SORBONNE UNIVERSITÉ

Is it possible to recognize a painting of Paul Cézanne from other famous painters more or less inspired by him?

Project Report

VCC

Student: Mojan Jamal Omidi

Student ID: 21120651

Supervisor: Professor Henri Maître

5/22/2022

Abstract

With the expeditious digitization of artworks from different sectors and with the striking advancement in the field of artificial intelligence, many art-classification tasks and annotations that used to consume an unnecessary amount of time and energy being done manually in the traditional way, can now be done using neural networks, without a vital need for expertise in art-related domains.

In this project, I develop a model that can distinguish the paintings of Cezanne from other artists who were to some extent inspired by him, using deep learning and convolutional neural networks. Due to the small size of the dataset, it was unfeasible to train a full-scale model from scratch. so, I used the VGG16 pretrained model as my base model and carried out different fine-tuning experiments on it using the provided dataset of the paintings.

The best accuracy I could achieve using this model was 98 percent. In addition, I did experiment with ResNet50 pretrained model as well, but the results were not as satisfactory. A comparison table will be provided in this report.

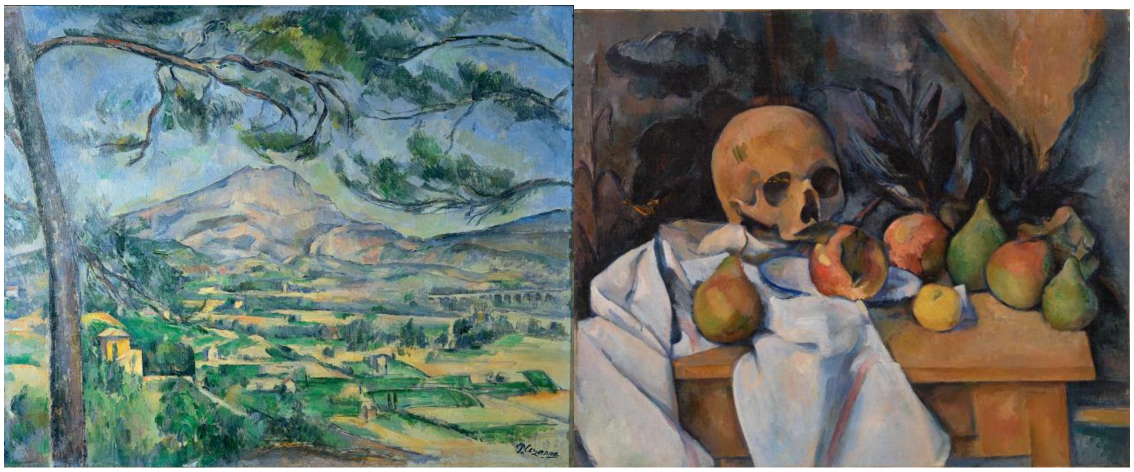
Table of Contents

Introduction

The transition of artwork towards digitization has gained significant importance. The recognition and classification of artwork is an imperative task for monitoring and understanding purposes. People express their emotions, and by gaining enough experience and knowledge in related spheres, people – specially art experts can learn to perceive the differences and separate different artists, painting styles and genres from one another. However, to computers a painting is just an array of pixel values, they do not comprehend abstract ideas such as emotions or delicacy. Many experiments have been carried out to teach computers these fine characteristics to be able to automate art classification tasks. By employing k-nearest neighbor and support vector machine models, many subject matter experts managed to realize many painting classification tasks like distinguishing between Baroque, Impressionism, Post-Impressionism styles, but recently, a significant number of papers have been published in which authors rely on Convolutional Neural Networks to solve classification problems rather than the traditional approach. Many CNN models with different architectures have been proposed which can achieve state-of-the-art performance on ImageNet. But there is a potential constraint that one might come across while working with CNNs, which is its need for large datasets with labeled images which is sometimes hard to collect for some tasks. As a matter of fact, this is the main obstacle I encountered while working on this project. The number of paintings were too few for the model to learn and perform well, which resulted in overfitting. This report mainly focuses on approaches I took to tackle this problem; I eventually increased the size of my dataset by converting each painting into many sub-images which satisfied the network's demand for data.

Aesthetics

It is very difficult for normal human beings to classify paintings of different artists. However, there are some features that make Cezanne's paintings distinct from other painters, like his recognizable color palette. Cezanne had a preference for planes of saturated hues. This tonal treatment is particularly pronounced in his landscapes. In these paintings, blue shadows often help unify the surface.



Cezanne embraced a unique method of paint application, he employed “Constructive strokes”, meticulously arranged marks that worked together to create geometric forms. He also avoided the use of dark lines and relied on this contrasting brushwork to define the outlines of objects when their points of contact are tenuous and delicate. All colors often have an equal intensity and this, combined with the brushstrokes, tends to flatten the space as there is no distinction between near objects and far ones.

Up until years ago, these subtle characteristics used to be only easy to grasp for art experts, but recently with the visible boost in translating physical artwork into digital image format and subsequently, with the abundance of the available data, many classification tasks that were impossible to carry out in practice due to insufficient amount of data, are now attainable with the help of state-of-the-art convolutional neural network architectures.

Project Complications

As mentioned earlier, the main obstacle in this project is the size of the dataset which is overly small. The small size of the dataset impacts our training process drastically. Considering the number of paintings in the dataset, running a large number of iterations can result in overfitting. A large dataset helps with avoiding overfitting and generalizes better as it captures the inherent data distribution more effectively.

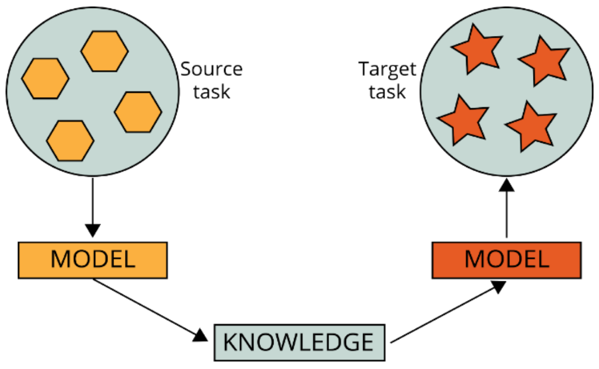
Another obstacle would be the size of our images. Convolutional neural networks are designed to take square images as input. So, I had to convert our images into square-shaped ones while maintaining the aspect ratio. In addition, I had to reduce the size of the images before feeding them to the neural network without losing too much data, since keeping subtle features like color palettes, as mentioned before, are crucial in art classification challenges.

Making Use of Pretrained Models

To address the insufficient amount of data problem, instead of training a model from scratch I used pretrained models.

A convolutional neural network that has been trained on a related large-scale problem such as ImageNet can be used in other visual recognition tasks without the need to train the first few layers. Those fixed layers are fixed feature detectors.

The upper layers can be fine-tuned to match the current problem at hand. This is normally done to speed up the learning and reduce the need for very large training datasets. I only have to teach these models to classify the images using the features they extract from the paintings. I had to choose a pretrained model that is deep enough for the subtle task of art classification.



At first, I was under the assumption that this alone would be enough to fix the problem of the small dataset. However, this alone did not turn out to be enough to achieve a high accuracy.

Also, initially my hypothesis was that if I choose deeper models like ResNet50 my model would perform better, but to my surprise, VGG16 performed much better than ResNet50 and lead to a much higher accuracy.

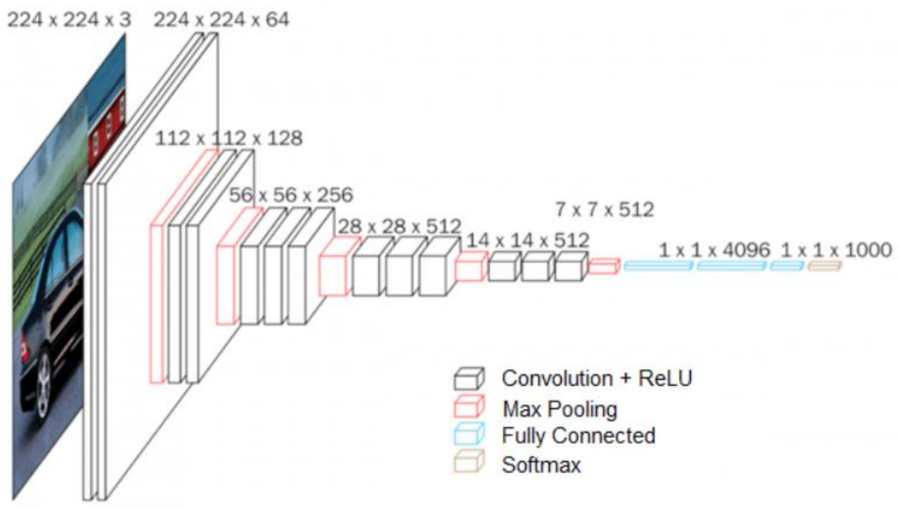
Architectures

Keras contains 10 pretrained models along with their pretrained weights for image classification which are trained on ImageNet data; one of them is the VGG model. VGG has two different CNN models, a 16-layer model and a 19-layer model. VGG models are very powerful models and they are used both as image classifiers and as the basis for new models that use image inputs. For this project I use VGG16 as my starting point. However, I did experiment with ResNet50 as well.

VGG16 Pretrained Model:

The VGG model can be loaded and used in the Keras deep learning library. Keras provides an application interface for loading and using pre-trained models. After creating the models by running the necessary code, Keras downloads the weights files and stores them in the *~/.keras/models* directory. The weights are about 528 megabytes, so it could take a few minutes to download them for the first run. With the next runs of the same block, the weights are loaded locally and the models will be ready for us within a couple of seconds.

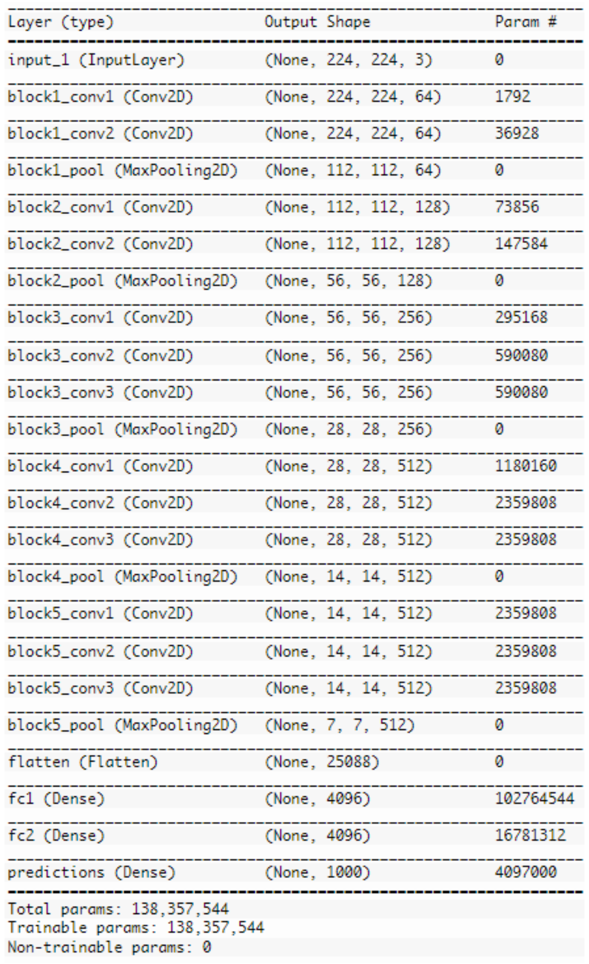
The picture below shows the structure of VGG16:



In the given figure, we can observe the initial structure of the model before modification. By default, the model expects images as input 224x224 pixels with three channels (R, G, B), but throughout this project, I modify the input shape based on the different scenarios that I carry out.

The only pre-processing done is normalizing the RGB values for every pixel. This is achieved by subtracting the mean value from every pixel.

Image is passed through the first stack of 2 convolutional layers of the very small receptive size of 3 x 3, followed by ReLU activations. Each of these two layers contains 64 filters. The convolution stride is fixed at 1 pixel, and the padding is 1 pixel. This configuration preserves the spatial resolution, and the size of the output activation map is the same as the input image dimensions. The activation maps are then passed through spatial max pooling over a 2 x 2-pixel window, with a stride of 2 pixels. This halves the size of the activations. Thus, the size of the activations at the end of the first stack is 112 x 112 x 64.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
```

The activations then flow through a similar second stack, but with 128 filters as against 64 in the first one. Consequently, the size after the second stack becomes 56 x 56 x 128. This is followed by the third stack with three convolutional layers and a max pool layer. The number of filters applied here are 256, making the output size of the stack 28 x 28 x 256. This is followed by two stacks of three convolutional layers, with each containing 512 filters. The output at the end of both these stacks will be 7 x 7 x 512.

The stacks of convolutional layers are followed by three fully connected layers with a flattening layer in-between. The first two have 4,096 neurons each, and the last fully connected layer serves as the output layer and has 1,000 neurons corresponding to the 1,000 possible classes for the ImageNet dataset. The output layer is followed by the Softmax activation layer used for categorical classification. However, in this project I discard the fully connected layers and construct a new one that is suitable for this task. I modify the input dimensions based on the different scenarios that I carry out throughout this report because I experiment with different image sizes.

Modifying the Original Architecture (Constructing a New Fully-Connected Layer):

I disregarded the fully connected layers by setting include top = False. Basically, I used completely different fully connected and output layers. The output layer is a vector of 2 positions, one representing Paul Cezanne and one representing others.

I constructed the fully connected layer by adding the following layers on top of the model's five convolutional blocks:

- A Flatten layer
- A Dropout layer with the value of 0.3
- A Dense layer of size 256 with rectified linear activation function as its activation function
- A Dense Layer of size 1 with sigmoid function as its activation function because the task is a binary class classification (final layer)

The flatten Layer: Flattening is used to convert all the resultant 2-Dimensional arrays from pooled feature maps into a single long continuous linear vector. The flattened matrix is fed as input to the fully connected layer to classify the image.

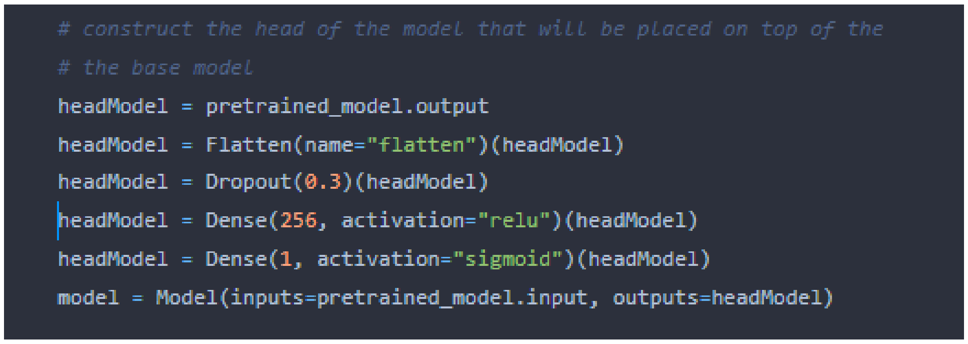
The Dropout Layer: The Dropout layer is a mask that nullifies the contribution of some neurons towards the next layer and leaves unmodified all others.

The Dense Layers: A Dense layer feeds all outputs from the previous layer to all its neurons, each neuron providing one output to the next layer. It's the most basic layer in neural networks.

Relu Function: The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero.

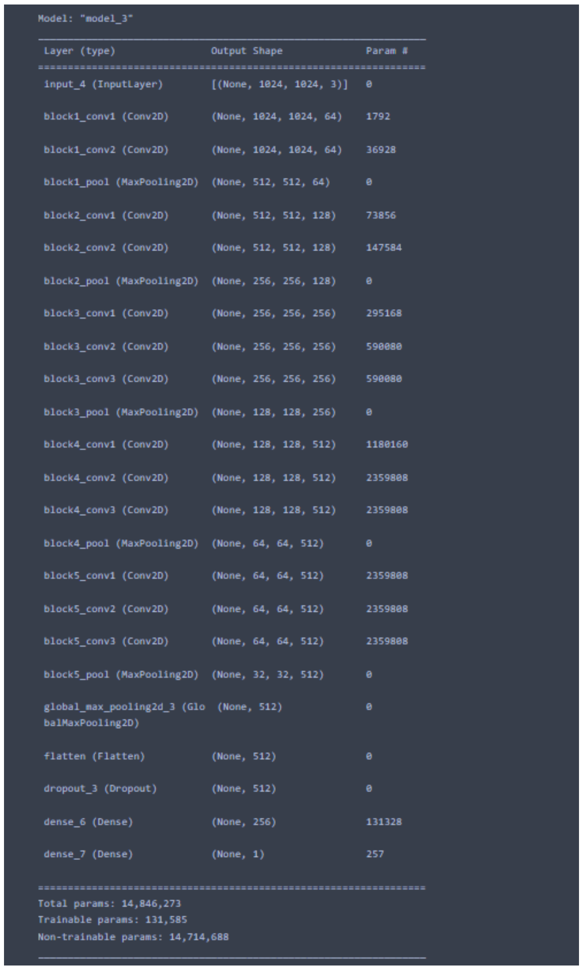
Sigmoid Function: When the activation function for a neuron is a sigmoid function it is a guarantee that the output of this unit will always be between 0 and 1. Also, as the sigmoid is a non-linear function, the output of this unit would be a non-linear function of the weighted sum of inputs. Sigmoid is equivalent to a 2-element Softmax, where the second element is assumed to be zero. Therefore, sigmoid is mostly used for binary classification.

The following figure shows the block of code that implements this construction.



```
# construct the head of the model that will be placed on top of the
# the base model
headModel = pretrained_model.output
headModel = Flatten(name="flatten")(headModel)
headModel = Dropout(0.3)(headModel)
headModel = Dense(256, activation="relu")(headModel)
headModel = Dense(1, activation="sigmoid")(headModel)
model = Model(inputs=pretrained_model.input, outputs=headModel)
```

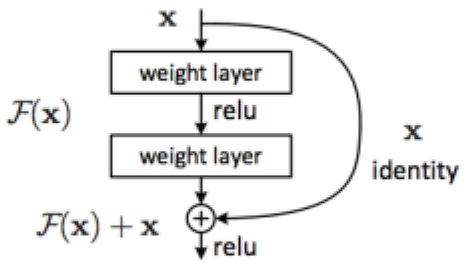
The summary of the architecture of the final model is shown below. This is the architecture I am going to experiment with in this report. As you can see, this architecture has total number of **14,846,273** parameters.



