FINAL PROJECT REPORT

An application of deep learning Identify artists from arts

Author: Nguyen Minh Duc
Student ID: 20202026

1 Model Description

A lot of models have been built and trained from scratch. The following model is one of the best among them. In this model, I took advantage of the powerful Resnet50 model and added 3 more dense layers to obtain 11 probabilities of each artists in the data set.
Firstly, images were re-size to $(224, 224, 3)$ to fit in resnet50 model. Convolution layers were used to reduce the size of the image, this is a part of the Resnet50 architecture.

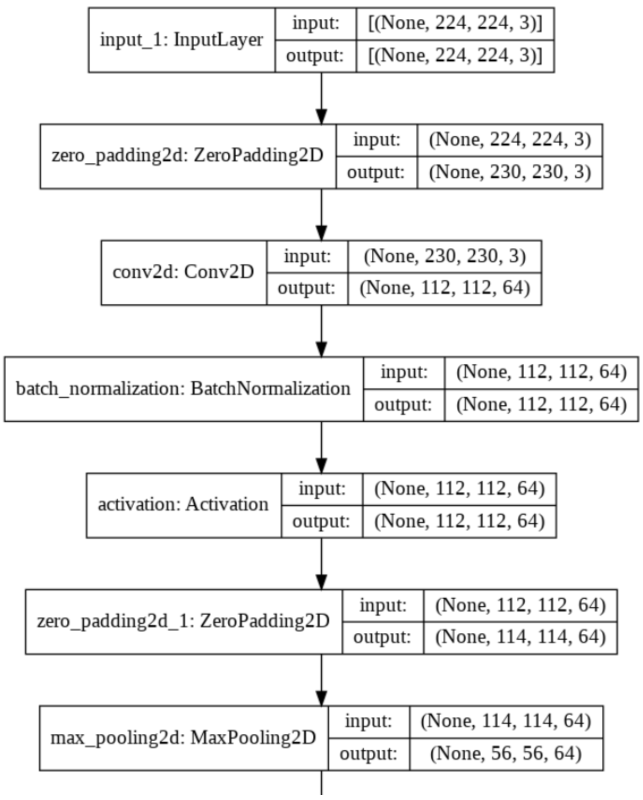

input_1: InputLayer
input: [(None, 224, 224, 3)]
output: [(None, 224, 224, 3)]
zero_padding2d: ZeroPadding2D
input: (None, 224, 224, 3)
output: (None, 230, 230, 3)
conv2d: Conv2D
input: (None, 230, 230, 3)
output: (None, 112, 112, 64)
batch_normalization: BatchNormalization
input: (None, 112, 112, 64)
output: (None, 112, 112, 64)
activation: Activation
input: (None, 112, 112, 64)
output: (None, 112, 112, 64)
zero_padding2d_1: ZeroPadding2D
input: (None, 112, 112, 64)
output: (None, 114, 114, 64)
max_pooling2d: MaxPooling2D
input: (None, 114, 114, 64)
output: (None, 56, 56, 64)

