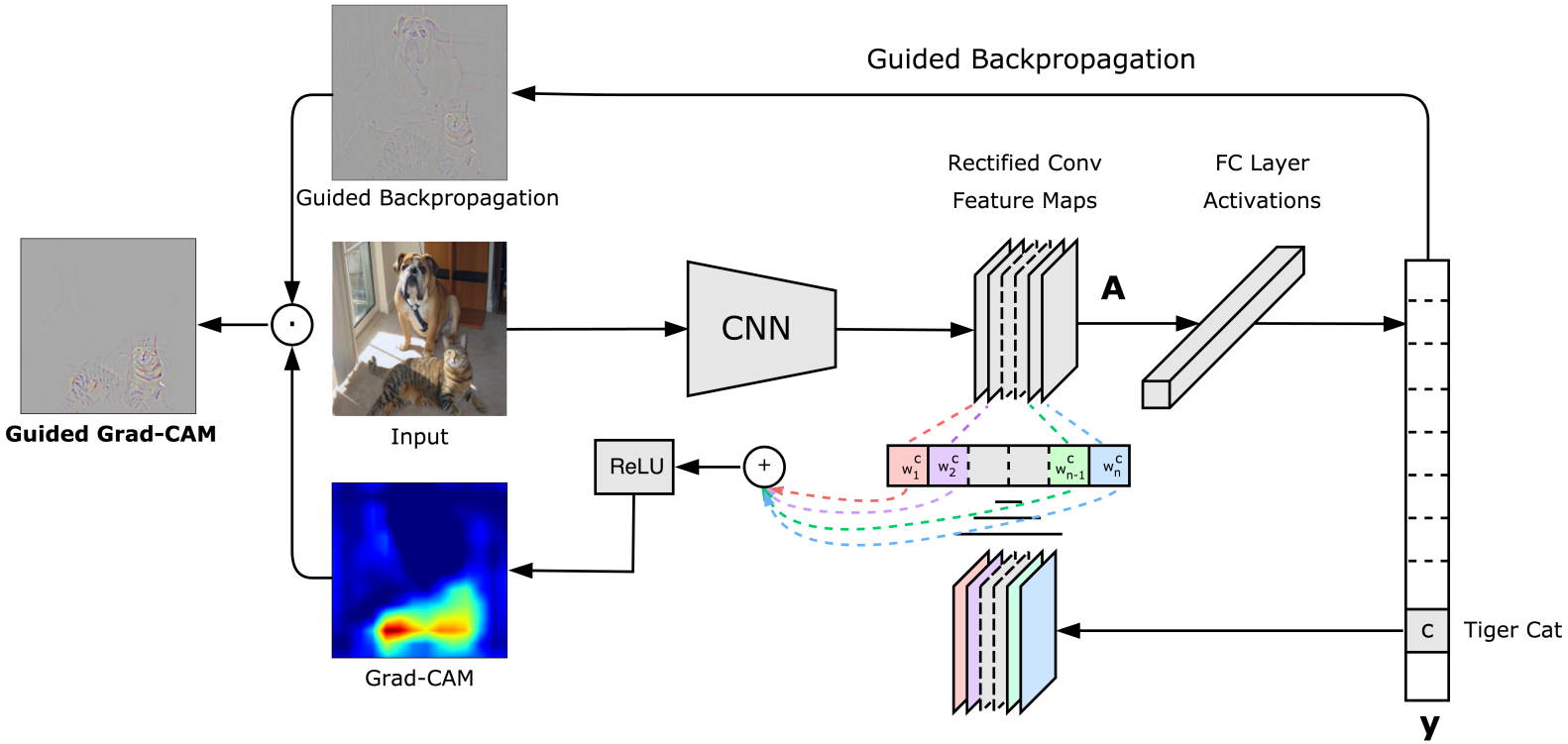


The above image clearly includes some textual data but it's difficult for the human eyes to read after the preprocessing. But the model has predicted it right as a textual data since there's no diagram or nontextual present.