```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_4 (InputLayer)        [(None, 1024, 1024, 3)]   0         
                                                                 
 block1_conv1 (Conv2D)       (None, 1024, 1024, 64)    1792      
                                                                 
 block1_conv2 (Conv2D)       (None, 1024, 1024, 64)    36928     
                                                                 
 block1_pool (MaxPooling2D)  (None, 512, 512, 64)      0         
                                                                 
 block2_conv1 (Conv2D)       (None, 512, 512, 128)     73856     
                                                                 
 block2_conv2 (Conv2D)       (None, 512, 512, 128)     147584    
                                                                 
 block2_pool (MaxPooling2D)  (None, 256, 256, 128)     0         
                                                                 
 block3_conv1 (Conv2D)       (None, 256, 256, 256)     295168    
                                                                 
 block3_conv2 (Conv2D)       (None, 256, 256, 256)     590080    
                                                                 
 block3_conv3 (Conv2D)       (None, 256, 256, 256)     590080    
                                                                 
 block3_pool (MaxPooling2D)  (None, 128, 128, 256)     0         
                                                                 
 block4_conv1 (Conv2D)       (None, 128, 128, 512)     1180160   
                                                                 
 block4_conv2 (Conv2D)       (None, 128, 128, 512)     2359808   
                                                                 
 block4_conv3 (Conv2D)       (None, 128, 128, 512)     2359808   
                                                                 
 block4_pool (MaxPooling2D)  (None, 64, 64, 512)       0         
                                                                 
 block5_conv1 (Conv2D)       (None, 64, 64, 512)       2359808   
                                                                 
 block5_conv2 (Conv2D)       (None, 64, 64, 512)       2359808   
                                                                 
 block5_conv3 (Conv2D)       (None, 64, 64, 512)       2359808   
                                                                 
 block5_pool (MaxPooling2D)  (None, 32, 32, 512)       0         
                                                                 
 global_max_pooling2d_3 (Glo  (None, 512)              0         
 balMaxPooling2D)                                                
                                                                 
 flatten (Flatten)           (None, 512)               0         
                                                                 
 dropout_3 (Dropout)         (None, 512)               0         
                                                                 
 dense_6 (Dense)             (None, 256)               131328    
                                                                 
 dense_7 (Dense)             (None, 1)                 257       
                                                                 
=================================================================
Total params: 14,846,273
Trainable params: 131,585
Non-trainable params: 14,714,688
_________________________________________________________________
```

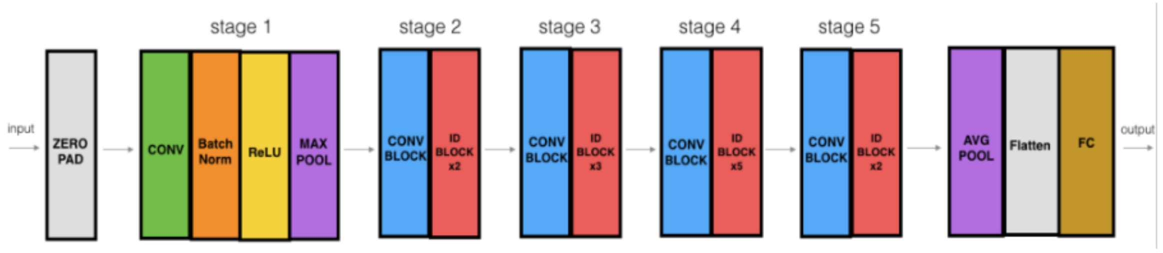
ResNet50 Pretrained Model

Deep Convolutional neural networks have a major disadvantage that is ‘Vanishing Gradient Problem’. During backpropagation, the value of gradient decreases significantly, thus hardly any change comes to weights. To overcome this, ResNet is used. ResNet makes use of “SKIP CONNECTION”. ResNet-50 model is a convolutional neural network that is 50 layers deep. These networks use skip connections to overcome the vanishing gradient problem. Skip connection is a direct connection that skips over some layers of the model. The identity matrix transmits forward the input data that avoids the loose of information (the data vanishing problem).


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

According to the papers of I have read ResNet50 does a much better job when it comes to extracting delicate and fine details thanks to its very deep architecture. So, I decided to run a couple of experiments with ResNet50 as well.

The Resnet50 has 48 Convolutional layers along with a MaxPool and an Average Pool layer. The ResNet50 model consists of 5 stages each with a convolution and Identity block. Each convolution block has 3 convolutional layers and each identity block also has 3 convolutional layers. The ResNet-50 has over 23 million trainable parameters.



Modifying the Original Architecture (Constructing a New Fully-Connected Layer):

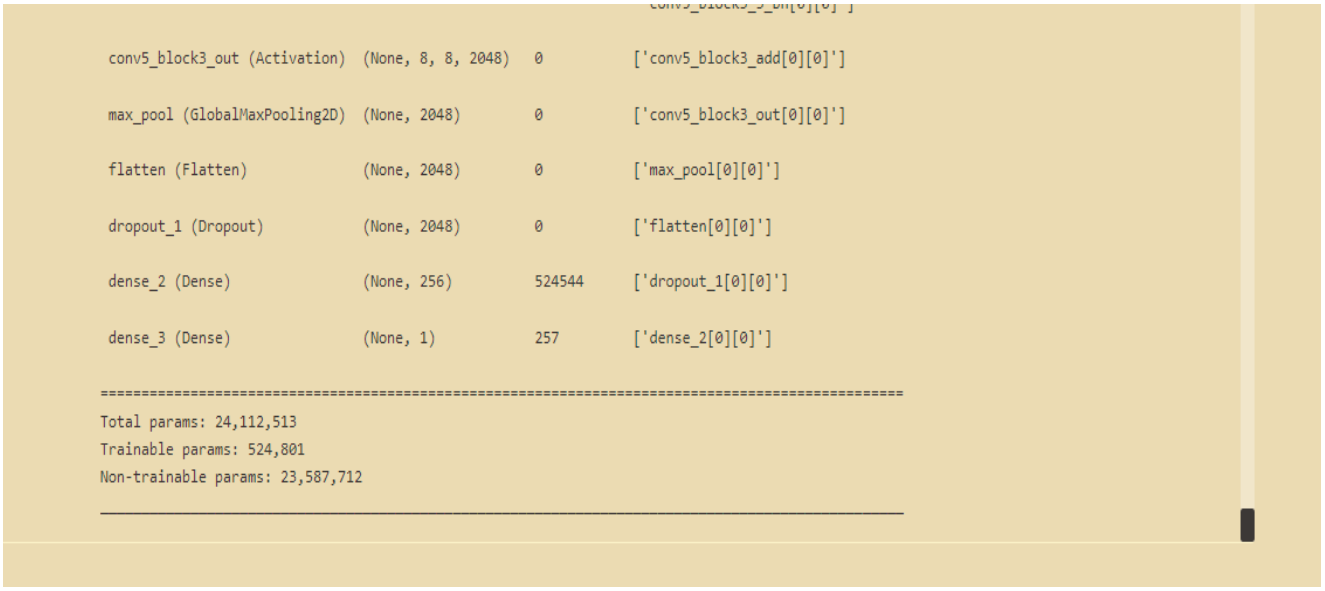
Just like before, I discarded the fully connected layers and only kept the five blocks and added the same fully connected layers that I used for constructing the VGG16 final architecture, on top of the fifth block.

I constructed the fully connected layers by adding the following layers on top the model’s five convolutional blocks:

- A Flatten layer
- A Dropout layer with the value of 0.3

- A Dense layer of size 256 with rectified linear activation function as its activation function
- A Dense Layer of size 1 with sigmoid function as its activation function because the task is a binary class classification (final layer)

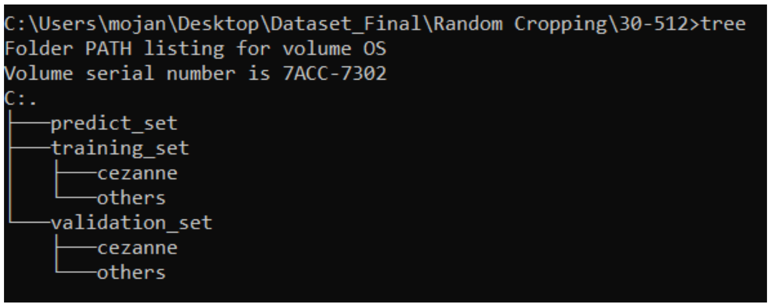
The summary of the architecture of the final model is shown below. This is the architecture I am going to experiment with in this report. As we can see, this architecture has total number of 24,112,512 parameters.



```
 conv5_block3_out (Activation)  (None, 8, 8, 2048)   0           ['conv5_block3_add[0][0]']

 max_pool (GlobalMaxPooling2D)  (None, 2048)         0           ['conv5_block3_out[0][0]']

 flatten (Flatten)              (None, 2048)         0           ['max_pool[0][0]']

 dropout_1 (Dropout)            (None, 2048)         0           ['flatten[0][0]']

 dense_2 (Dense)                (None, 256)          524544      ['dropout_1[0][0]']

 dense_3 (Dense)                (None, 1)            257         ['dense_2[0][0]']

==================================================================================================
Total params: 24,112,513
Trainable params: 524,801
Non-trainable params: 23,587,712
__________________________________________________________________________________________________
```

Dataset and Augmentation

I split my data into a training_set, a validation_set and a predict_set folder, maintaining a 70/30 ratio between my training and validation set. Here we can observe the structure of my dataset:

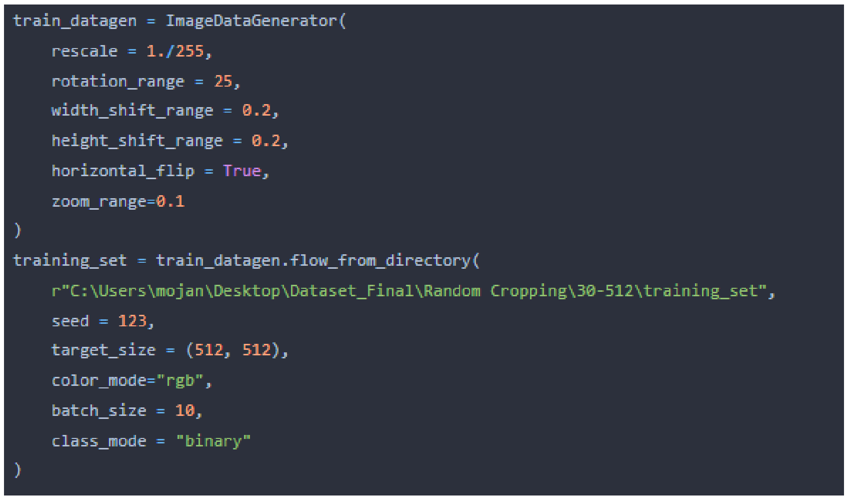


```
C:\Users\mojan\Desktop\Dataset_Final\Random Cropping\30-512>tree
Folder PATH listing for volume OS
Volume serial number is 7ACC-7302
C:.
├───predict_set
├───training_set
│   ├───cezanne
│   └───others
└───validation_set
    ├───cezanne
    └───others
```

In my model, I chose to keep RGB values, as color is an important aspect in identifying the artist. Each RGB value is a given layer, where a pixel is between the number 0–255.

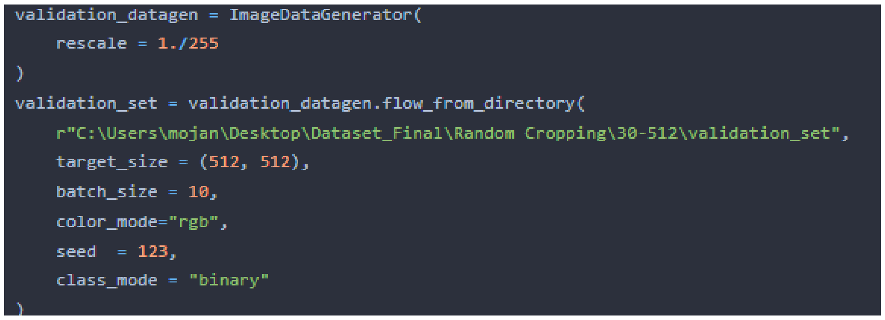
Since the dataset is small, I made use of image augmentation. I augmented the images so that for every epoch a new transformation of every image is generated. Thus, the model sees the same number of images in every epoch (as many as there are in the original training data), albeit a new version of those images each time.

I applied rotation, width and height shift, zoom range and horizontal flip with the values and ranges shown in the picture below:



```
train_datagen = ImageDataGenerator(
    rescale = 1./255,
    rotation_range = 25,
    width_shift_range = 0.2,
    height_shift_range = 0.2,
    horizontal_flip = True,
    zoom_range=0.1
)
training_set = train_datagen.flow_from_directory(
    r"C:\Users\mojan\Desktop\Dataset_Final\Random Cropping\30-512\training_set",
    seed = 123,
    target_size = (512, 512),
    color_mode="rgb",
    batch_size = 10,
    class_mode = "binary"
)
```

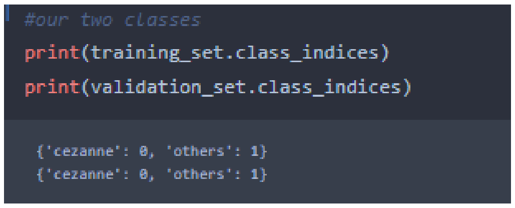
Data augmentation is done only on training set as it helps the model become more generalized and robust. So, there's no point of augmenting the validation set.



```
validation_datagen = ImageDataGenerator(
    rescale = 1./255
)
validation_set = validation_datagen.flow_from_directory(
    r"C:\Users\mojan\Desktop\Dataset_Final\Random Cropping\30-512\validation_set",
    target_size = (512, 512),
    batch_size = 10,
    color_mode="rgb",
    seed  = 123,
    class_mode = "binary"
)
```

I used the batch size of 10. I only have two classes which are Cezanne and others so the class mode is set to binary. Color mode is set to RGB in order to preserve the colors. I chose a specific value as my seed so I would not have to worry about random number generator which can vary weights, changing network dynamics.

As mentioned before, the input dimensions and target_size values change throughout this report depending on different approaches and 512x512 in the images above is an example.



```
#our two classes
print(training_set.class_indices)
print(validation_set.class_indices)

{'cezanne': 0, 'others': 1}
{'cezanne': 0, 'others': 1}
```

Workflow

At first, I froze all the convolutional layers and set their status as untrainable and only trained the fully connected layers and compiled and fit the model once. What is being done here is basically Transfer Learning. I will start fine-tuning the model after this step is done and the model is compiled once.

It is critical to only do the fine-tuning step after the model with frozen layers has been trained to convergence. If I mix randomly-initialized trainable layers with trainable layers that hold pre-trained features, the randomly-initialized layers will cause very large gradient updates during training, which will destroy my pre-trained features. I also kept all of the batch normalization layers frozen and allowed other layers to be altered

For fine-tuning, I included variations of the extent to which the error from the new task is being back propagated within the network or, in other words, how many of the transferred layers are kept frozen. In this report I test four different scenarios which result in different numbers of trainable parameters for each model.

VGG16 Model Parameters:

i) Only the weights of the FC layers are being modified, meaning we have **131,585** trainable parameters (basically transfer learning)

ii) Skip first 4 - the weights of the first four convolutional blocks are kept frozen and only conv_block5 and the FC layers are being trained, meaning we have **7,1211,009** trainable parameters

iii) Skip first 3 - the weights of the first three convolutional blocks are kept frozen and only conv_block_4, conv_block5 and the FC layers are being trained, meaning we have **13,110,785** trainable parameters

iv) All – upon each iteration, the weights of all models are being modified, meaning the status of the entire **14,846,273** parameters is set to trainable

ResNet50 Model Parameters:

i. Only the weights of the FC layers are being modified, meaning we have **524,801** trainable parameters (basically transfer learning)

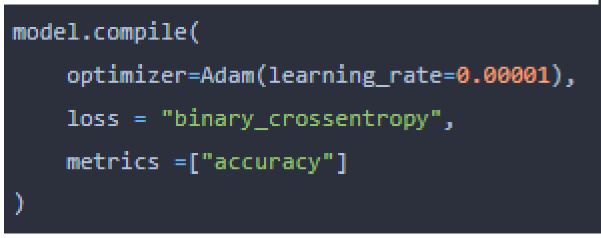
ii. Last Convolutional Block (block 5) - the weights of the first four convolutional blocks are kept frozen and only conv_block5 and the FC layers are being trained, meaning we have **15,500,901** trainable parameters

iii. Last Two Convolutional Blocks (blocks 4 and 5) - the weights of the first three convolutional blocks are kept frozen and only conv_block_4, conv_block5 and the FC layers are being trained, meaning we have **22,609,409** trainable parameters

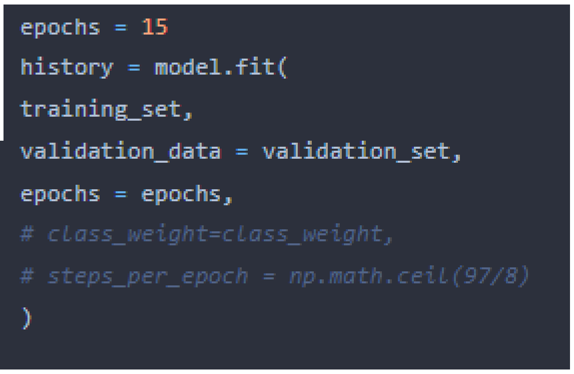
iv. All – upon each iteration, the weights of all models are being modified, meaning the status of the entire **24,059,393** parameters is set to trainable

Model Compiling and Fitting

When it came to compilation and fitting, I chose the same configurations for both the Transfer Learning phase and Fine-tuning phase in both VGG16 and ResNet50 cases. I set the model to run for 15 epochs, with Adam as the optimize and learning rate of 0.00001. The only difference is that I fitted it for 20 epochs for both models



```
model.compile(
    optimizer=Adam(learning_rate=0.00001),
    loss = "binary_crossentropy",
    metrics =["accuracy"]
)
```

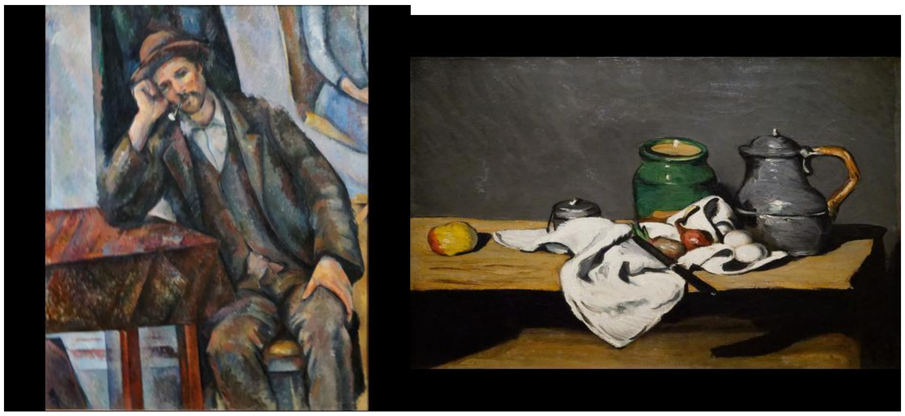