Figure 1: The first few layers

Secondly, 16 residual units were added according to the following figure

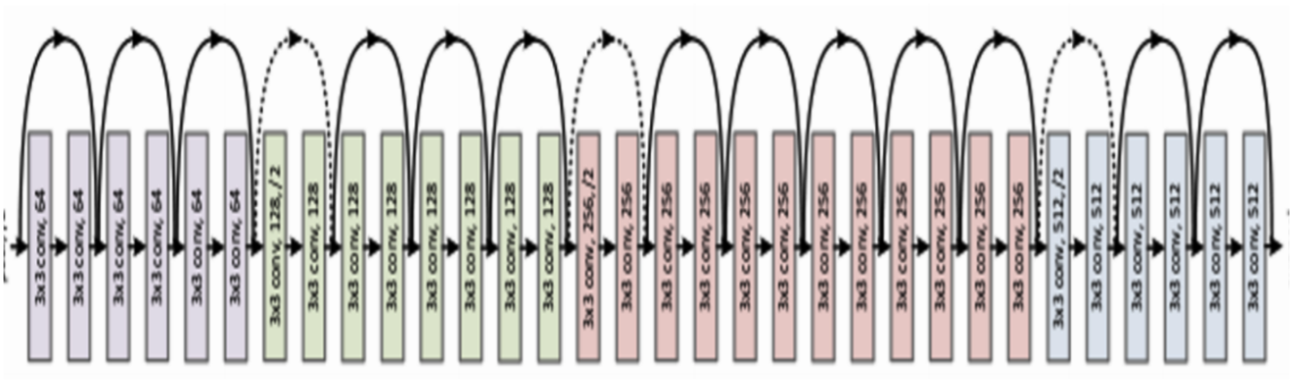


Figure 2: Residual units architecture

The following is the implementation of the residual units.

```python
class ResidualUnit50(Layer):
  def __init__(self, filters, strides = 1, activation = 'relu',
               is_first_layer = False, **kwargs):
    super().__init__(**kwargs)
    self.activation = tf.keras.activations.get(activation)
    if is_first_layer:
      strides = 1
    self.main_layers = [Conv2D(filters, 1, strides = strides,
                               kernel_initializer='he_normal'),
                        BatchNormalization(axis = 3),
                        self.activation,
                        Conv2D(filters, 3, padding = 'same',
                               kernel_initializer='he_normal'),
                        BatchNormalization(axis = 3),
                        self.activation,
                        Conv2D(4 * filters, 1, kernel_initializer='he_normal'),
                        BatchNormalization(axis = 3)]
    self.skip_layers = []
    if strides > 1 or is_first_layer:
      self.skip_layers = [Conv2D(4 * filters, 1, strides = strides,
                                 kernel_initializer='he_normal'),
                          BatchNormalization(axis = 3)]

  def call(self, inputs):
    Z = inputs
    for layer in self.main_layers:
      Z = layer(Z)
    skip_Z = inputs
    for layer in self.skip_layers:
      skip_Z = layer(skip_Z)
    return self.activation(Z + skip_Z)

prev_filters = 0
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
```

```
strides = 1 if filters == prev_filters else 2
model.add(ResidualUnit50(filters, strides = strides,
                         is_first_layer = (prev_filters == 0)))
prev_filters = filters
```

Lastly, to obtain the probabilities of each of the artists, 3 more fully connected layers were added to the model. In keras library, this was done by adding Dense layers. Moreover, to avoid over-fitting, 2 dropout layers with probability of 0.3 were introduced.

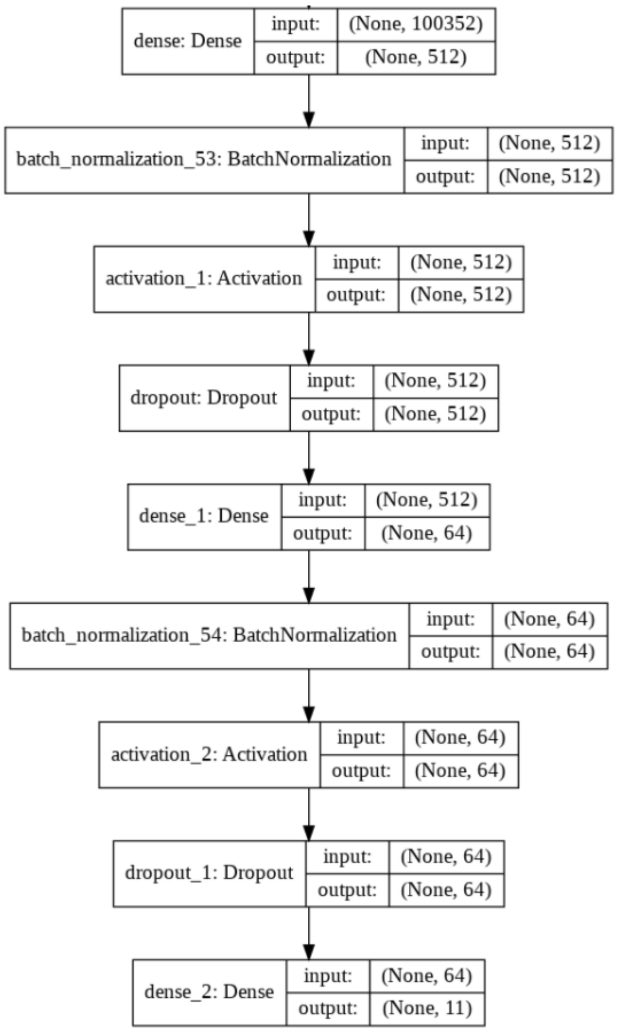


Figure 3: Fully connected layers

The output should be 11 probabilities with respected to 11 artists. Altogether, they had more than 75 million parameters, in which 54 thousand were non-trainable