Applying Grad-CAM

In this section, I'll be implementing the GRAD-CAM on different images from the classes: `NonTextData` and `TextData` in order to understand how our model VGG16 interprets and make sits predictions. From the following image, the reader an understand the working of the Grad-CAM on the images distribution.

In [36]:
```
display(Image('http://gradcam.cloudcv.org/static/images/network.png'))
```



Model explanability is a vital topic while working with complex models such as Deep Learning or pre - trained models. They help achieve high accuracy but to understand how and why the models predicts what it does is an exciting knowledge as it can aid in fine - tuning the model. This is done by understand the following aspects:

Since, in CNN's the last layer is the decision defining layer therefore, these neurons store some probabilities of image for each category.

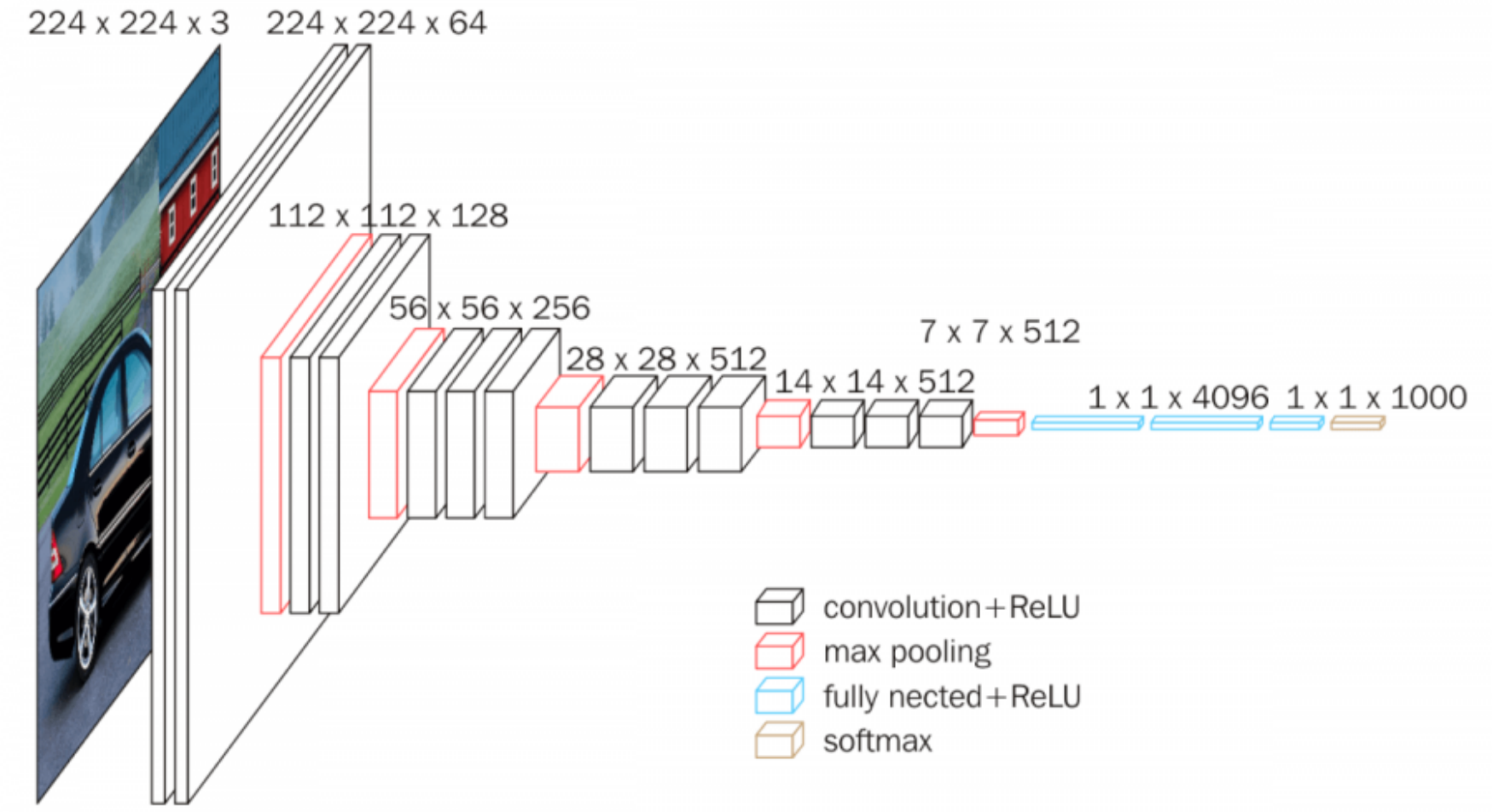
The Grad-CAM works by looking for the gradient information of the images in each neurons and assigns a number to each by following the steps:

1. Highlighting the important pixels of predictions
2. Visualizing the partial derivatives of the predicted scores with respect to the pixel's intensities
3. Deconvolution by making the modifications to the raw gradients
4. Methods that synthesize images to maximally activate a network unit
5. Inverting the latent representation of the images

In addition to following the above steps, it's required to conduct class discrimination i.e, localizing the category in the image and print the image in high resolution so that we can identify the minute grains and pixels of the image.

The advantages of using Gradient - Weighted Class Activation Mapping (GradCAM) are they include class specific localization, generate understanding visualization of the models and it doesn't require retraining of the model.

In [37]:
```
display(Image("https://neurohive.io/wp-content/uploads/2018/11/vgg16-1-e1542731207177.png"))
```



From the above image, the architecture of the VGG16 pre-trained model can be understood. We can conclude that for our VGG16, the modified parameters are:

- Image input size: (48,48,3)
- Last Convolutional Layer Name: block5_conv3
- Heatmap Dimensions: (14,14)

In [287…

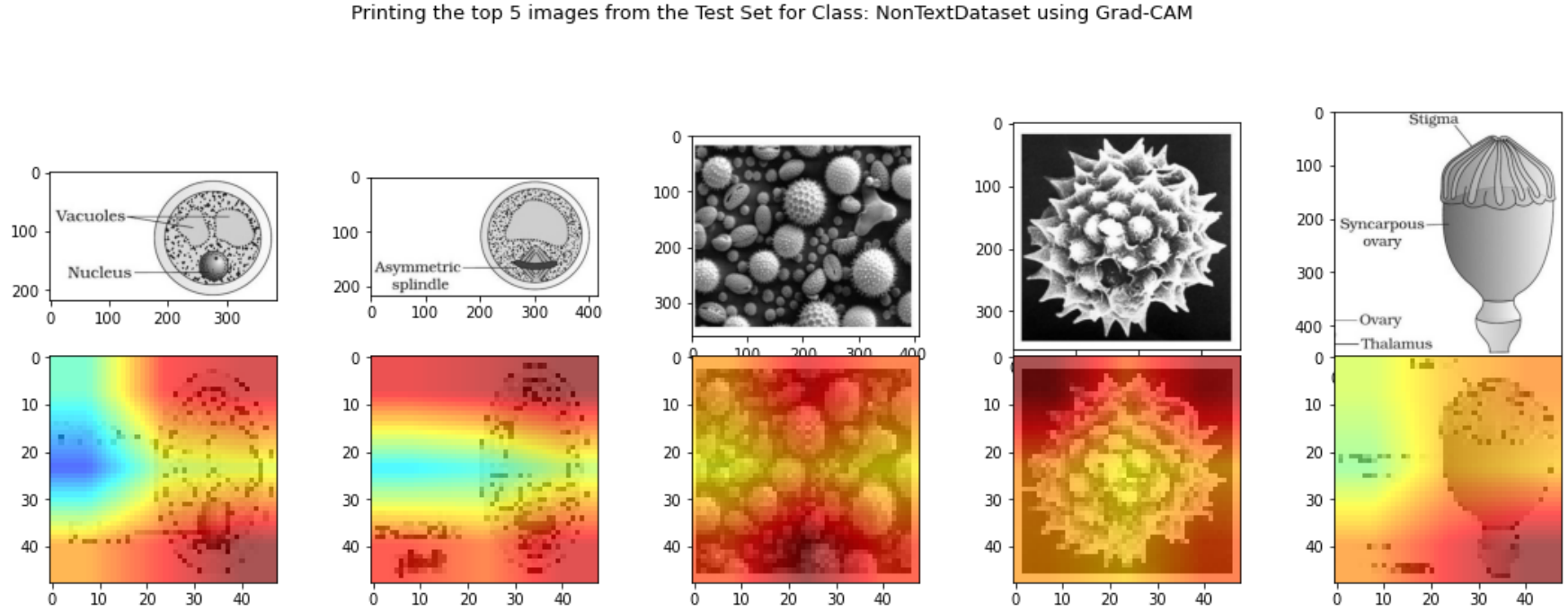
```
#setting the learning phase of the vgg16 to be in the training phase
K.set_learning_phase(1)

def plot_grad_cam(modelname, pic_array, layer_name):

    #firstly, making image as an input and preprocessing it to make it compatible with the model
    pic = np.expand_dims(pic_array, axis=0)
    pic = pic.astype('float32')
    processed_img = pic / 255.0


    #Use the model to make predictions on the sample image and decode the prediction
    yhat = modelname.predict(processed_img)
    label = np.argmax(yhat[0])
    class_output = modelname.output[:, label]

    #finding the gradients of the target class score with respect to the feature maps of the last convolutional layer
    #it helps us to see how the distribution of neurons is done at the last prediction layer
    #the output is a tensor value that is used for further steps
    conv_output = modelname.get_layer(layer_name).output
    grads = K.gradients(class_output, conv_output)[0]

    #taking the image as the input
    #the gradients are global average pooled values to get the neuron decided weights assigned to each classes