```
epochs = 15
history = model.fit(
training_set,
validation_data = validation_set,
epochs = epochs,
# class_weight=class_weight,
# steps_per_epoch = np.math.ceil(97/8)
)
```

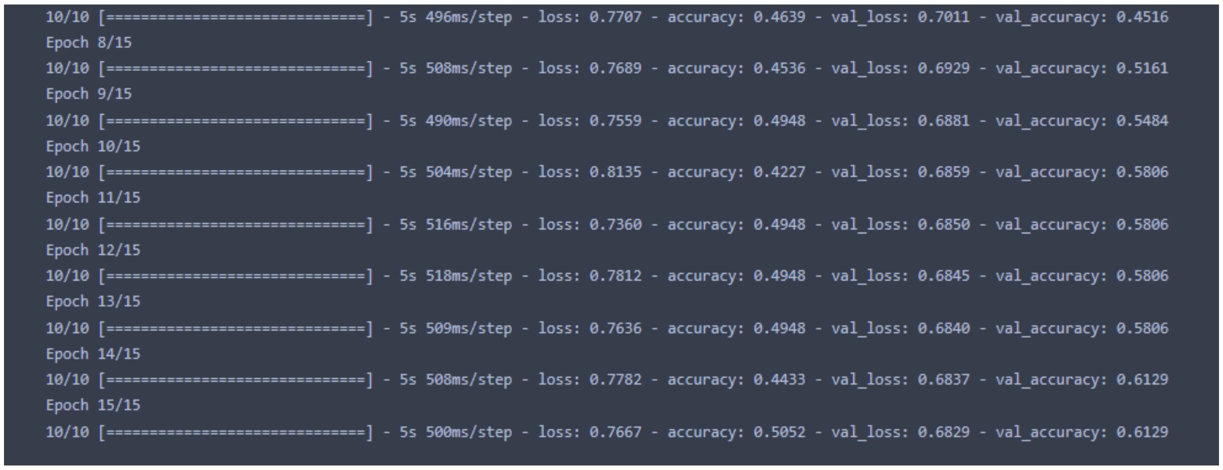
Black Padding Approach

In this section, I converted the images into square shaped ones while preserving the aspect ratio. I created a blank square image and pasted the resized image on it to form a new image. The final images are of size 512×512.

The derived images look like this:

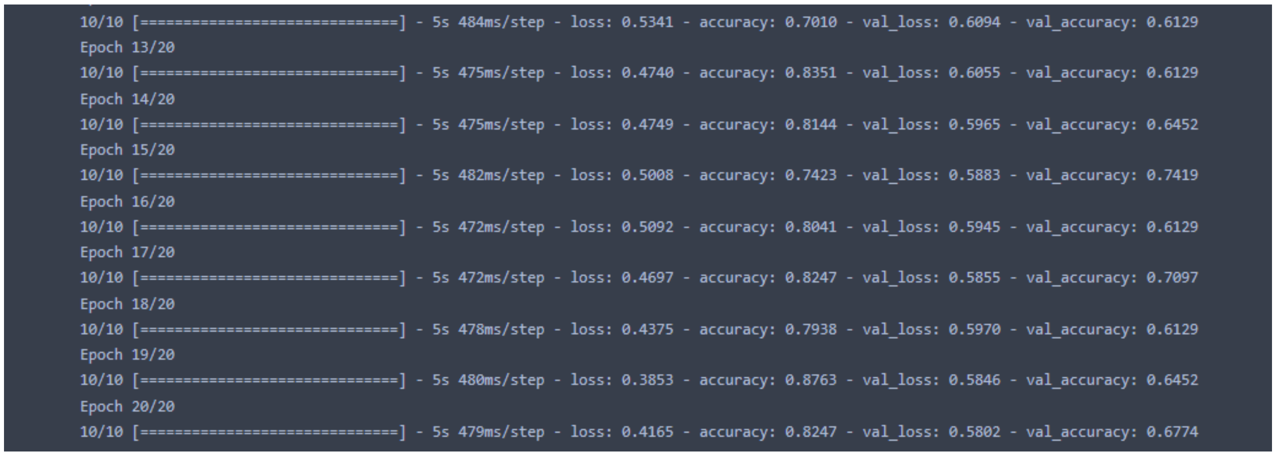


i) After transfer learning (training FC layers only), by the end of the 15th epoch, I got the training accuracy of 50, and validation accuracy of 61.



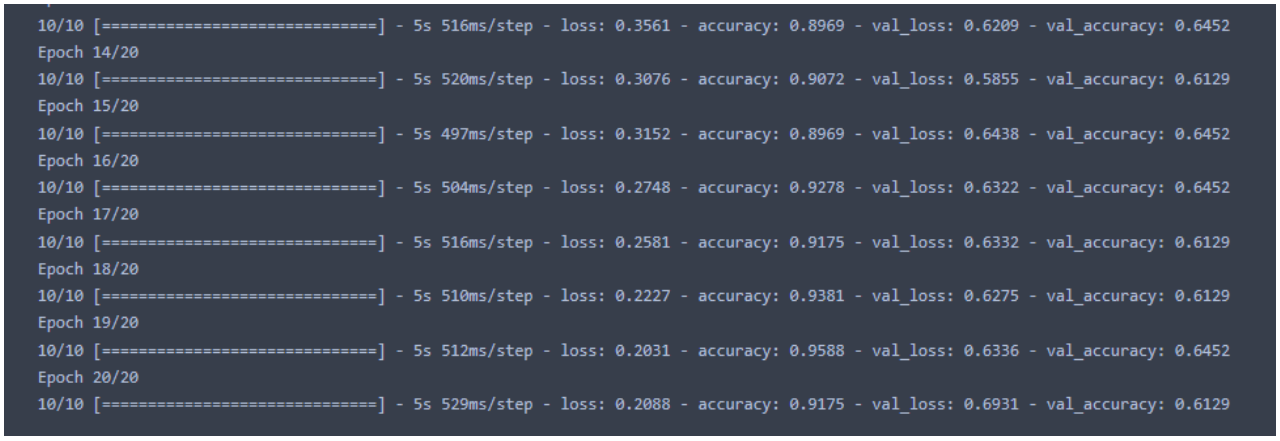
```
10/10 [==============================] - 5s 496ms/step - loss: 0.7707 - accuracy: 0.4639 - val_loss: 0.7011 - val_accuracy: 0.4516
Epoch 8/15
10/10 [==============================] - 5s 508ms/step - loss: 0.7689 - accuracy: 0.4536 - val_loss: 0.6929 - val_accuracy: 0.5161
Epoch 9/15
10/10 [==============================] - 5s 490ms/step - loss: 0.7559 - accuracy: 0.4948 - val_loss: 0.6881 - val_accuracy: 0.5484
Epoch 10/15
10/10 [==============================] - 5s 504ms/step - loss: 0.8135 - accuracy: 0.4227 - val_loss: 0.6859 - val_accuracy: 0.5806
Epoch 11/15
10/10 [==============================] - 5s 516ms/step - loss: 0.7360 - accuracy: 0.4948 - val_loss: 0.6850 - val_accuracy: 0.5806
Epoch 12/15
10/10 [==============================] - 5s 518ms/step - loss: 0.7812 - accuracy: 0.4948 - val_loss: 0.6845 - val_accuracy: 0.5806
Epoch 13/15
10/10 [==============================] - 5s 509ms/step - loss: 0.7636 - accuracy: 0.4948 - val_loss: 0.6840 - val_accuracy: 0.5806
Epoch 14/15
10/10 [==============================] - 5s 508ms/step - loss: 0.7782 - accuracy: 0.4433 - val_loss: 0.6837 - val_accuracy: 0.6129
Epoch 15/15
10/10 [==============================] - 5s 500ms/step - loss: 0.7667 - accuracy: 0.5052 - val_loss: 0.6829 - val_accuracy: 0.6129
```

ii) After fine-tuning conv_block_5 along with the FC layers, by the end of the 20th epoch, I got the training accuracy of 82, and validation accuracy of 67.



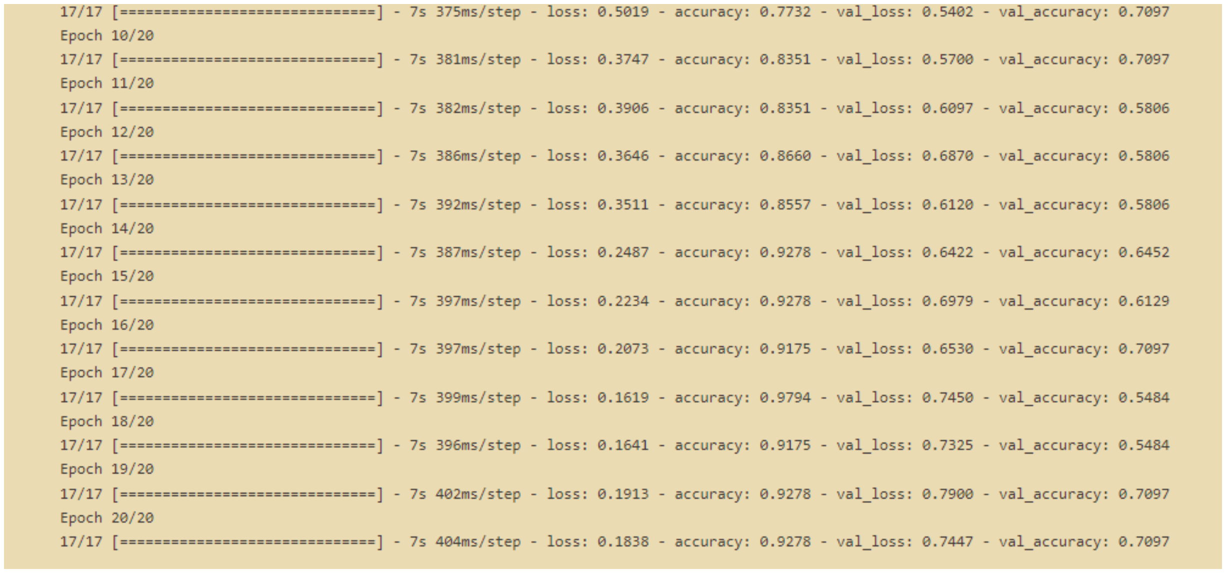
```
10/10 [==============================] - 5s 484ms/step - loss: 0.5341 - accuracy: 0.7010 - val_loss: 0.6094 - val_accuracy: 0.6129
Epoch 13/20
10/10 [==============================] - 5s 475ms/step - loss: 0.4740 - accuracy: 0.8351 - val_loss: 0.6055 - val_accuracy: 0.6129
Epoch 14/20
10/10 [==============================] - 5s 475ms/step - loss: 0.4749 - accuracy: 0.8144 - val_loss: 0.5965 - val_accuracy: 0.6452
Epoch 15/20
10/10 [==============================] - 5s 482ms/step - loss: 0.5008 - accuracy: 0.7423 - val_loss: 0.5883 - val_accuracy: 0.7419
Epoch 16/20
10/10 [==============================] - 5s 472ms/step - loss: 0.5092 - accuracy: 0.8041 - val_loss: 0.5945 - val_accuracy: 0.6129
Epoch 17/20
10/10 [==============================] - 5s 472ms/step - loss: 0.4697 - accuracy: 0.8247 - val_loss: 0.5855 - val_accuracy: 0.7097
Epoch 18/20
10/10 [==============================] - 5s 478ms/step - loss: 0.4375 - accuracy: 0.7938 - val_loss: 0.5970 - val_accuracy: 0.6129
Epoch 19/20
10/10 [==============================] - 5s 480ms/step - loss: 0.3853 - accuracy: 0.8763 - val_loss: 0.5846 - val_accuracy: 0.6452
Epoch 20/20
10/10 [==============================] - 5s 479ms/step - loss: 0.4165 - accuracy: 0.8247 - val_loss: 0.5802 - val_accuracy: 0.6774
```

iii) After fine-tuning conv_block4 and conv_block5 along with the FC layers, by the end of the 20th epoch, I got the training accuracy of 91, and validation accuracy of 61. Training and validation loss reduced to 0.2 and 0.69, respectively.



```
10/10 [==============================] - 5s 516ms/step - loss: 0.3561 - accuracy: 0.8969 - val_loss: 0.6209 - val_accuracy: 0.6452
Epoch 14/20
10/10 [==============================] - 5s 520ms/step - loss: 0.3076 - accuracy: 0.9072 - val_loss: 0.5855 - val_accuracy: 0.6129
Epoch 15/20
10/10 [==============================] - 5s 497ms/step - loss: 0.3152 - accuracy: 0.8969 - val_loss: 0.6438 - val_accuracy: 0.6452
Epoch 16/20
10/10 [==============================] - 5s 504ms/step - loss: 0.2748 - accuracy: 0.9278 - val_loss: 0.6322 - val_accuracy: 0.6452
Epoch 17/20
10/10 [==============================] - 5s 516ms/step - loss: 0.2581 - accuracy: 0.9175 - val_loss: 0.6332 - val_accuracy: 0.6129
Epoch 18/20
10/10 [==============================] - 5s 510ms/step - loss: 0.2227 - accuracy: 0.9381 - val_loss: 0.6275 - val_accuracy: 0.6129
Epoch 19/20
10/10 [==============================] - 5s 512ms/step - loss: 0.2031 - accuracy: 0.9588 - val_loss: 0.6336 - val_accuracy: 0.6452
Epoch 20/20
10/10 [==============================] - 5s 529ms/step - loss: 0.2088 - accuracy: 0.9175 - val_loss: 0.6931 - val_accuracy: 0.6129
```

iv) Unfortunately, my machine was not powerful enough to carry out this experiment and came short on training the entire parameters of the model. So, inconveniently I reduced the batch size from 10 to 6. After training the entire model for 20 epochs, I got the training accuracy of 92, and validation accuracy of almost 70. Training loss is very low but validation loss is about 74, which can be an indication of the model being highly overfitted