2 Training

2.1 Data pre-processing

First and foremost, 11 artists were encoded by index numbers, ranging from 0 to 10.
Secondly, the data set was divided into two sets: one for training (80%) and one for validation

(20%)
Thirdly, I notice the data set is not very balanced. The first artist, Vincent van Gogh, has over 800 arts, and Edgar Degas has almost 700, while others have around 200 to 400 paintings. Thus, it is natural to assign each class a class-weight base on how many paintings that artist has with respected to everyone to control the model not to bias anyone. In particular, a class-weight is defined as

$$w_i = \frac{n}{mp_i},$$

where w_i, p_i are class-weight and the number of paintings of the i-th artist, and n, m are the total number of paintings and the number of artists respectively. By doing this, the artists that have fewer paintings will have higher class-weight, hence, should not be overwhelmed by others.
Then, the data set is augmented by randomly flipping both vertically and horizontally, by normalizing the RGB matrix and by stretching the image, which should provide more opportunity for the model to learn.
Lastly, batch size and input shape must be determined. Because I used Resnet50, input shape was $(224, 224, 3)$ and after a lot of experiments, batch_size = 16 yielded the best result.

2.2 Training details

To avoid over-fitting, I used dropout with probability of 0.3 after each dense layer. Details of why 0.3 was chosen will be mentioned in Section 4.
Before starting the training, epoch was set to 100 and two regularizer from keras were chosen: EarlyStopping to stop the model when there aren't any improvements and ReduceLROnPlateau to reduce the learning rate if there aren't any improvements.
The last component was choosing the right optimizer. I experimented with Adam, SGD and RMSprop and Adam with 10^{-4} initial learning rate provides the best result. Details of why Adam was chosen will be mentioned in Section 4.
To further prevent over-fitting, data would be randomly shuffled during training process.

```
optimizer = Adam(learning_rate = 0.0001)
model.compile(loss = 'categorical_crossentropy',
              optimizer = optimizer, metrics = ['accuracy'])

n_epoch = 100

reduce_learning_rate = ReduceLROnPlateau(monitor='val_loss',
                                         factor = 0.1, patience = 5,
                                         verbose = 1)

early_stop = EarlyStopping(monitor='val_loss', patience = 10, verbose=1,
                           mode = 'auto', restore_best_weights = True)

history = model.fit_generator(generator = train_generator,
                              steps_per_epoch=STEP_SIZE_TRAIN,
                              validation_data=valid_generator,
                              validation_steps=STEP_SIZE_VALID,
                              epochs=n_epoch,
                              class_weight = class_weights,
                              shuffle = True,
                              verbose=1,
                              callbacks = [reduce_learning_rate, early_stop],
                              use_multiprocessing = True,
                              workers = 16)
```

3 Performance Evaluation

After the training, the model was evaluated as follows.

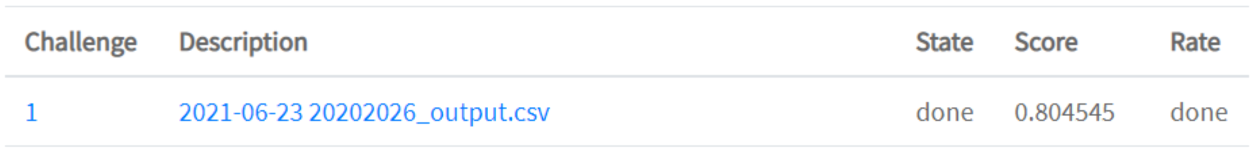
```
output = model.evaluate(train_generator)
print("Training evaluation:", output)

194/194 [==============================] - 96s 496ms/step - loss: 0.6318 -
accuracy: 0.8102
Training evaluation: [0.6317920088768005, 0.8101552128791809]

output = model.evaluate(valid_generator)
print("Validation evaluation:", output)

48/48 [==============================] - 24s 492ms/step - loss: 1.1515 -
accuracy: 0.6258
Validation evaluation: [1.1515074968338013, 0.6258148550987244]
```

Submitting to the scoreboard website, I obtained the result from test data, which is very close to the training evaluation.



Challenge	Description	State	Score	Rate
1	2021-06-23 20202026_output.csv	done	0.804545	done

Figure 4: Test result

Two graphs were plotted to compare the performance of the model to the training and validation data set. It could be seen that the two graphs were good approximation of each other, however, after around 35 epochs, the two slowly separated and started to converge. This is when EarlyStopping function stopped the training and restored the best weights.