    #returns back the pooled gradients along with the activation maps from the last convolution layers
    gradient_function = K.function([modelname.input], [conv_output, grads])

    output, grads_val = gradient_function([processed_img])
    output, grads_val = output[0], grads_val[0]

    # Average the weights and multiply the output of the layer
    #this helps us to learn about the final category prediction by the model
    weights = np.mean(grads_val, axis=(0, 1))
    layer_output = np.dot(output, weights)

    # Create an image and combine it into a heat map
    #Then dividing each of the intensity values of the heatmap with the maximum value for normalizing the values of heatm
    #applying the rELU on the resulting heating map in order to keep the important features from the last layer
    layer_output = cv2.resize(layer_output, (48,48), cv2.INTER_LINEAR)
```

```
    layer_output = np.maximum(layer_output, 0)
    layer_output = layer_output / layer_output.max()

    #Finally, un decode the resulting heatmap to match the dimensions, shape of the input image
    # and add the heatmap colors on it
    heatmap = cv2.applyColorMap(np.uint8(255 * layer_output), cv2.COLORMAP_JET)
    heatmap = cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB)   #convert color of the heatmap to an RGB format
    heatmap = (np.float32(heatmap) + pic / 2)   #add the background color of the heatmap to the plotted image
    return heatmap
```

Printing the top 5 images from the Test Set from the different categories

In [288…

```
#Plotting the first 30 images from our NonText directory
fig=plt.figure(figsize=(15, 9))
for i in range(1,6):
    img = load_img(test_dir + 'NonTextDataset/' + os.listdir(test_dir + 'NonTextDataset/')[i])
    fig.add_subplot(2,5,i)
    plt.imshow(img, cmap = "gray")

    pic_array = img_to_array(load_img(test_dir + 'NonTextDataset/' + os.listdir(test_dir + 'NonTextDataset/')[i], target_
    pic = pic_array.reshape((1,) + pic_array.shape)
    array_to_img(pic_array)
    picture = plot_grad_cam(modelvgg, pic_array, 'block5_conv3')
    picture = picture[0,:,:,]
    array_to_img(picture)
    fig.add_subplot(1,5,i)
    plt.imshow(array_to_img(picture), cmap = "gray")
fig.tight_layout()
plt.suptitle('Printing the top 5 images from the Test Set for Class: NonTextDataset using Grad-CAM',size=13);
```