```
17/17 [==============================] - 7s 375ms/step - loss: 0.5019 - accuracy: 0.7732 - val_loss: 0.5402 - val_accuracy: 0.7097
Epoch 10/20
17/17 [==============================] - 7s 381ms/step - loss: 0.3747 - accuracy: 0.8351 - val_loss: 0.5700 - val_accuracy: 0.7097
Epoch 11/20
17/17 [==============================] - 7s 382ms/step - loss: 0.3906 - accuracy: 0.8351 - val_loss: 0.6097 - val_accuracy: 0.5806
Epoch 12/20
17/17 [==============================] - 7s 386ms/step - loss: 0.3646 - accuracy: 0.8660 - val_loss: 0.6870 - val_accuracy: 0.5806
Epoch 13/20
17/17 [==============================] - 7s 392ms/step - loss: 0.3511 - accuracy: 0.8557 - val_loss: 0.6120 - val_accuracy: 0.5806
Epoch 14/20
17/17 [==============================] - 7s 387ms/step - loss: 0.2487 - accuracy: 0.9278 - val_loss: 0.6422 - val_accuracy: 0.6452
Epoch 15/20
17/17 [==============================] - 7s 397ms/step - loss: 0.2234 - accuracy: 0.9278 - val_loss: 0.6979 - val_accuracy: 0.6129
Epoch 16/20
17/17 [==============================] - 7s 397ms/step - loss: 0.2073 - accuracy: 0.9175 - val_loss: 0.6530 - val_accuracy: 0.7097
Epoch 17/20
17/17 [==============================] - 7s 399ms/step - loss: 0.1619 - accuracy: 0.9794 - val_loss: 0.7450 - val_accuracy: 0.5484
Epoch 18/20
17/17 [==============================] - 7s 396ms/step - loss: 0.1641 - accuracy: 0.9175 - val_loss: 0.7325 - val_accuracy: 0.5484
Epoch 19/20
17/17 [==============================] - 7s 402ms/step - loss: 0.1913 - accuracy: 0.9278 - val_loss: 0.7900 - val_accuracy: 0.7097
Epoch 20/20
17/17 [==============================] - 7s 404ms/step - loss: 0.1838 - accuracy: 0.9278 - val_loss: 0.7447 - val_accuracy: 0.7097
```

- Conclusion

The black padding approach turned out to be a failure. By padding a big part of the image, the CNN will have to learn that the black part of the image is not relevant and does not help with distinguishing between classes, as there is no correlation between the pixels in the black part and being associated to a given class. The CNN has to learn this by gradient descent, and this might take some epochs and be a waste of computational power.

But this is not the main reason why this approach may not be the best idea as this is not necessarily a hard thing for CNN to learn specially with all the images having these paddings. The main issue is the size of the data set which is exceedingly small considering the number of trainable parameters, which has led to a highly overfitted model.

The results of this approach show that given the large number of parameters to fit, my model requires a larger dataset for learning without overfitting on the training set. Splitting the small dataset into a smaller training set and validation set leads to an even weaker indication of how good the model is at generalization to unseen data. Even with various data augmentations, the performance of the model did not improve. So, I proceeded to expand the size of my dataset.

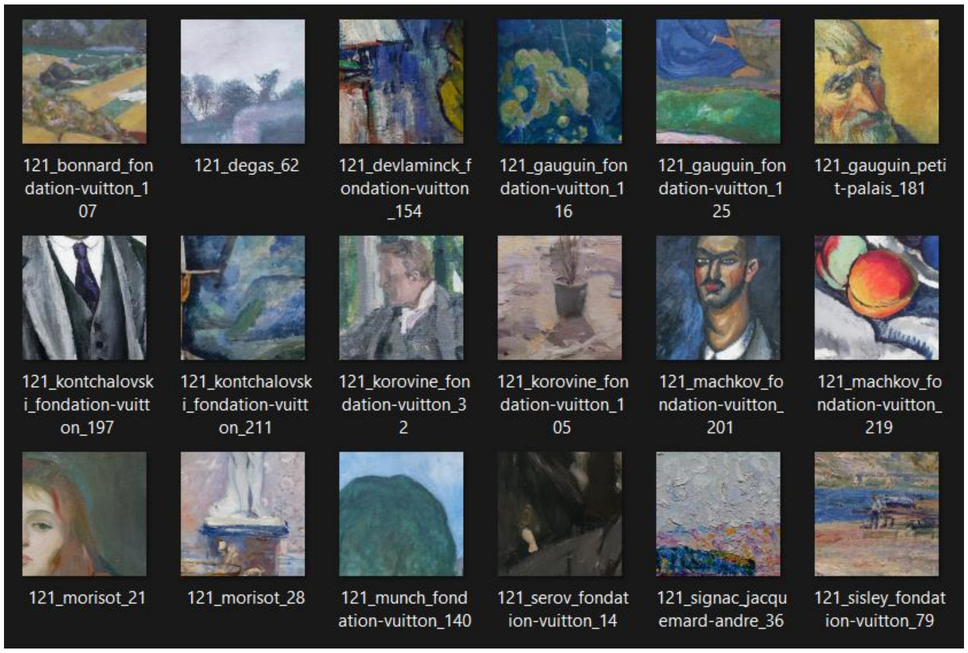
Random Cropping Approach

In this section, I decided to experiment with a theory I read about in a paper (Artist Identification with Convolutional Neural Networks). In that paper the assumption was that there is no need to feed each painting in its entirety as an input to the model and that the style of an artist should be present in each and every crop of painting and not limited to certain areas. So, the model should be able to train itself and guess properly even if the dataset consists of crops of the paintings, because the crops of paintings should contain enough information for the CNN to determine the artist. In fact, this randomness adds variety to the training data and helps avoid overfitting.

For expanding the size of the dataset, I chose and put aside 10 paintings for prediction set and applied random cropping on the rest of the paintings, 57 of which were painted by Cezanne and the remaining 68 paintings belonged to other artists. I tried the three following schemes:

A. Taking 30 random crops of size 512x512 from each painting in the dataset
B. Taking 120 random crops of size 256x256 from each painting in the dataset
C. Taking 8 random crops of size 1024x1024 from each painting in the dataset

The picture below depicts how my dataset looks like now:


121_bonnard_fondation-vuitton_107
121_degas_62
121_devlaminck_fondation-vuitton_154
121_gauguin_fondation-vuitton_116
121_gauguin_fondation-vuitton_125
121_gauguin_petit-palais_181
121_kontchalovski_fondation-vuitton_197
121_kontchalovski_fondation-vuitton_211
121_korovine_fondation-vuitton_32
121_korovine_fondation-vuitton_105
121_machkov_fondation-vuitton_201
121_machkov_fondation-vuitton_219
121_morisot_21
121_morisot_28
121_munch_fondation-vuitton_140
121_serov_fondation-vuitton_14
121_signac_jacquemard-andre_36
121_sisley_fondation-vuitton_79

For testing my model, I took fixed center crops from the paintings in the prediction set, meaning that the final model will make prediction solely based on these center crops. regardless of what the rest of the painting depicts. For each scenario these center crops have the same size as the randomly cropped images.

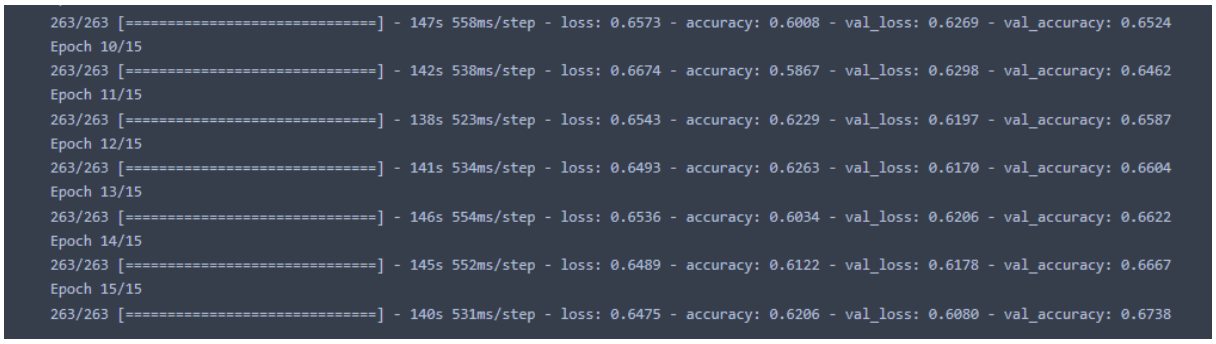
- ## VGG16:

Scenario A

By taking 30 random crops of each painting in the dataset, I ended up with 1710 images belonging to “Cezanne” and 2040 images belonging to the “Others” class, all of which have the size of 512x512. I used 70 precent of the images in each group for training and 30 percent of them for validating my model. Overall, 2625 and 1125 images are used for training and validation set, respectively.

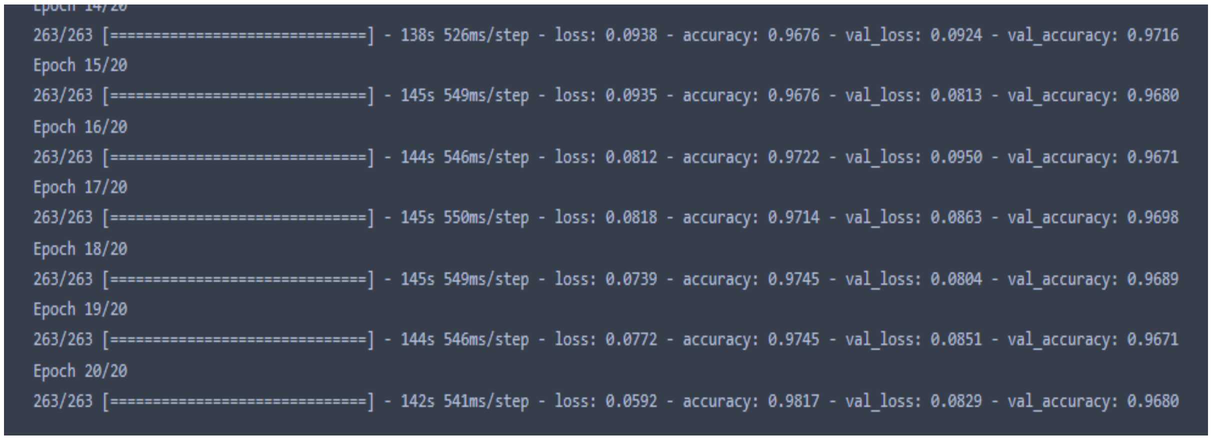
After applying all four fine-tuning scenarios that were discussed earlier, I got the following results:

i) After transfer learning (training FC layers only), by the end of the 15th epoch, I got the training accuracy of 63, and validation accuracy of 67. Loss for both training and validating kept decreasing in general.



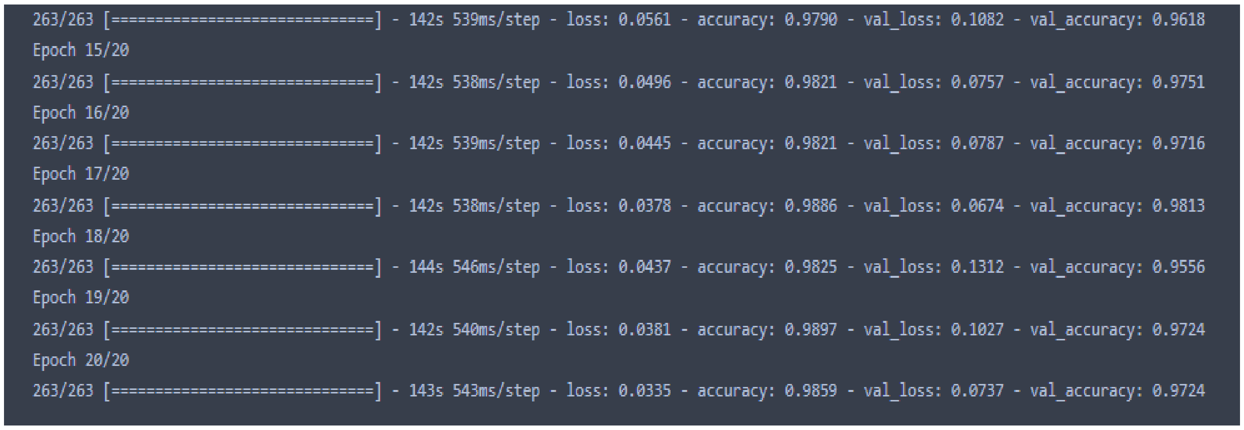
```
263/263 [==============================] - 147s 558ms/step - loss: 0.6573 - accuracy: 0.6008 - val_loss: 0.6269 - val_accuracy: 0.6524
Epoch 10/15
263/263 [==============================] - 142s 538ms/step - loss: 0.6674 - accuracy: 0.5867 - val_loss: 0.6298 - val_accuracy: 0.6462
Epoch 11/15
263/263 [==============================] - 138s 523ms/step - loss: 0.6543 - accuracy: 0.6229 - val_loss: 0.6197 - val_accuracy: 0.6587
Epoch 12/15
263/263 [==============================] - 141s 534ms/step - loss: 0.6493 - accuracy: 0.6263 - val_loss: 0.6170 - val_accuracy: 0.6604
Epoch 13/15
263/263 [==============================] - 146s 554ms/step - loss: 0.6536 - accuracy: 0.6034 - val_loss: 0.6206 - val_accuracy: 0.6622
Epoch 14/15
263/263 [==============================] - 145s 552ms/step - loss: 0.6489 - accuracy: 0.6122 - val_loss: 0.6178 - val_accuracy: 0.6667
Epoch 15/15
263/263 [==============================] - 140s 531ms/step - loss: 0.6475 - accuracy: 0.6206 - val_loss: 0.6080 - val_accuracy: 0.6738
```

ii) After fine-tuning conv_block_5 along with the FC layers, by the end of the 20th epoch, I got the training accuracy of 98, and validation accuracy of 96. Loss for both training and validating kept decreasing in general.



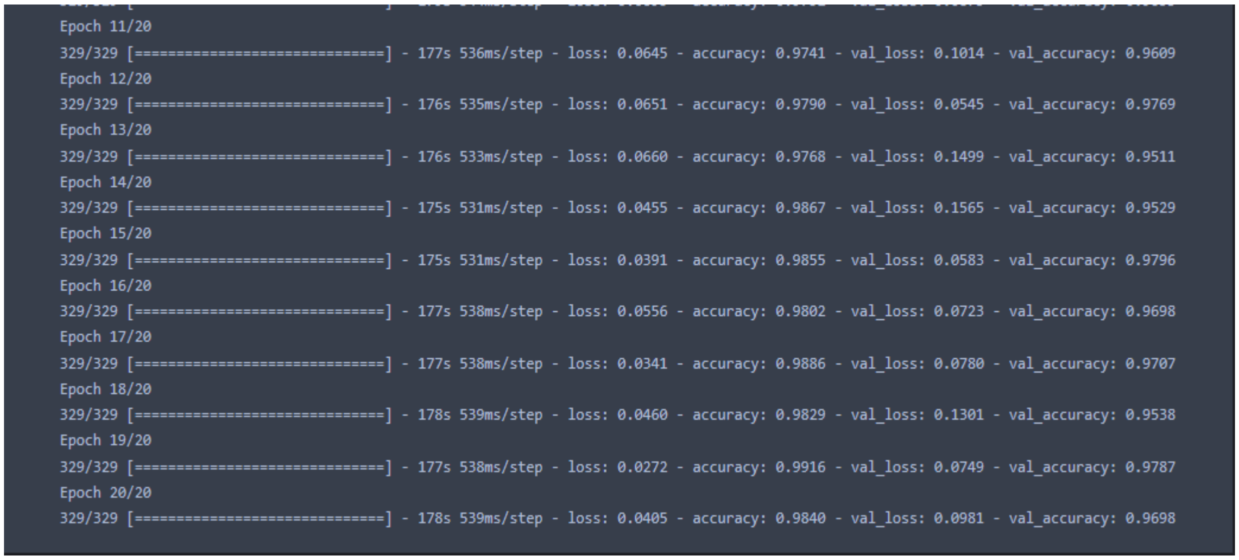
```
Epoch 14/20
263/263 [==============================] - 138s 526ms/step - loss: 0.0938 - accuracy: 0.9676 - val_loss: 0.0924 - val_accuracy: 0.9716
Epoch 15/20
263/263 [==============================] - 145s 549ms/step - loss: 0.0935 - accuracy: 0.9676 - val_loss: 0.0813 - val_accuracy: 0.9680
Epoch 16/20
263/263 [==============================] - 144s 546ms/step - loss: 0.0812 - accuracy: 0.9722 - val_loss: 0.0950 - val_accuracy: 0.9671
Epoch 17/20
263/263 [==============================] - 145s 550ms/step - loss: 0.0818 - accuracy: 0.9714 - val_loss: 0.0863 - val_accuracy: 0.9698
Epoch 18/20
263/263 [==============================] - 145s 549ms/step - loss: 0.0739 - accuracy: 0.9745 - val_loss: 0.0804 - val_accuracy: 0.9689
Epoch 19/20
263/263 [==============================] - 144s 546ms/step - loss: 0.0772 - accuracy: 0.9745 - val_loss: 0.0851 - val_accuracy: 0.9671
Epoch 20/20
263/263 [==============================] - 142s 541ms/step - loss: 0.0592 - accuracy: 0.9817 - val_loss: 0.0829 - val_accuracy: 0.9680
```

iii) After fine-tuning conv_block4 and conv_block5 along with the FC layers, by the end of the 20th epoch, I got the training accuracy of 98, and validation accuracy of 97. Loss for both training and validating kept decreasing in general.



```
263/263 [==============================] - 142s 539ms/step - loss: 0.0561 - accuracy: 0.9790 - val_loss: 0.1082 - val_accuracy: 0.9618
Epoch 15/20
263/263 [==============================] - 142s 538ms/step - loss: 0.0496 - accuracy: 0.9821 - val_loss: 0.0757 - val_accuracy: 0.9751
Epoch 16/20
263/263 [==============================] - 142s 539ms/step - loss: 0.0445 - accuracy: 0.9821 - val_loss: 0.0787 - val_accuracy: 0.9716
Epoch 17/20
263/263 [==============================] - 142s 538ms/step - loss: 0.0378 - accuracy: 0.9886 - val_loss: 0.0674 - val_accuracy: 0.9813
Epoch 18/20
263/263 [==============================] - 144s 546ms/step - loss: 0.0437 - accuracy: 0.9825 - val_loss: 0.1312 - val_accuracy: 0.9556
Epoch 19/20
263/263 [==============================] - 142s 540ms/step - loss: 0.0381 - accuracy: 0.9897 - val_loss: 0.1027 - val_accuracy: 0.9724
Epoch 20/20
263/263 [==============================] - 143s 543ms/step - loss: 0.0335 - accuracy: 0.9859 - val_loss: 0.0737 - val_accuracy: 0.9724
```

iv) Unfortunately, my machine was not powerful enough to carry out this experiment and came short on training the entire parameters of the model. So inconveniently, I reduced the batch size from 10 to 8. After training the entire model for 20 epochs, I got the training accuracy of 98, and validation accuracy of almost 97. Loss for both training and validating kept decreasing in general.