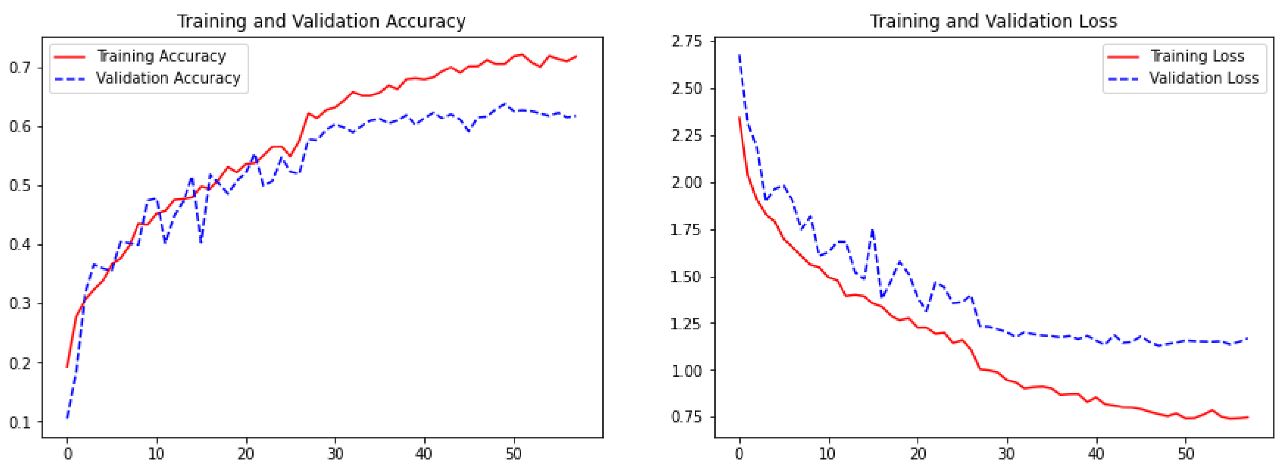
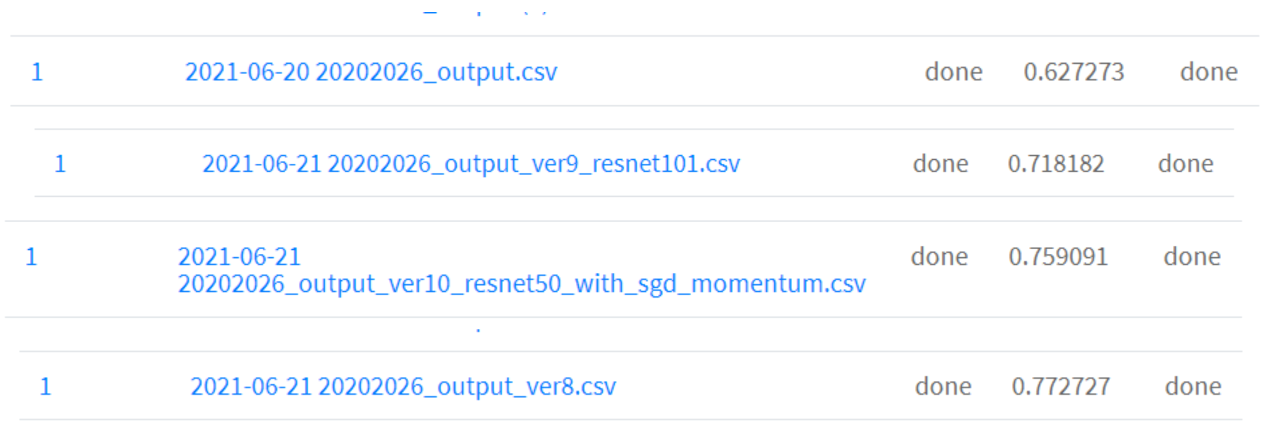