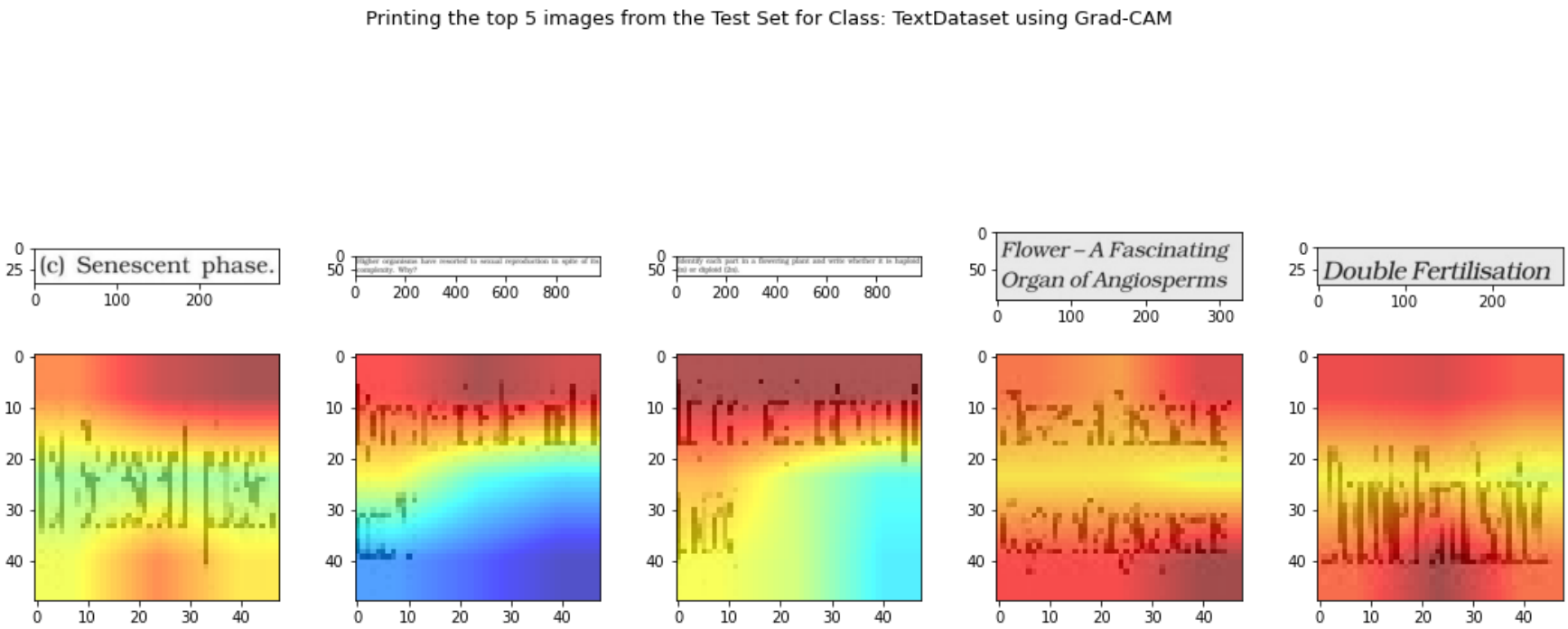


Analysis: From the above plot, it can be inferred the first five images from the test set for category NonTextDataset. It's understood that there are some images from this category that include not just diagrams but also labeled texts. Using the Grad-CAM algorithm, it becomes clear for the user and builder for interpreting the results. The red color represent where the model focused the most while the blue area is with the least focus. This is correctly done as for the first image, VGG16 focused on the diagram part of the image more than the labeled part. We can see a similar trend in higher degree in the fifth image as well.