```
Epoch 11/20
329/329 [==============================] - 177s 536ms/step - loss: 0.0645 - accuracy: 0.9741 - val_loss: 0.1014 - val_accuracy: 0.9609
Epoch 12/20
329/329 [==============================] - 176s 535ms/step - loss: 0.0651 - accuracy: 0.9790 - val_loss: 0.0545 - val_accuracy: 0.9769
Epoch 13/20
329/329 [==============================] - 176s 533ms/step - loss: 0.0660 - accuracy: 0.9768 - val_loss: 0.1499 - val_accuracy: 0.9511
Epoch 14/20
329/329 [==============================] - 175s 531ms/step - loss: 0.0455 - accuracy: 0.9867 - val_loss: 0.1565 - val_accuracy: 0.9529
Epoch 15/20
329/329 [==============================] - 175s 531ms/step - loss: 0.0391 - accuracy: 0.9855 - val_loss: 0.0583 - val_accuracy: 0.9796
Epoch 16/20
329/329 [==============================] - 177s 538ms/step - loss: 0.0556 - accuracy: 0.9802 - val_loss: 0.0723 - val_accuracy: 0.9698
Epoch 17/20
329/329 [==============================] - 177s 538ms/step - loss: 0.0341 - accuracy: 0.9886 - val_loss: 0.0780 - val_accuracy: 0.9707
Epoch 18/20
329/329 [==============================] - 178s 539ms/step - loss: 0.0460 - accuracy: 0.9829 - val_loss: 0.1301 - val_accuracy: 0.9538
Epoch 19/20
329/329 [==============================] - 177s 538ms/step - loss: 0.0272 - accuracy: 0.9916 - val_loss: 0.0749 - val_accuracy: 0.9787
Epoch 20/20
329/329 [==============================] - 178s 539ms/step - loss: 0.0405 - accuracy: 0.9840 - val_loss: 0.0981 - val_accuracy: 0.9698
```

➢ Conclusion:

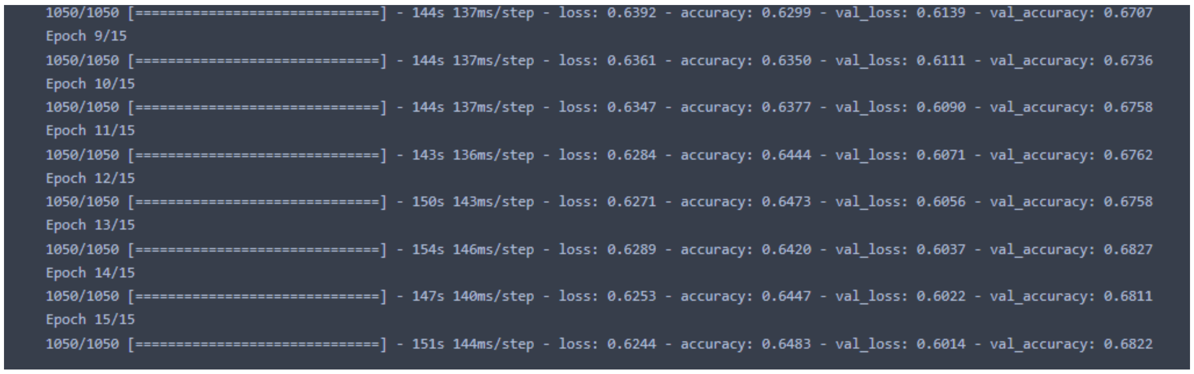
Training the last two convolutional blocks along with the fully connected layers was slightly better than training the entire model with all its parameters.

Scenario B

By taking 120 random crops of each painting in the dataset, I ended up with 6840 sub-images belonging to “Cezanne” and 8160 sub-images belonging to the “Others” class, all of which have the size of 256x256. I used 70 precent of the images in each group for training and 30 percent of them for validating my model. Overall, 10500 and 4500 images are used for training and validation set, respectively.

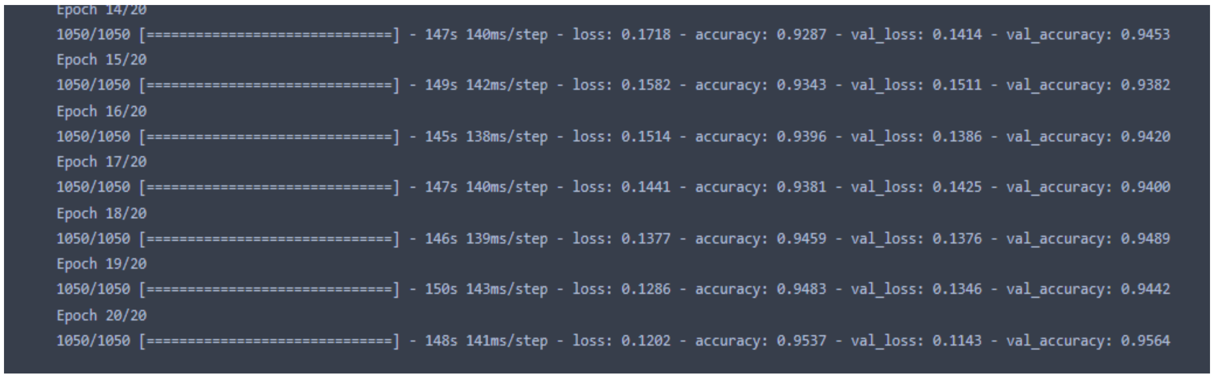
The results of each fine-tuning scenarios are listed below:

i) After transfer learning (training FC layers only), by the end of the 15th epoch, I got the training accuracy of 64, and validation accuracy of 68. Loss for both training and validating kept decreasing in general.



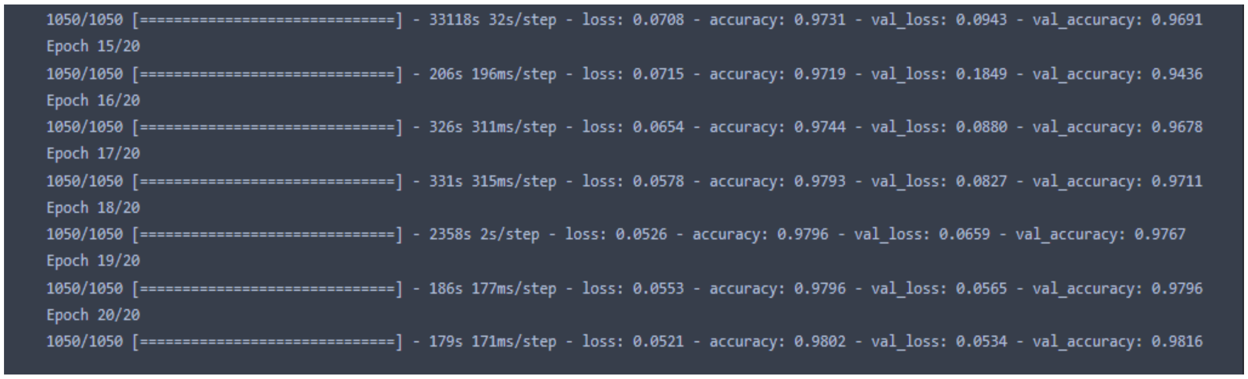
```
1050/1050 [==============================] - 144s 137ms/step - loss: 0.6392 - accuracy: 0.6299 - val_loss: 0.6139 - val_accuracy: 0.6707
Epoch 9/15
1050/1050 [==============================] - 144s 137ms/step - loss: 0.6361 - accuracy: 0.6350 - val_loss: 0.6111 - val_accuracy: 0.6736
Epoch 10/15
1050/1050 [==============================] - 144s 137ms/step - loss: 0.6347 - accuracy: 0.6377 - val_loss: 0.6090 - val_accuracy: 0.6758
Epoch 11/15
1050/1050 [==============================] - 143s 136ms/step - loss: 0.6284 - accuracy: 0.6444 - val_loss: 0.6071 - val_accuracy: 0.6762
Epoch 12/15
1050/1050 [==============================] - 150s 143ms/step - loss: 0.6271 - accuracy: 0.6473 - val_loss: 0.6056 - val_accuracy: 0.6758
Epoch 13/15
1050/1050 [==============================] - 154s 146ms/step - loss: 0.6289 - accuracy: 0.6420 - val_loss: 0.6037 - val_accuracy: 0.6827
Epoch 14/15
1050/1050 [==============================] - 147s 140ms/step - loss: 0.6253 - accuracy: 0.6447 - val_loss: 0.6022 - val_accuracy: 0.6811
Epoch 15/15
1050/1050 [==============================] - 151s 144ms/step - loss: 0.6244 - accuracy: 0.6483 - val_loss: 0.6014 - val_accuracy: 0.6822
```

ii) After fine-tuning conv_block_5 along with the FC layers, by the end of the 20th epoch, the accuracy rate reached 95 for both validation set and training set. Loss for both training and validating kept decreasing in general.



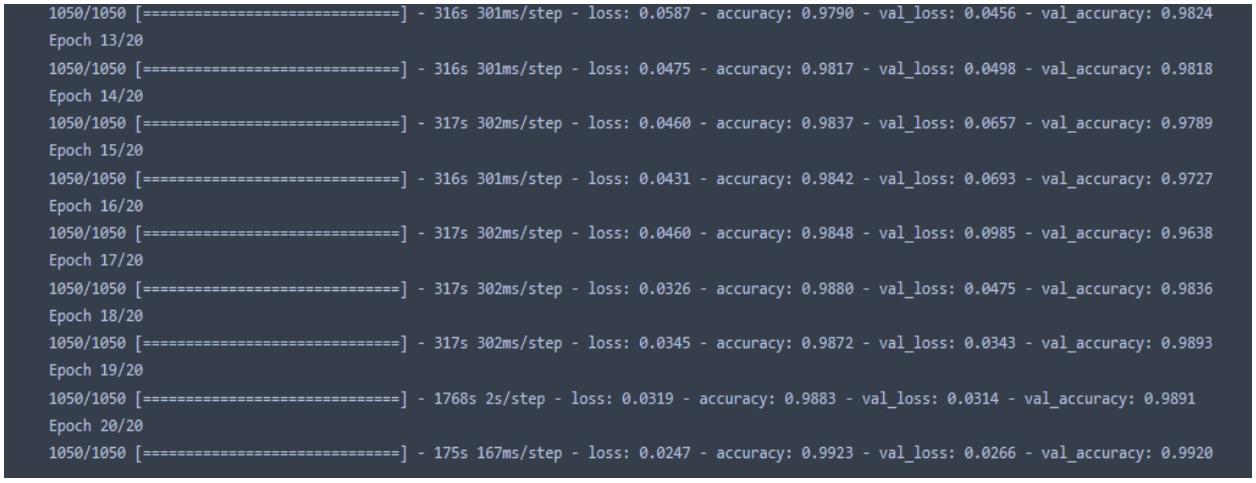
```
Epoch 14/20
1050/1050 [==============================] - 147s 140ms/step - loss: 0.1718 - accuracy: 0.9287 - val_loss: 0.1414 - val_accuracy: 0.9453
Epoch 15/20
1050/1050 [==============================] - 149s 142ms/step - loss: 0.1582 - accuracy: 0.9343 - val_loss: 0.1511 - val_accuracy: 0.9382
Epoch 16/20
1050/1050 [==============================] - 145s 138ms/step - loss: 0.1514 - accuracy: 0.9396 - val_loss: 0.1386 - val_accuracy: 0.9420
Epoch 17/20
1050/1050 [==============================] - 147s 140ms/step - loss: 0.1441 - accuracy: 0.9381 - val_loss: 0.1425 - val_accuracy: 0.9400
Epoch 18/20
1050/1050 [==============================] - 146s 139ms/step - loss: 0.1377 - accuracy: 0.9459 - val_loss: 0.1376 - val_accuracy: 0.9489
Epoch 19/20
1050/1050 [==============================] - 150s 143ms/step - loss: 0.1286 - accuracy: 0.9483 - val_loss: 0.1346 - val_accuracy: 0.9442
Epoch 20/20
1050/1050 [==============================] - 148s 141ms/step - loss: 0.1202 - accuracy: 0.9537 - val_loss: 0.1143 - val_accuracy: 0.9564
```

iii) After fine-tuning conv_block4 and conv_block5 along with the FC layers, by the end of the 20th epoch, the accuracy rate reached 98 for both validation set and training set. Loss for both training and validating kept decreasing in general.



```
1050/1050 [==============================] - 33118s 32s/step - loss: 0.0708 - accuracy: 0.9731 - val_loss: 0.0943 - val_accuracy: 0.9691
Epoch 15/20
1050/1050 [==============================] - 206s 196ms/step - loss: 0.0715 - accuracy: 0.9719 - val_loss: 0.1849 - val_accuracy: 0.9436
Epoch 16/20
1050/1050 [==============================] - 326s 311ms/step - loss: 0.0654 - accuracy: 0.9744 - val_loss: 0.0880 - val_accuracy: 0.9678
Epoch 17/20
1050/1050 [==============================] - 331s 315ms/step - loss: 0.0578 - accuracy: 0.9793 - val_loss: 0.0827 - val_accuracy: 0.9711
Epoch 18/20
1050/1050 [==============================] - 2358s 2s/step - loss: 0.0526 - accuracy: 0.9796 - val_loss: 0.0659 - val_accuracy: 0.9767
Epoch 19/20
1050/1050 [==============================] - 186s 177ms/step - loss: 0.0553 - accuracy: 0.9796 - val_loss: 0.0565 - val_accuracy: 0.9796
Epoch 20/20
1050/1050 [==============================] - 179s 171ms/step - loss: 0.0521 - accuracy: 0.9802 - val_loss: 0.0534 - val_accuracy: 0.9816
```

iv) After training the entire model for 20 epochs, the accuracy rate reached **99** for both validation set and training set which is the **best result** I have got so far. Loss for both training and validating kept decreasing in general.



```
1050/1050 [==============================] - 316s 301ms/step - loss: 0.0587 - accuracy: 0.9790 - val_loss: 0.0456 - val_accuracy: 0.9824
Epoch 13/20
1050/1050 [==============================] - 316s 301ms/step - loss: 0.0475 - accuracy: 0.9817 - val_loss: 0.0498 - val_accuracy: 0.9818
Epoch 14/20
1050/1050 [==============================] - 317s 302ms/step - loss: 0.0460 - accuracy: 0.9837 - val_loss: 0.0657 - val_accuracy: 0.9789
Epoch 15/20
1050/1050 [==============================] - 316s 301ms/step - loss: 0.0431 - accuracy: 0.9842 - val_loss: 0.0693 - val_accuracy: 0.9727
Epoch 16/20
1050/1050 [==============================] - 317s 302ms/step - loss: 0.0460 - accuracy: 0.9848 - val_loss: 0.0985 - val_accuracy: 0.9638
Epoch 17/20
1050/1050 [==============================] - 317s 302ms/step - loss: 0.0326 - accuracy: 0.9880 - val_loss: 0.0475 - val_accuracy: 0.9836
Epoch 18/20
1050/1050 [==============================] - 317s 302ms/step - loss: 0.0345 - accuracy: 0.9872 - val_loss: 0.0343 - val_accuracy: 0.9893
Epoch 19/20
1050/1050 [==============================] - 1768s 2s/step - loss: 0.0319 - accuracy: 0.9883 - val_loss: 0.0314 - val_accuracy: 0.9891
Epoch 20/20
1050/1050 [==============================] - 175s 167ms/step - loss: 0.0247 - accuracy: 0.9923 - val_loss: 0.0266 - val_accuracy: 0.9920
```

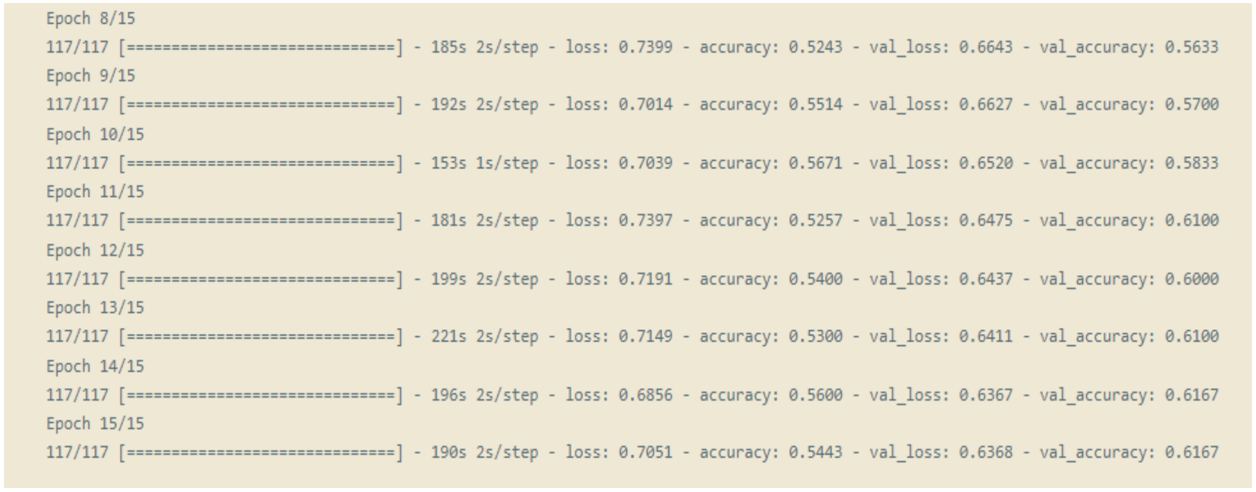
- Conclusion:

So far, this model has led to the best results, proving that no matter how small the crops are, they contain sufficient data for the model to extract features and patterns and adjust its weights with. The more layers I trained the better the results got, the final and the best result was obtained after training the entire model which gave a validation accuracy of 99 and a loss of 0.02.

Scenario C

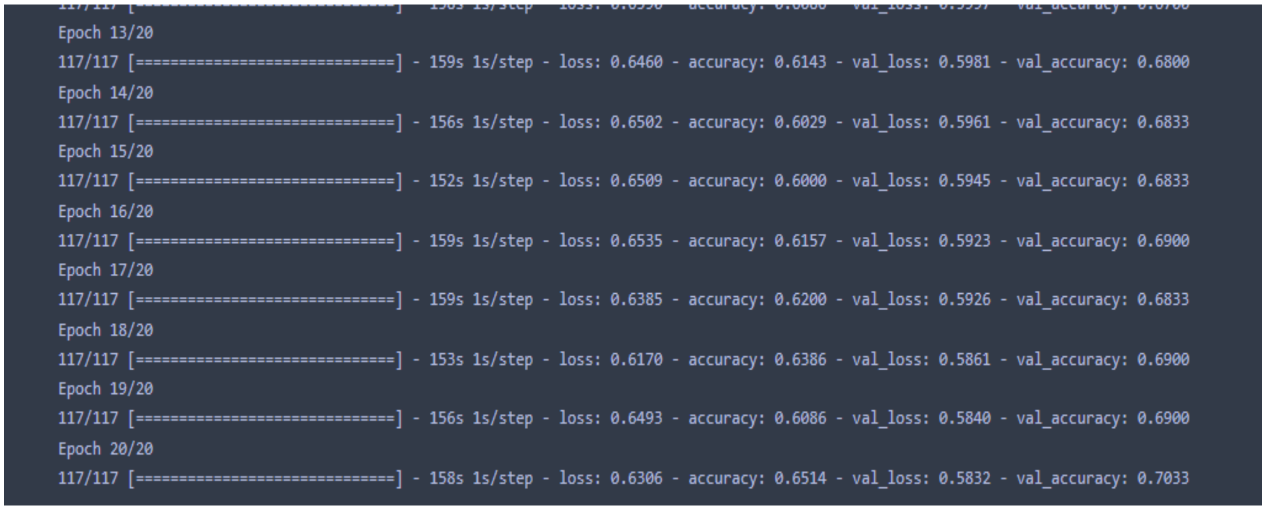
By taking 8 random crops of each painting in the dataset, I ended up with 456 sub-images belonging to "Cezanne" and 544 sub-images belonging to the "Others" class, all of which have the size of 1024x1024. I used 70 precent of the images in each group for training and 30 percent of them for validating my model. Overall, 700 and 300 images are used for training and validation set, respectively.

i) After transfer learning (training FC layers only), by the end of the 15th epoch, I got the training accuracy of 54, and validation accuracy of 61. Loss for both training and validating were fluctuating a lot but decreasing overall.



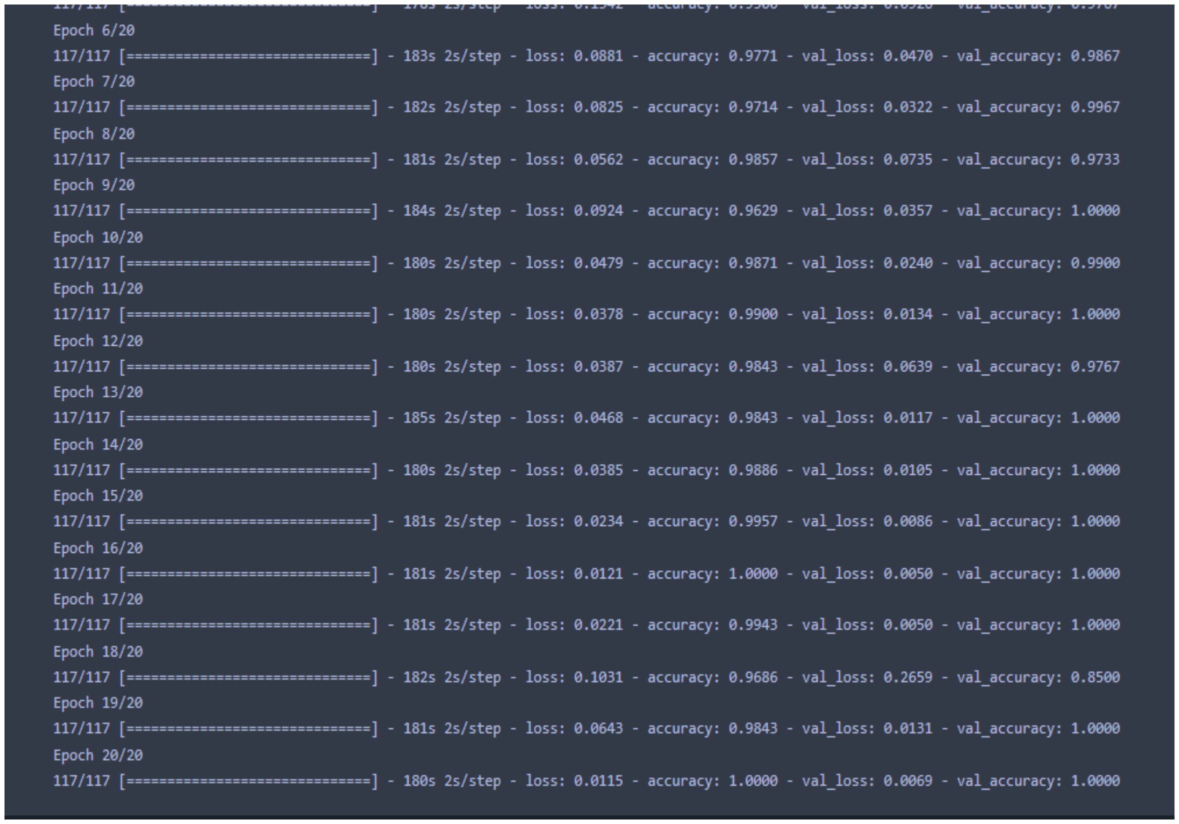
```
Epoch 8/15
117/117 [==============================] - 185s 2s/step - loss: 0.7399 - accuracy: 0.5243 - val_loss: 0.6643 - val_accuracy: 0.5633
Epoch 9/15
117/117 [==============================] - 192s 2s/step - loss: 0.7014 - accuracy: 0.5514 - val_loss: 0.6627 - val_accuracy: 0.5700
Epoch 10/15
117/117 [==============================] - 153s 1s/step - loss: 0.7039 - accuracy: 0.5671 - val_loss: 0.6520 - val_accuracy: 0.5833
Epoch 11/15
117/117 [==============================] - 181s 2s/step - loss: 0.7397 - accuracy: 0.5257 - val_loss: 0.6475 - val_accuracy: 0.6100
Epoch 12/15
117/117 [==============================] - 199s 2s/step - loss: 0.7191 - accuracy: 0.5400 - val_loss: 0.6437 - val_accuracy: 0.6000
Epoch 13/15
117/117 [==============================] - 221s 2s/step - loss: 0.7149 - accuracy: 0.5300 - val_loss: 0.6411 - val_accuracy: 0.6100
Epoch 14/15
117/117 [==============================] - 196s 2s/step - loss: 0.6856 - accuracy: 0.5600 - val_loss: 0.6367 - val_accuracy: 0.6167
Epoch 15/15
117/117 [==============================] - 190s 2s/step - loss: 0.7051 - accuracy: 0.5443 - val_loss: 0.6368 - val_accuracy: 0.6167
```

ii) Unfortunately, my machine was not powerful enough to carry out this experiment and came short on training the entire parameters of the model. So inconveniently, I reduced the batch size from 10 to 6 for this case and the following iii and iv. After training the last convolutional block(conv_block_5) for 20 epochs, for the training and validation accuracy I achieved the accuracy rate of 62 and 68, respectively. The change in loss values were not very notable, neither for training loss nor the validation loss.



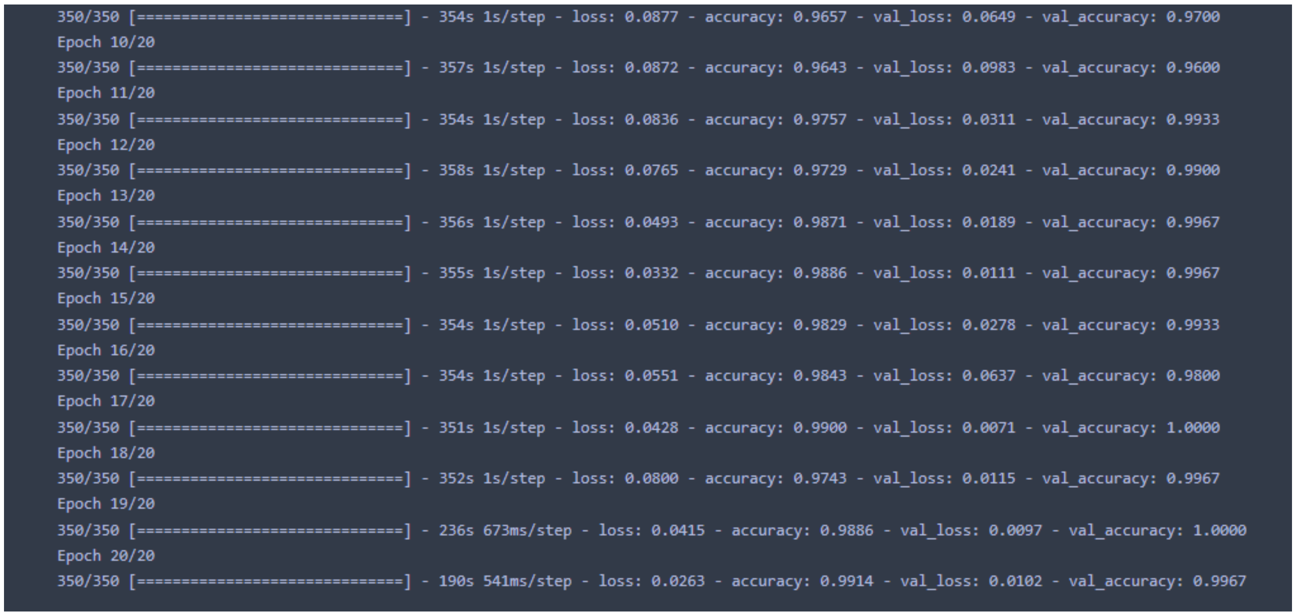
```
Epoch 13/20
117/117 [==============================] - 159s 1s/step - loss: 0.6460 - accuracy: 0.6143 - val_loss: 0.5981 - val_accuracy: 0.6800
Epoch 14/20
117/117 [==============================] - 156s 1s/step - loss: 0.6502 - accuracy: 0.6029 - val_loss: 0.5961 - val_accuracy: 0.6833
Epoch 15/20
117/117 [==============================] - 152s 1s/step - loss: 0.6509 - accuracy: 0.6000 - val_loss: 0.5945 - val_accuracy: 0.6833
Epoch 16/20
117/117 [==============================] - 159s 1s/step - loss: 0.6535 - accuracy: 0.6157 - val_loss: 0.5923 - val_accuracy: 0.6900
Epoch 17/20
117/117 [==============================] - 159s 1s/step - loss: 0.6385 - accuracy: 0.6200 - val_loss: 0.5926 - val_accuracy: 0.6833
Epoch 18/20
117/117 [==============================] - 153s 1s/step - loss: 0.6170 - accuracy: 0.6386 - val_loss: 0.5861 - val_accuracy: 0.6900
Epoch 19/20
117/117 [==============================] - 156s 1s/step - loss: 0.6493 - accuracy: 0.6086 - val_loss: 0.5840 - val_accuracy: 0.6900
Epoch 20/20
117/117 [==============================] - 158s 1s/step - loss: 0.6306 - accuracy: 0.6514 - val_loss: 0.5832 - val_accuracy: 0.7033
```

iii) After training the last two convolutional blocks (conv_block_5 and conv_block_4) for 20 epochs, for the training and validation accuracy I achieved the accuracy rate of 100. Loss for both training and validating kept decreasing in general. There was significant fluctuation in 18th epoch.