Figure 5: The graphs of accuracy and loss during the training

4 Analysis

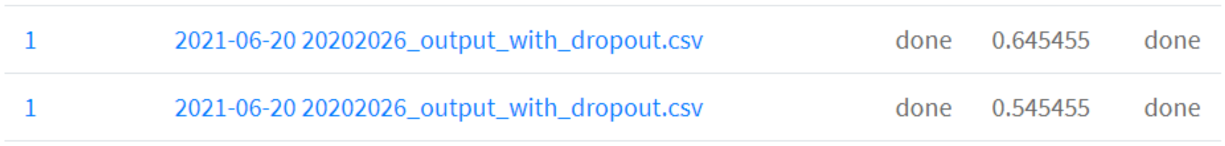
In this section, I will provide some experiments that justify my choices of parameters and model architecture to obtain the result.
To begin with, let's analyze the optimal batch size for the model. At first, I fixed the batch size at 32 since this was pretty standard and tried different models. Now, let's talk about why Resnet50 was chosen. I first implemented a simple CNN model with only a few convolution and fully connected layers, which yielded about 60% accuracy on the test data. Then, I proceeded with Resnet101 with 3 fully connected layers: Dense(512), Dense(16) and Dense(11). But the model converged too soon and obtained 75.9 % accuracy on the test data. Thus, it is natural to increase the number of layers to Resnet101, however, it converged even faster, yielded 71.82% accuracy. I also tried to build another architecture such as Xception, Inception and VGGnet but they converged too fast that did not pass 40% accuracy on the training data set. Now, Resnet50 might be the best model to use, I then tried to experimented with different batch size: 16, 64 and 128 and found out that batch_size = 16 produces best result at 77%.



1	2021-06-20 20202026_output.csv	done	0.627273	done
1	2021-06-21 20202026_output_ver9_resnet101.csv	done	0.718182	done
1	2021-06-21 20202026_output_ver10_resnet50_with_sgd_momentum.csv	done	0.759091	done
1	2021-06-21 20202026_output_ver8.csv	done	0.772727	done

Figure 6: Accuracy of models on selecting architecture and batch size phase

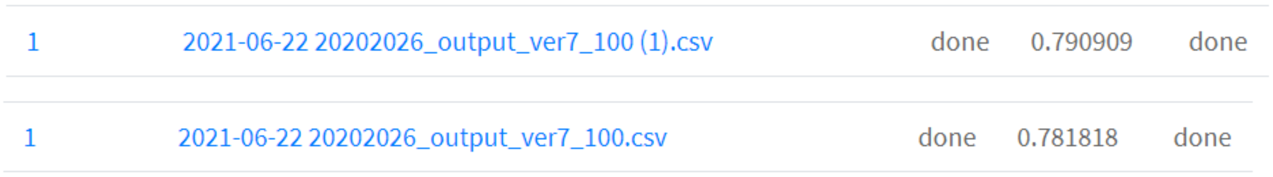
At this point, the model performs well on the training data set at more than 92% but on the validation and test data set, it was not as impressive. Thus, something had to be done to avoid over-fitting. This is when regularizers were introduced. Two dropout layers with probability 0.5 were added after each Dense layer. EarlyStopping and ReduceLROnPlateau functions were also added on the training process, which gave 54.5% accuracy on test data. The performance was worse but the training and validation loss were good approximation of each other. I then reduced to 0.3 and obtained 64.5% accuracy.



1	2021-06-20 20202026_output_with_dropout.csv	done	0.645455	done
1	2021-06-20 20202026_output_with_dropout.csv	done	0.545455	done

Figure 7: Accuracy of models on selecting regularizers phase

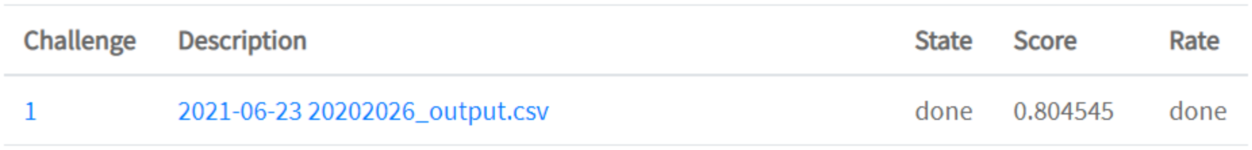
Then, it was time to experimented with the best optimizers. The above results were carried out by SGD with momentum. I then tried Adam and RMSprop, on which Adam outperformed the three giving 79% accuracy



1	2021-06-22 20202026_output_ver7_100 (1).csv	done	0.790909	done
1	2021-06-22 20202026_output_ver7_100.csv	done	0.781818	done

Figure 8: Accuracy of models with Adam and RMSprop

The last thing could be changed is the number of dimension of fully connected layers. The original model was Dense(512) and Dense(16). After going through different dimensions and adding more layers, I discovered that Dense(512) and Dense(64) performed best at 80% accuracy while other choices could not pass 70% threshold.



Challenge	Description	State	Score	Rate
1	2021-06-23 20202026_output.csv	done	0.804545	done

Figure 9: Changing the last Dense layer to 64 units

This is the best model that I could build during the project.