In [266…

```
#Plotting the first 5 images from our NonText directory
fig=plt.figure(figsize=(15, 9))
for i in range(1,6):
    #plotting the original images from the nontext category
    img = load_img(test_dir + 'TextDataset/' + os.listdir(test_dir + 'TextDataset/')[i])
    fig.add_subplot(2,5,i)
    plt.imshow(img, cmap = "gray")

    #preprocessing the image to an array and plotting the grad-cap heatmap
    pic_array = img_to_array(load_img(test_dir + 'TextDataset/' + os.listdir(test_dir + 'TextDataset/')[i], target_size =
    pic = pic_array.reshape((1,) + pic_array.shape)
    array_to_img(pic_array)
    picture = plot_grad_cam(modelvgg, pic_array, 'block5_conv3')
    picture = picture[0,:,:,]
    array_to_img(picture)
```

```
    fig.add_subplot(1,5,i)
    plt.imshow(array_to_img(picture), cmap = "gray")
fig.tight_layout()
plt.suptitle('Printing the top 5 images from the Test Set for Class: TextDataset using Grad-CAM',size=13);
```



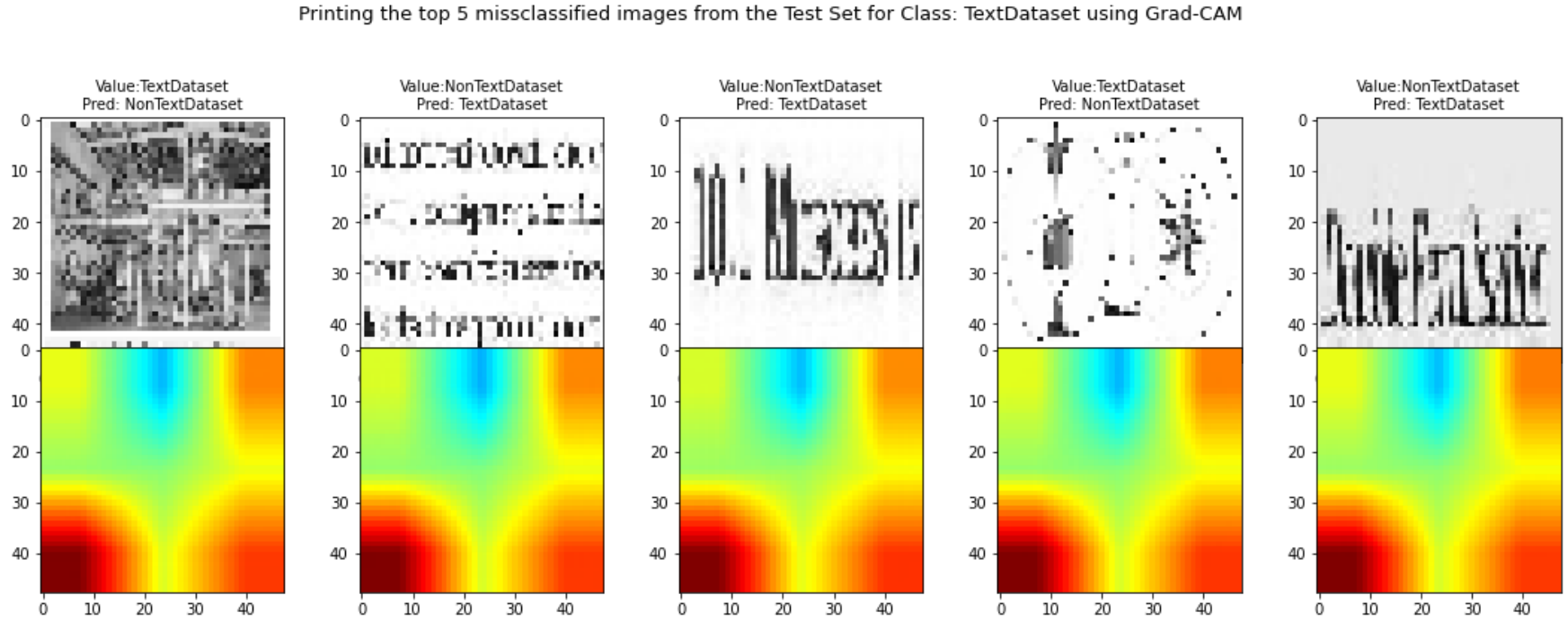
In the above plot, the top five images from the test set for images including the textual data are illustrated. By using Grad-CAM, we can see that the VGG16 model trained focused more on the textual part than the empty white space. As we can see the perfect representation of it in the second image. But with the images which have textual data present all over the shape, the heatmap's color is spread more red color. This is interesting to understand as by human knowledge it's easy to follow and conclude.