```
Epoch 6/20
117/117 [==============================] - 183s 2s/step - loss: 0.0881 - accuracy: 0.9771 - val_loss: 0.0470 - val_accuracy: 0.9867
Epoch 7/20
117/117 [==============================] - 182s 2s/step - loss: 0.0825 - accuracy: 0.9714 - val_loss: 0.0322 - val_accuracy: 0.9967
Epoch 8/20
117/117 [==============================] - 181s 2s/step - loss: 0.0562 - accuracy: 0.9857 - val_loss: 0.0735 - val_accuracy: 0.9733
Epoch 9/20
117/117 [==============================] - 184s 2s/step - loss: 0.0924 - accuracy: 0.9629 - val_loss: 0.0357 - val_accuracy: 1.0000
Epoch 10/20
117/117 [==============================] - 180s 2s/step - loss: 0.0479 - accuracy: 0.9871 - val_loss: 0.0240 - val_accuracy: 0.9900
Epoch 11/20
117/117 [==============================] - 180s 2s/step - loss: 0.0378 - accuracy: 0.9900 - val_loss: 0.0134 - val_accuracy: 1.0000
Epoch 12/20
117/117 [==============================] - 180s 2s/step - loss: 0.0387 - accuracy: 0.9843 - val_loss: 0.0639 - val_accuracy: 0.9767
Epoch 13/20
117/117 [==============================] - 185s 2s/step - loss: 0.0468 - accuracy: 0.9843 - val_loss: 0.0117 - val_accuracy: 1.0000
Epoch 14/20
117/117 [==============================] - 180s 2s/step - loss: 0.0385 - accuracy: 0.9886 - val_loss: 0.0105 - val_accuracy: 1.0000
Epoch 15/20
117/117 [==============================] - 181s 2s/step - loss: 0.0234 - accuracy: 0.9957 - val_loss: 0.0086 - val_accuracy: 1.0000
Epoch 16/20
117/117 [==============================] - 181s 2s/step - loss: 0.0121 - accuracy: 1.0000 - val_loss: 0.0050 - val_accuracy: 1.0000
Epoch 17/20
117/117 [==============================] - 181s 2s/step - loss: 0.0221 - accuracy: 0.9943 - val_loss: 0.0050 - val_accuracy: 1.0000
Epoch 18/20
117/117 [==============================] - 182s 2s/step - loss: 0.1031 - accuracy: 0.9686 - val_loss: 0.2659 - val_accuracy: 0.8500
Epoch 19/20
117/117 [==============================] - 181s 2s/step - loss: 0.0643 - accuracy: 0.9843 - val_loss: 0.0131 - val_accuracy: 1.0000
Epoch 20/20
117/117 [==============================] - 180s 2s/step - loss: 0.0115 - accuracy: 1.0000 - val_loss: 0.0069 - val_accuracy: 1.0000
```

iv) For this case I had to choose an even lower batch size because my system came short power-wise. Even though theory says that the bigger the batch size, the lesser is the noise in the gradients and so better is the gradient estimate, I had to choose 2 as my batch size as a tradeoff with my system's memory and forfeit a more efficient step towards minima. By the end of the 11th epoch, I achieved the accuracy of about 100 (there were some slight fluctuations from here on). The final training loss and validation loss are 0.02 and 0.01 respectively. So far this CNN with these crop sizes has proved to perform the best.



```
350/350 [==============================] - 354s 1s/step - loss: 0.0877 - accuracy: 0.9657 - val_loss: 0.0649 - val_accuracy: 0.9700
Epoch 10/20
350/350 [==============================] - 357s 1s/step - loss: 0.0872 - accuracy: 0.9643 - val_loss: 0.0983 - val_accuracy: 0.9600
Epoch 11/20
350/350 [==============================] - 354s 1s/step - loss: 0.0836 - accuracy: 0.9757 - val_loss: 0.0311 - val_accuracy: 0.9933
Epoch 12/20
350/350 [==============================] - 358s 1s/step - loss: 0.0765 - accuracy: 0.9729 - val_loss: 0.0241 - val_accuracy: 0.9900
Epoch 13/20
350/350 [==============================] - 356s 1s/step - loss: 0.0493 - accuracy: 0.9871 - val_loss: 0.0189 - val_accuracy: 0.9967
Epoch 14/20
350/350 [==============================] - 355s 1s/step - loss: 0.0332 - accuracy: 0.9886 - val_loss: 0.0111 - val_accuracy: 0.9967
Epoch 15/20
350/350 [==============================] - 354s 1s/step - loss: 0.0510 - accuracy: 0.9829 - val_loss: 0.0278 - val_accuracy: 0.9933
Epoch 16/20
350/350 [==============================] - 354s 1s/step - loss: 0.0551 - accuracy: 0.9843 - val_loss: 0.0637 - val_accuracy: 0.9800
Epoch 17/20
350/350 [==============================] - 351s 1s/step - loss: 0.0428 - accuracy: 0.9900 - val_loss: 0.0071 - val_accuracy: 1.0000
Epoch 18/20
350/350 [==============================] - 352s 1s/step - loss: 0.0800 - accuracy: 0.9743 - val_loss: 0.0115 - val_accuracy: 0.9967
Epoch 19/20
350/350 [==============================] - 236s 673ms/step - loss: 0.0415 - accuracy: 0.9886 - val_loss: 0.0097 - val_accuracy: 1.0000
Epoch 20/20
350/350 [==============================] - 190s 541ms/step - loss: 0.0263 - accuracy: 0.9914 - val_loss: 0.0102 - val_accuracy: 0.9967
```

➢ Conclusion:

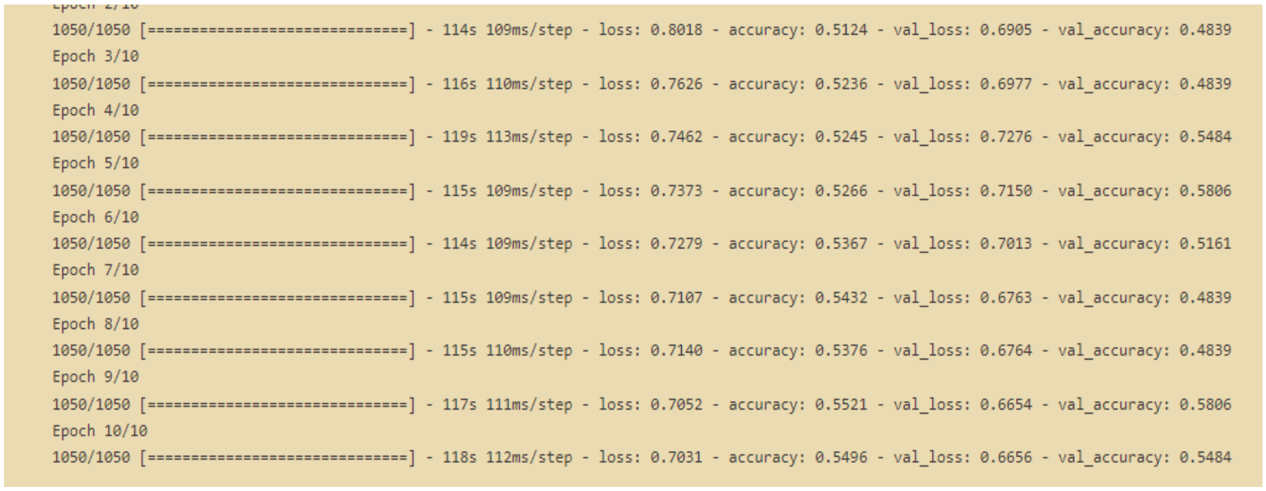
This model led to an impressive validation accuracy of about 100 and loss of 0.01. However, as we will see in the evaluation section, when it came to predicting the unseen data, it performed much less impressive than the previous model.

- ## ResNet50:

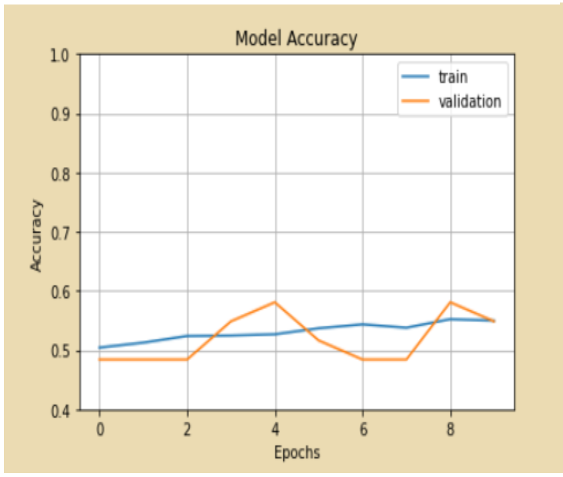
For ResNet I only experimented with random crops of 256 and trained the last convolutional layer along with the fully connected layers because unfortunately my system came short when it came to training with crops of bigger sizes.

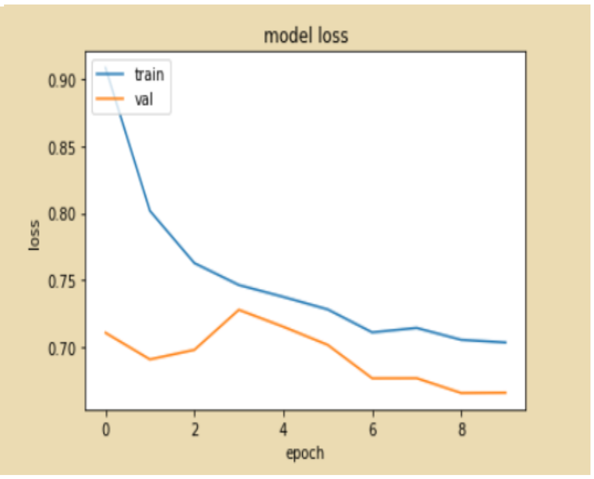
As mentioned before, by taking 120 random crops of each painting in the dataset, I ended up with 6840 sub-images belonging to “Cezanne” and 8160 sub-images belonging to the “Others” class, all of which have the size of 256x256. I used 70 precent of the images in each group for training and 30 percent of them for validating my model. Overall, 10500 and 4500 images are used for training and validation set, respectively.

The results after training the FC layers only are given below (524,801 parameters with the batch size of 10 and 10 epochs):



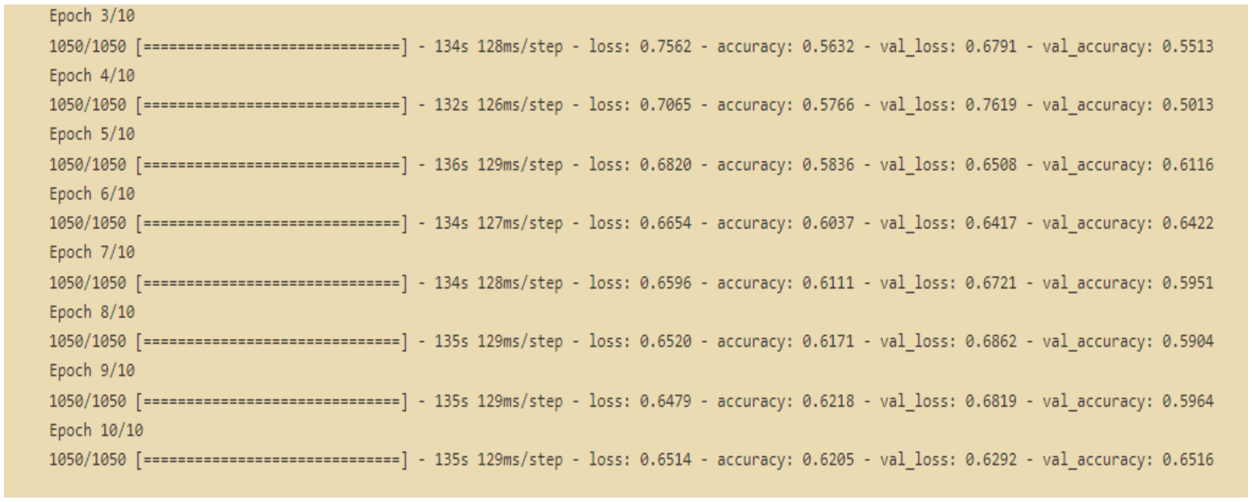
```
Epoch 2/10
1050/1050 [==============================] - 114s 109ms/step - loss: 0.8018 - accuracy: 0.5124 - val_loss: 0.6905 - val_accuracy: 0.4839
Epoch 3/10
1050/1050 [==============================] - 116s 110ms/step - loss: 0.7626 - accuracy: 0.5236 - val_loss: 0.6977 - val_accuracy: 0.4839
Epoch 4/10
1050/1050 [==============================] - 119s 113ms/step - loss: 0.7462 - accuracy: 0.5245 - val_loss: 0.7276 - val_accuracy: 0.5484
Epoch 5/10
1050/1050 [==============================] - 115s 109ms/step - loss: 0.7373 - accuracy: 0.5266 - val_loss: 0.7150 - val_accuracy: 0.5806
Epoch 6/10
1050/1050 [==============================] - 114s 109ms/step - loss: 0.7279 - accuracy: 0.5367 - val_loss: 0.7013 - val_accuracy: 0.5161
Epoch 7/10
1050/1050 [==============================] - 115s 109ms/step - loss: 0.7107 - accuracy: 0.5432 - val_loss: 0.6763 - val_accuracy: 0.4839
Epoch 8/10
1050/1050 [==============================] - 115s 110ms/step - loss: 0.7140 - accuracy: 0.5376 - val_loss: 0.6764 - val_accuracy: 0.4839
Epoch 9/10
1050/1050 [==============================] - 117s 111ms/step - loss: 0.7052 - accuracy: 0.5521 - val_loss: 0.6654 - val_accuracy: 0.5806
Epoch 10/10
1050/1050 [==============================] - 118s 112ms/step - loss: 0.7031 - accuracy: 0.5496 - val_loss: 0.6656 - val_accuracy: 0.5484
```



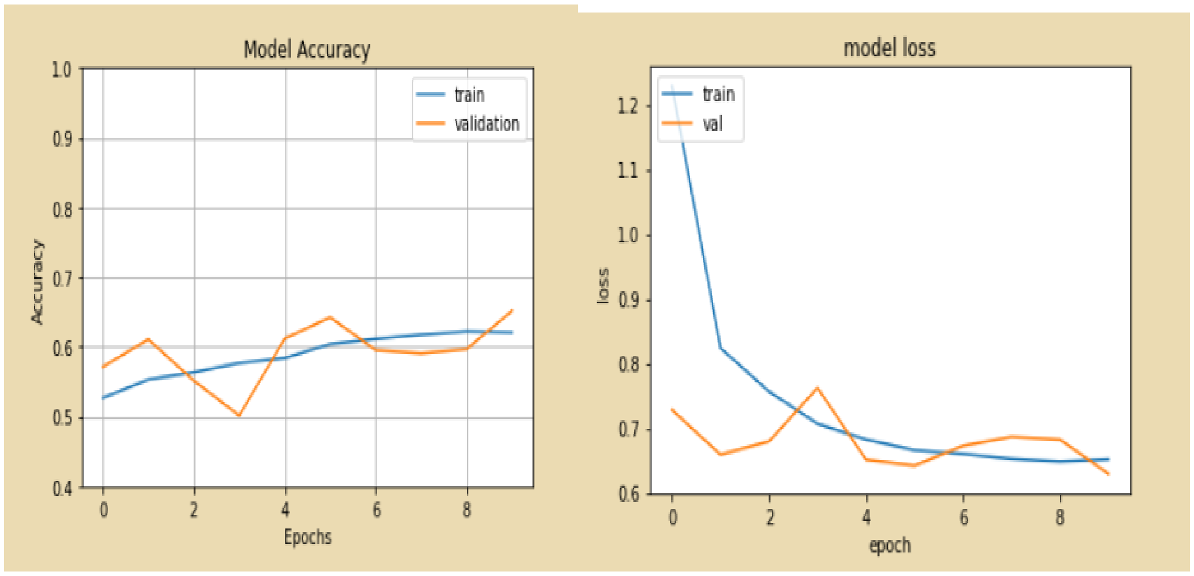


Here, I unfreeze the last convolutional block and start the fine-tuning process. Basically, I have 15,500,901 trainable parameters now.

The results of training the convolutional block 5 and the fully connected layers is given below:



```
Epoch 3/10
1050/1050 [==============================] - 134s 128ms/step - loss: 0.7562 - accuracy: 0.5632 - val_loss: 0.6791 - val_accuracy: 0.5513
Epoch 4/10
1050/1050 [==============================] - 132s 126ms/step - loss: 0.7065 - accuracy: 0.5766 - val_loss: 0.7619 - val_accuracy: 0.5013
Epoch 5/10
1050/1050 [==============================] - 136s 129ms/step - loss: 0.6820 - accuracy: 0.5836 - val_loss: 0.6508 - val_accuracy: 0.6116
Epoch 6/10
1050/1050 [==============================] - 134s 127ms/step - loss: 0.6654 - accuracy: 0.6037 - val_loss: 0.6417 - val_accuracy: 0.6422
Epoch 7/10
1050/1050 [==============================] - 134s 128ms/step - loss: 0.6596 - accuracy: 0.6111 - val_loss: 0.6721 - val_accuracy: 0.5951
Epoch 8/10
1050/1050 [==============================] - 135s 129ms/step - loss: 0.6520 - accuracy: 0.6171 - val_loss: 0.6862 - val_accuracy: 0.5904
Epoch 9/10
1050/1050 [==============================] - 135s 129ms/step - loss: 0.6479 - accuracy: 0.6218 - val_loss: 0.6819 - val_accuracy: 0.5964
Epoch 10/10
1050/1050 [==============================] - 135s 129ms/step - loss: 0.6514 - accuracy: 0.6205 - val_loss: 0.6292 - val_accuracy: 0.6516
```



- Conclusion:

The results were not what I expected them to be. I was under the assumption that since ResNet is much deeper than VGG16 and has significantly more parameters, it would definitely outperform VGG16. However, after only training VGG16's last convolutional blocks I had achieved much better results than ResNet50.

Evalution and Comparison

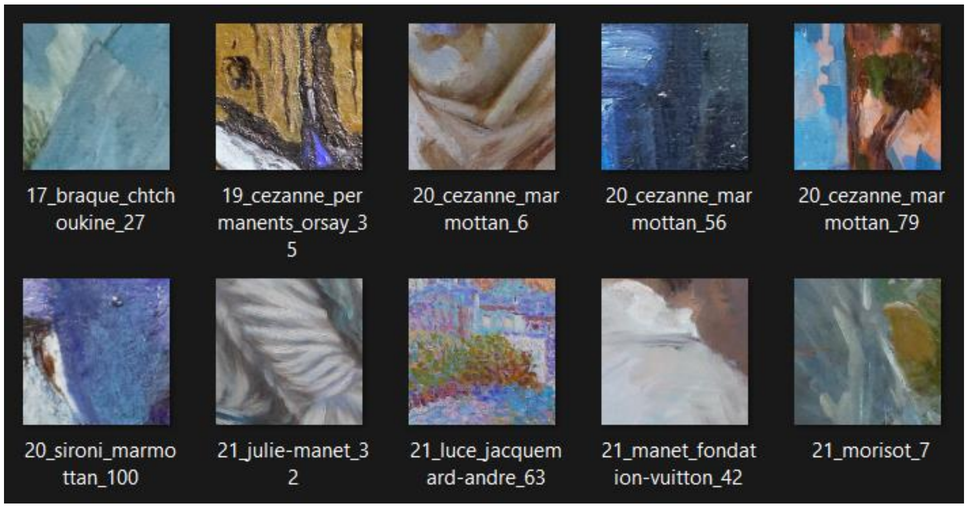
In this part, I visualize the results and evaluate the models that performed the best in each case:

- VGG16 for which I took 120 crops of size 256 from each painting and trained the **entire** model on them.

- VGG16 for which I took 8 crops of size 1024 from each painting and trained the **convolutional blocks 4th and 5th** along with the **FC layers** on them.

- VGG16 for which I resized the original dataset to 512x512, added black padding to keep the aspect ratio intact trained the **entire** model on them.

I test these models on the painting in my predict set. As mentioned earlier, I have taken fixed center crops of these painting. The following figure lists the painting on which I want to test these models.



I will summarize how each of them performed on the testing data by creating a Confusion Matrix for each method and assessing the precision, recall and F1-score for them. Precision and recall are defined as:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

The F1 score, a weighted average of precision and recall, is defined as:

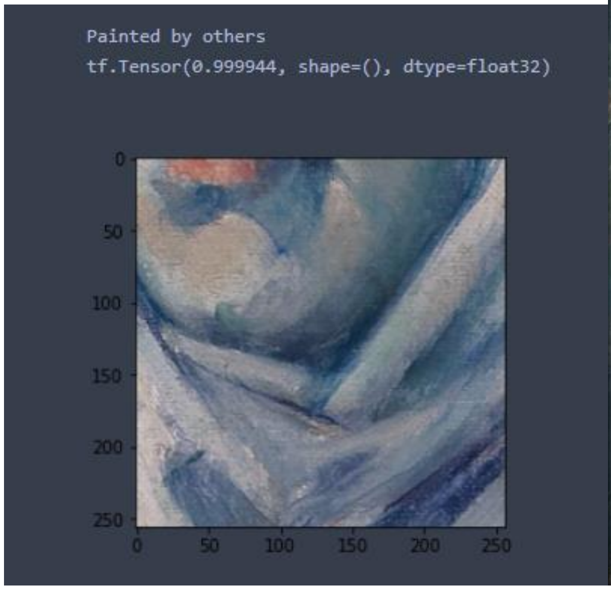
$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

Confusion Matrix:

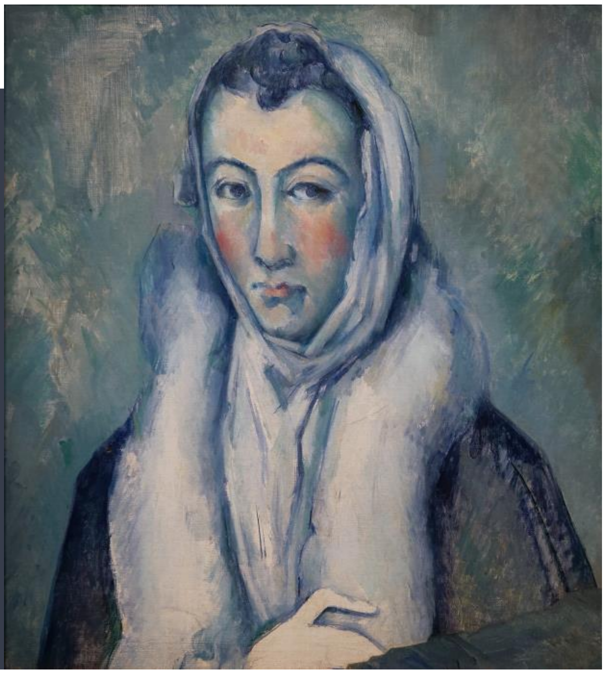
The rows in the Confusion Matrix correspond to what the model has predicted and the columns correspond to the known truth. Since there are only two categories to choose from – Cezanne or Others- then the top left corner contains True Positives. These are paintings that are painted by Cezanne and that were correctly identified by the model. The True Negatives are in the bottom right-corner. These are paintings that are not painted by Cezanne and that were correctly identified by the model. The bottom left corner contains the False Negatives. False Negatives are paintings that are painted by Others but the model predicted that they were not. Lastly, the top right corner contains the False Positives. False positives are paintings that were drawn by Cezanne but the models says that they were not, in other words, they were misclassified. The numbers along the diagonal tell how many times the samples were correctly classified. The numbers not on the diagonal are samples on which the model guessed wrong.

Results of VGG16 Random Crops 120-256:

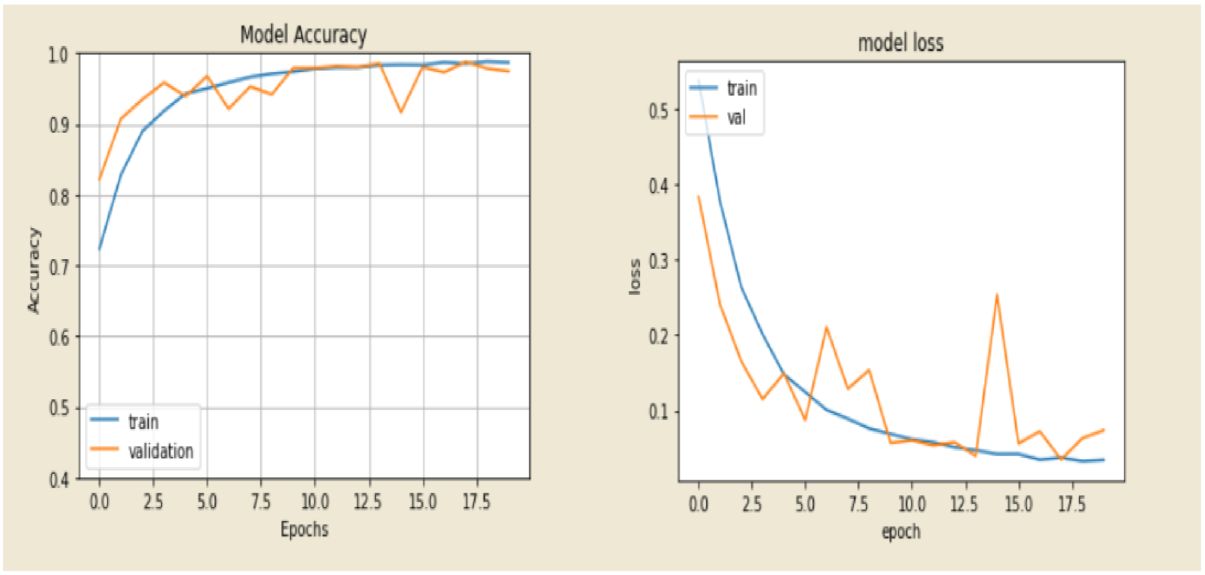
This model managed to classify 9 out 10 images in the prediction set correctly. The image given below was the only one that was associated to the wrong class. This painting was painted by Cezanne. However, the model mistakenly labeled it as Others.



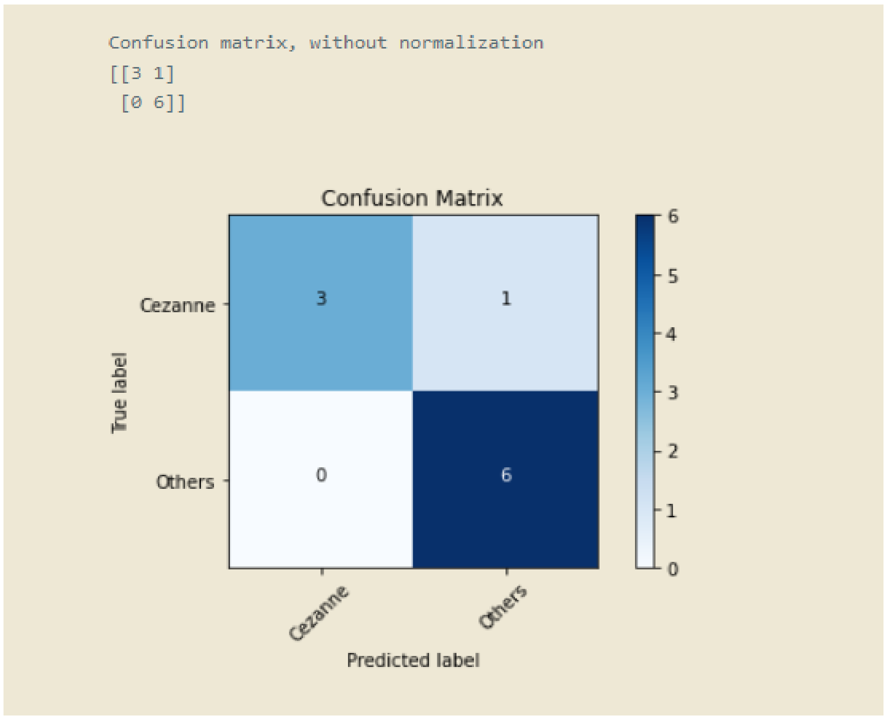
```
Painted by others
tf.Tensor(0.999944, shape=(), dtype=float32)
```



By visualizing the accuracy and loss curves, we can observe that the training accuracy and training loss are strictly increasing and decreasing respectively. For validation accuracy and loss, even though there are some fluctuations, they are improving notably.

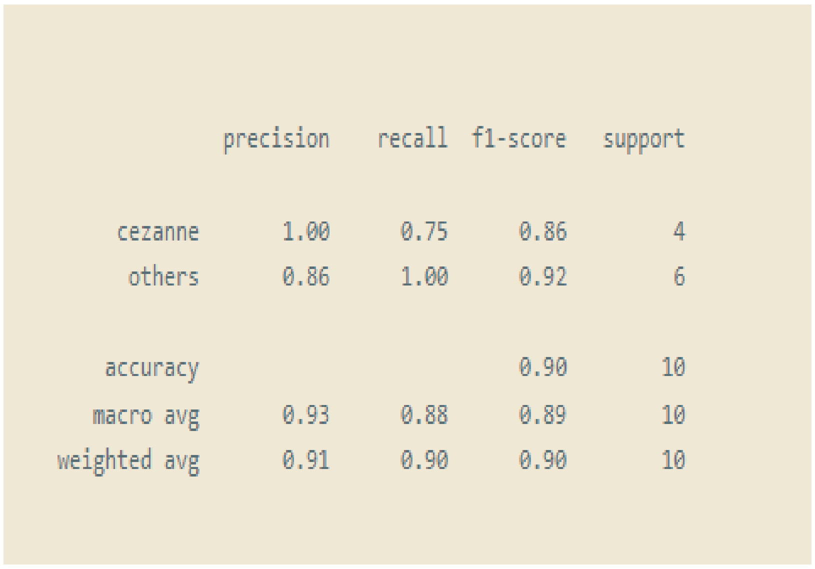


By visualizing the confusion matrix, we can observe that this model made the correct predictions for the 9 out of the 10 unseen images.