Printing the test images that are classified right and some that are classified wrongly by VGG16

In [273…

```
#Plotting the first 30 images from our NonText directory
fig=plt.figure(figsize=(15, 9))
for i in range(1,6):
    plt.subplot(2,5,i)
    plt.imshow(np.squeeze(images[i]))
    plt.title(f'Value:{labels[i]}\nPred: {predicted_val[i]}', size = 10)
```

```
    pic_array = np.squeeze(images[i])
    pic = pic_array.reshape((1,) + pic_array.shape)
    array_to_img(pic_array)
    picture = plot_grad_cam(modelvgg, pic_array, 'block5_conv3')
    picture = picture[0,:,:,]
    array_to_img(picture)
    fig.add_subplot(1,5,i)
    plt.imshow(array_to_img(picture), cmap = "gray")

fig.tight_layout()
plt.suptitle('Printing the top 5 missclassified images from the Test Set for Class: TextDataset using Grad-CAM',size=13);
```

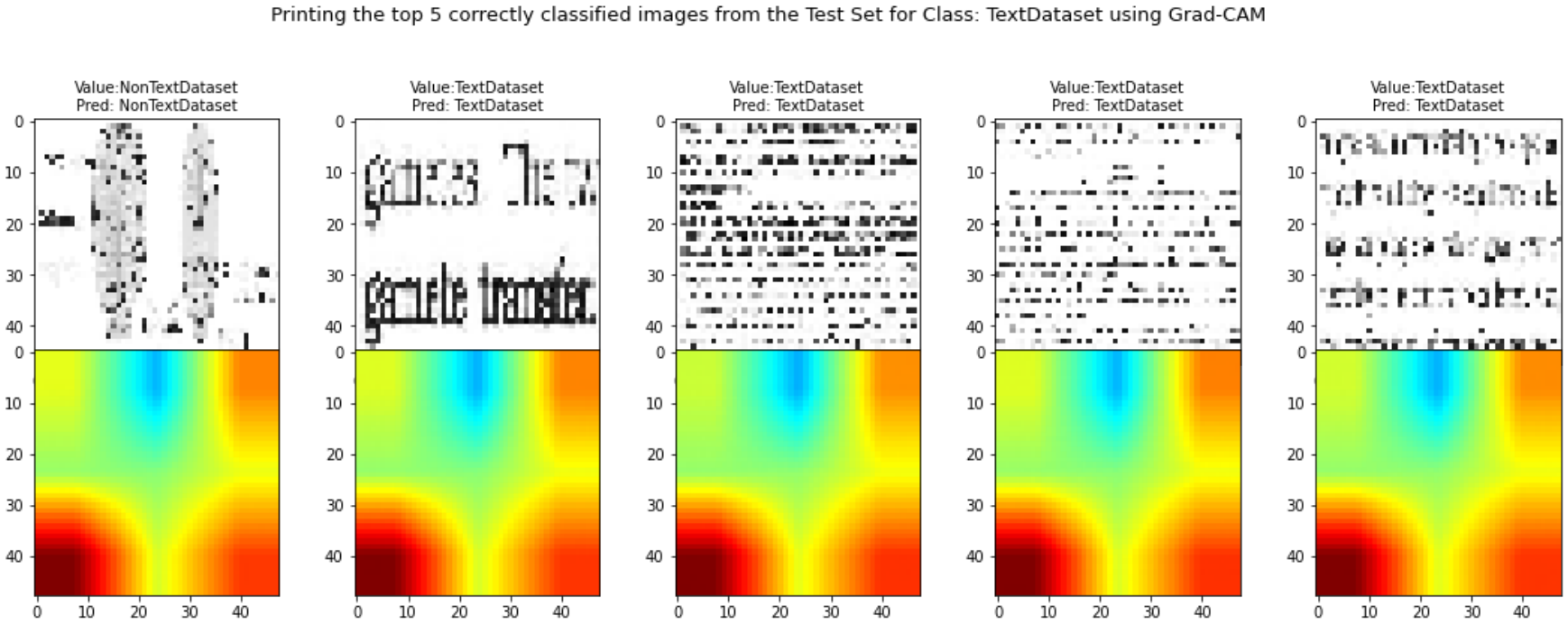


Analysis: From the above plot, we can see the top 5 images which are misclassified by the trained VGG16 model. Since, the model has been trained on images with shape 48x48, the image's resolution and clearity has been reduced with it being the input of the last convolutional layer. Since, the last layer used *block5_conv3* gives out the ouput with the most important features for giving the final output of the image. The results of Grad-CAM are mostly similar to each one of the images. This can be possible due to the following reasons:

1. The size of the images being small (48x48)
2. Since, the last convolutional layer gives the transformed image after being passed through different layers, the data inside it reduces the changes for showing the grains in it.

In [276…

```
#Plotting the first correctly classified 5 images using Grad-CAM
fig=plt.figure(figsize=(15, 9))
for i in range(1,6):
    plt.subplot(2,5,i)
    plt.imshow(np.squeeze(correct_images[i]))
    plt.title(f'Value:{correct_labels[i]}\nPred: {correct_pred_val[i]}', size = 10)

    pic_array = np.squeeze(correct_images[i])
    pic = pic_array.reshape((1,) + pic_array.shape)
    array_to_img(pic_array)
    picture = plot_grad_cam(modelvgg, pic_array, 'block5_conv3')
    picture = picture[0,:,:,]
    array_to_img(picture)
    fig.add_subplot(1,5,i)