```
Confusion matrix, without normalization
[[3 1]
 [0 6]]
```

The figure below shows the performance of the CNN concerning the key metrics.

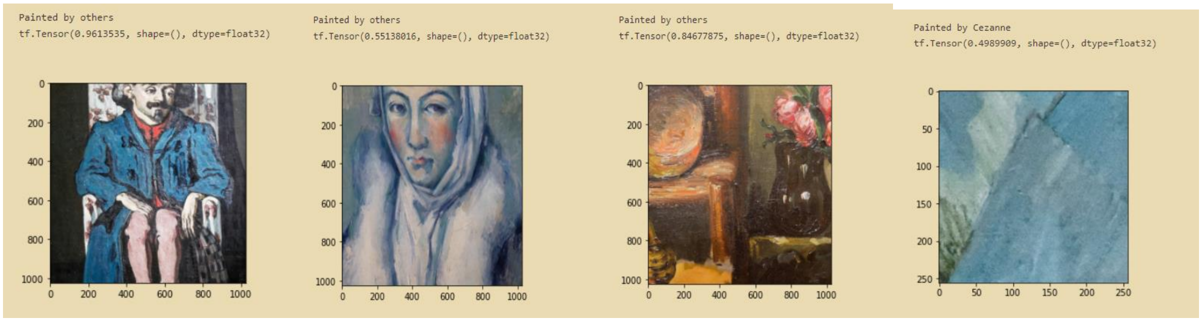


```
              precision    recall  f1-score   support

     cezanne       1.00      0.75      0.86         4
      others       0.86      1.00      0.92         6

    accuracy                           0.90        10
   macro avg       0.93      0.88      0.89        10
weighted avg       0.91      0.90      0.90        10
```

Results of VGG16 Random Crops 8-1024:

To my surprise, even though this model had an accuracy rate of about 100, it performed poorer than the previous CNN. The model misclassified 4 out of 10 paintings. As we can see, the model often tends to mistake Cezanne's works with others.

This CNN model classified the following painting incorrectly:



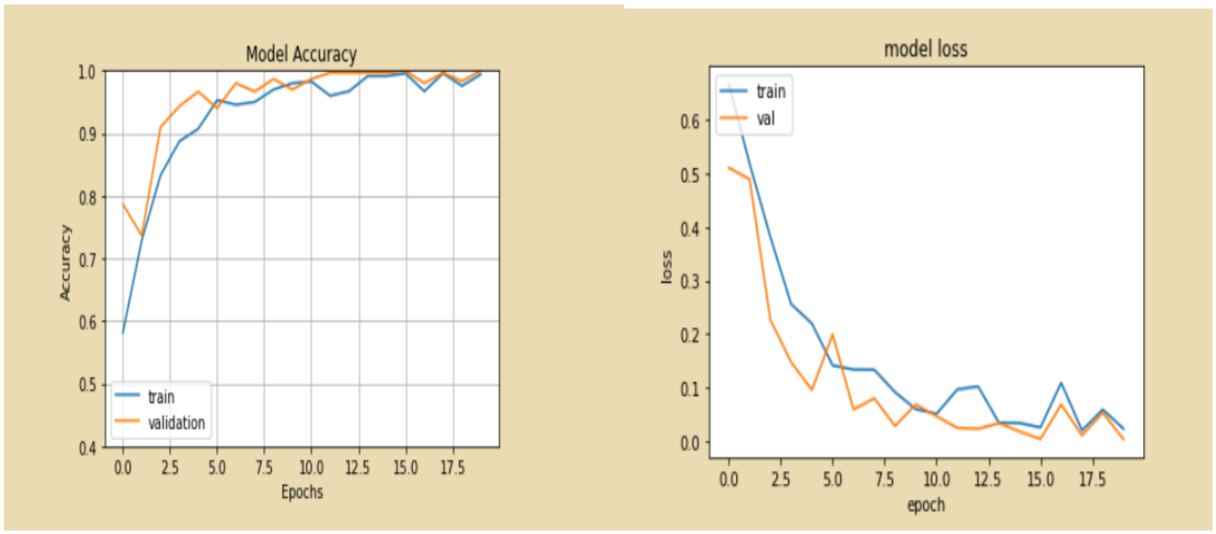
```
Painted by others
tf.Tensor(0.9613535, shape=(), dtype=float32)
```

```
Painted by others
tf.Tensor(0.55138016, shape=(), dtype=float32)
```

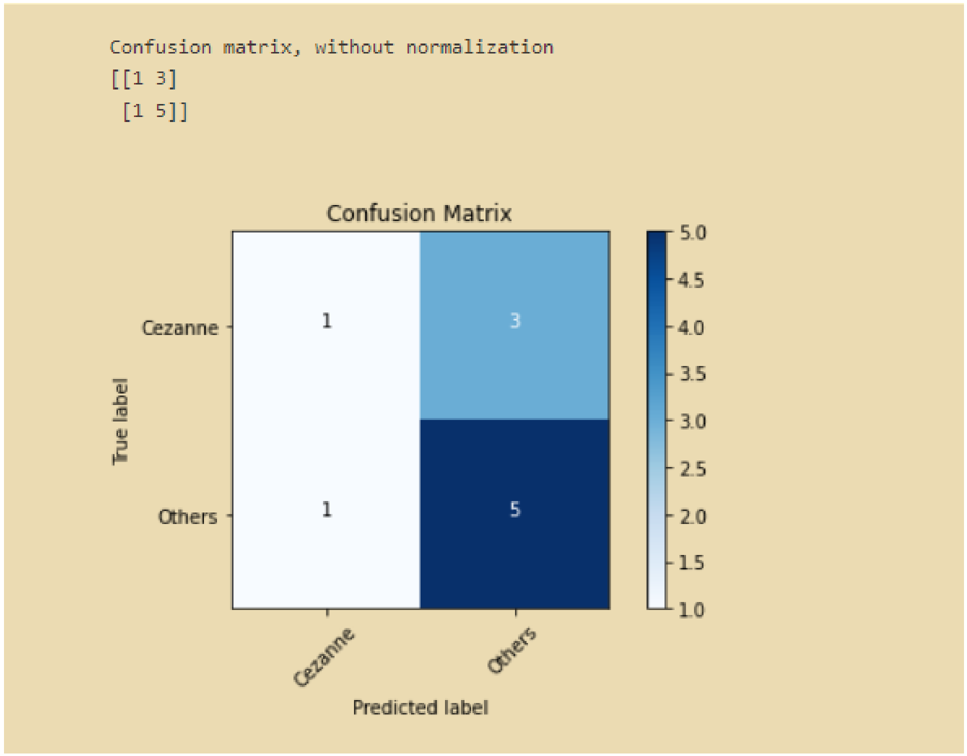
```
Painted by others
tf.Tensor(0.84677875, shape=(), dtype=float32)
```

```
Painted by Cezanne
tf.Tensor(0.4989909, shape=(), dtype=float32)
```

By visualizing the accuracy and loss curves, we can observe that the training accuracy and training loss are strictly increasing and decreasing respectively. For validation accuracy and loss, even though there are some fluctuations, they are improving notably.



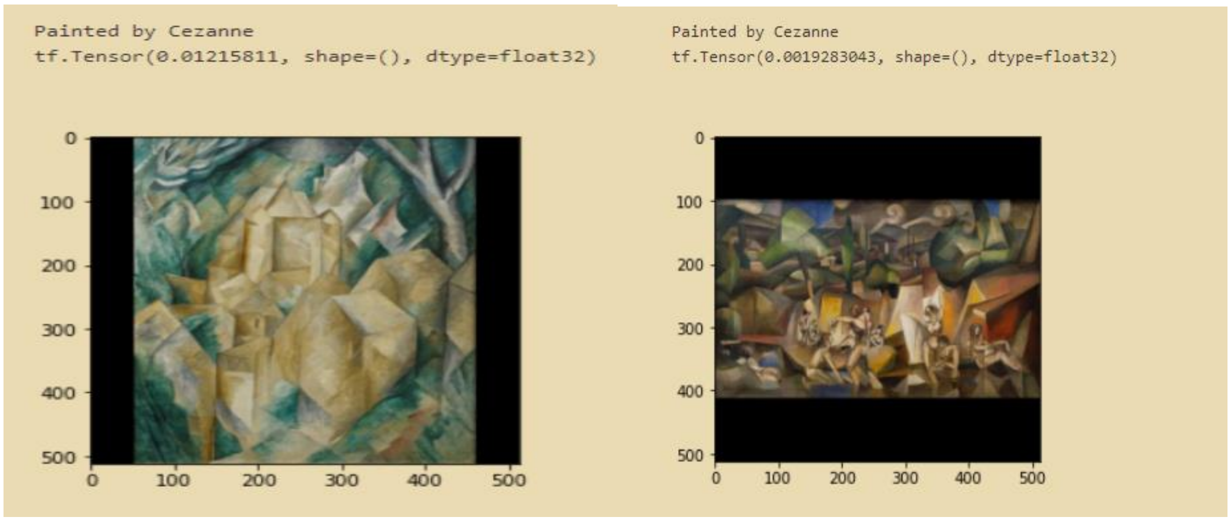
By visualizing the confusion matrix, we can observe that this model made the correct predictions for the 6 out of the 10 unseen images.



```
Confusion matrix, without normalization
[[1 3]
 [1 5]]
```

Results of VGG16 Black Padding Approach 512x512:

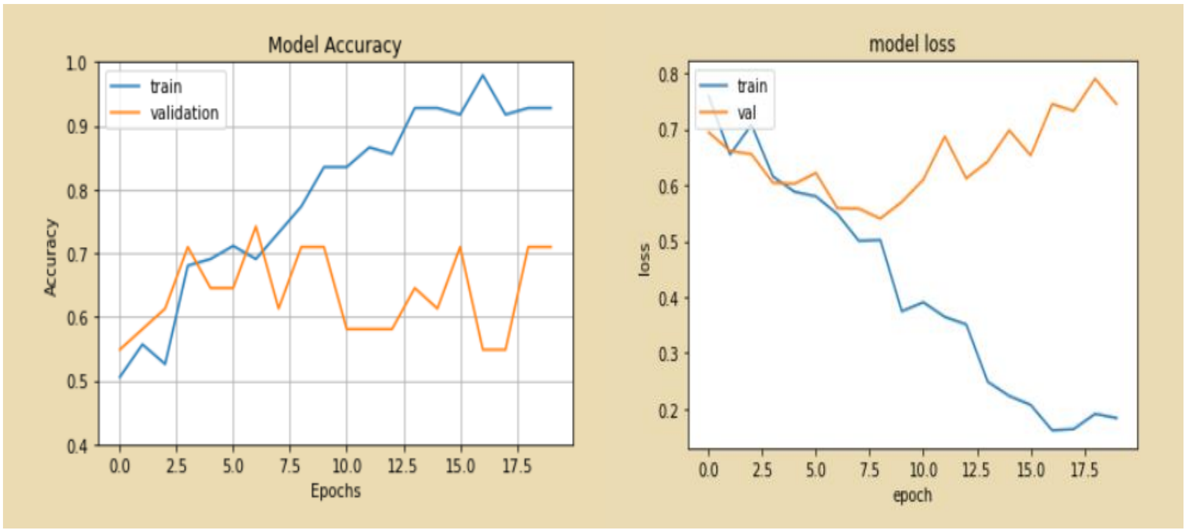
For the black padding approach which had the validation accuracy rate and validation loss of 74 and 84 respectively, I had separated 7 painting for the prediction set, all of which have been reduced to size 512x512. This model managed to classify 5 images in the prediction set correctly, which is impressive I found impressive considering its high validation loss value. The images given below are ones that have been misclassified



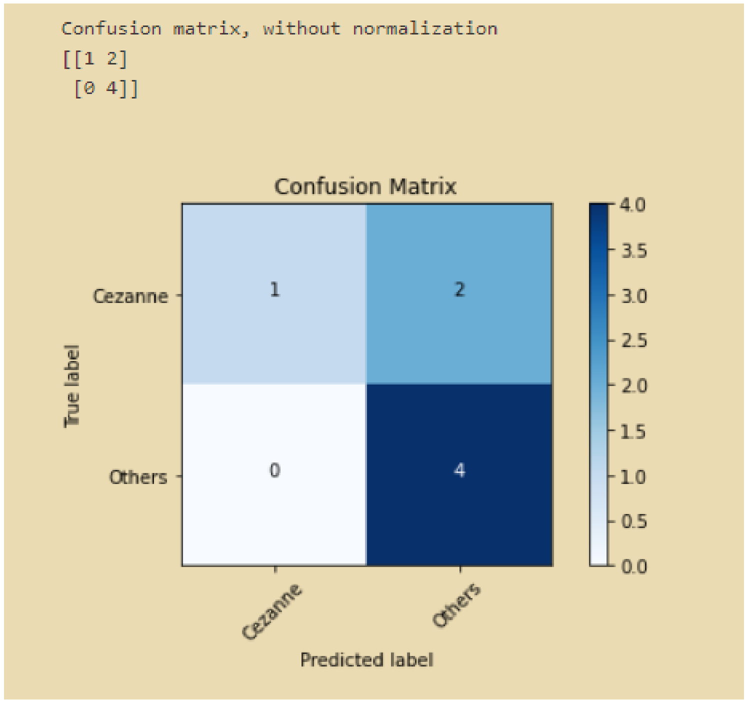
```
Painted by Cezanne
tf.Tensor(0.01215811, shape=(), dtype=float32)
```

```
Painted by Cezanne
tf.Tensor(0.0019283043, shape=(), dtype=float32)
```

By visualizing the accuracy and loss curves, we can observe that the training accuracy gets as high as 92 while the validation accuracy end with 70. The validation high is also notably high. This means that the model is highly overfitted.

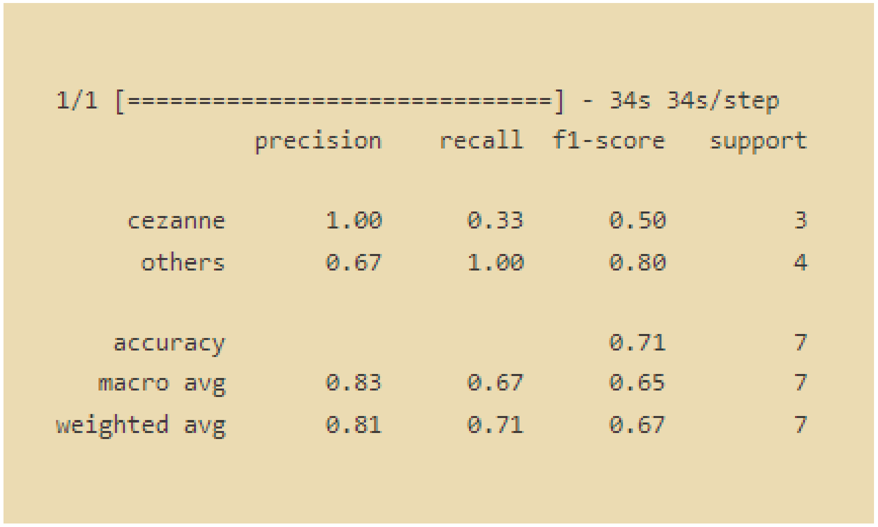


By visualizing the confusion matrix, we can observe that this model made the correct predictions for the 5 out of the 7 unseen images.



```
Confusion matrix, without normalization
[[1 2]
 [0 4]]
```

The figure below shows the performance of the CNN concerning the key metrics.



```
1/1 [==============================] - 34s 34s/step
              precision    recall  f1-score   support

     cezanne       1.00      0.33      0.50         3
      others       0.67      1.00      0.80         4

    accuracy                           0.71         7
   macro avg       0.83      0.67      0.65         7
weighted avg       0.81      0.71      0.67         7
```

Conclusion

For this project, I experimented with ResNet50 and VGG16 pretrained models. Contrary to my expectations, in my experiments the VGG16 outperformed the ResNet50, even though ResNet has much deeper layers and it is supposed extract much finer details than VGG16, but bearing in mind that I kept most of the convolutional blocks frozen and only trained the 5th block, I believe if I had adjusted the weights of all of the layers and had trained all of the parameters, I might have achieved high accuracies with ResNet50 as well.

I tried out different avenues and different lines of action regarding the size of the dataset its preprocessing phase. I assessed the outcomes of each and the results showed that after expanding the size of the dataset, the accuracy escalated significantly which shows the size of the dataset plays a crucial role when it comes to art classification tasks since the model needs enough samples to train and adapt its weights with. Based on this, one can validate that the style of the artist – at least for this particular case - is indeed present in each and every crop of the painting.

After trying out expansion of three different sizes and evaluating the models, the numbers proved that all three approaches performed relatively well with the VGG16 pretrained model. The best result belonged to the 120 crops of size 256x256 approach, upon which I trained all of the layers in model and achieved a final validation accuracy of 99.

Future Measures

For my final model that should be presented, I might try using the entire dataset for training the model rather than setting aside 12 paintings for testing, because considering the limited number of available paintings getting random crops of those 12 painting could lead to a much higher accuracy.

I will try fine-tuning more layers of the ResNet50 model and evaluate them as well to see if there are any notable improvements.

.

References

1. Eva Cetinic, Tomislav Lipic, Sonja Grgic. Fine-tuning Convolutional Neural Networks for fine art classification

2. Ibrahem Kandel, and Mauro Castelli. How Deeply to Fine-Tune a Convolutional Neural Network: A Case Study Using a Histopathology Dataset.

3. Kaiming He. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.

4. Dominik Scherer, Andreas Muller, and Sven Behnke. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition

5. Rob DiPietro. A friendly introduction to cross entropy loss.

6. Hadelin De Ponteves , Kirill Eremenko. Hands-On Artificial Neural Networks Udemy Course

7. Yiyu Hong1 and Jongweon Kim2. Art Painting Identification using Convolutional Neural Network

8. https://towardsdatascience.com/4-pre-trained-cnn-models-to-use-for-computer-vision-with-transfer-learning-885cb1b2dfc

9. Nitin Viswanathan, Artist Identification with Convolutional Neural Networks

10. Recognizing Art Style Automatically with deep learning Adrian Lecoutre, Benjamin Negrevergne, Florian Yger