    plt.imshow(array_to_img(picture), cmap = "gray")

fig.tight_layout()
plt.suptitle('Printing the top 5 correctly classified images from the Test Set for Class: TextDataset using Grad-CAM',siz
```



Analysis: From the above plot, we can see the images for the top 5 correctly classified images for the category of textualdataset. By using Grad-CAM we can interpret the results are similar with more concentration towards the left side and right side of the bottom part of the

image. Again, due to the fact that the images extracted from the last layer contain only the important features and shuts down the entire gradients.

Conclusions & Recommendations

This was a great experiment of working with image data for classification. The dataset was very interesting and most importantly, fine-tuning VGG16 model was a great learning experiment. I learnt about how the data size and affect the number of dense layer neuron, dropout rate towards the accuracy and loss of a model. For future recommendations, I would like to see add more data points for nontextual images and test how it would differ.

Additionally, adding the functionality of Grad-CAM has helped me understand how the model classifies the images into the two different categories. By making the heatmap on images from test for different categories, it was interesting to learn by plotting the heatmap and interpreting the distribution of colors red and blue. While plotting the images for the last layer of the convolutional layer, it was a great exercise for checking for concentration of the model. Overall, Grad-CAM is a great methology and I will be using this for my future projects to understand the model's decision making.

Resources Used

- https://www.analyticsvidhya.com/blog/2020/08/top-4-pre-trained-models-for-image-classification-with-python-code/
- https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate
- Semester 04- genuine challenge 01: https://github.com/A-shukla12/Predicting-the-Attention-Rate-of-Patients-by-Monitoring-their-Emotions-
- https://keras.io/examples/vision/grad_cam/
- https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-gradcam-554a85dd4e48
- https://github.com/tabayashi0117/Anomaly-Detection

In [ ]: