1. Introduction

In this project, we applied deep learning networks to recognize the corresponding artist of painting pictures based on the colors used and geometric patterns inside the pictures. To train the model, we used the dataset posted on Kaggle consisting of painting images labeled with the Artists' name. We experimented with four different neural networks, which are MLP, Customize CNN, Two pretrain models (ResNet and VGG). Our best model is the ResNet, which achieve an accuracy of over 85% on the test set.

2. Description of individual work

I worked on the data loading, implementing data preprocessing, building the models of MLP, ResNet and VGG. During the hyperparameter tuning, I explored methods to improve the models I built.

3. Description of portion

Data Loading

I first wrote the code to load and transform the data, using OpenCV. The code is included in the **Data_loading.py**. At the same time, I use loop to create the artist labels / target variable for the images.

```
for name in artists_top_name:
    for path in [f for f in os.listdir('/home/ubuntu/Final_project/best-artworks-of-all-time/images/images/'+ name)]:
        img = cv2.resize(cv2.imread('/home/ubuntu/Final_project/best-artworks-of-all-time/images/images/'+ name +"/" + path),resize)
        imgs.append(img)
        label.append(name)
```

Data Preprocessing

After I did some exploratory data analysis, I find the following issues needed to be addressed:

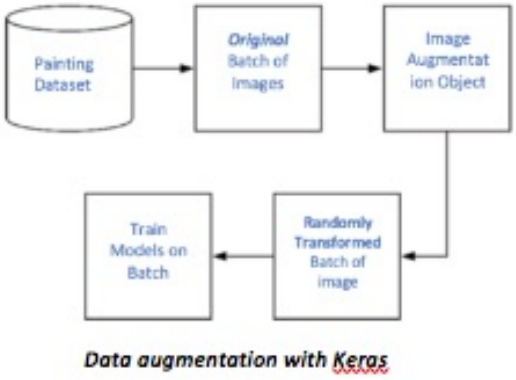
1) The dataset has 50 artists. However, some of the artists have only a few training samples.

2) Imbalance dataset. The largest class is almost 4 times larger than the smallest class.

3) Size of the images are different.

name	paintings	class_weight
Vincent van Gogh	877	0.445631
Edgar Degas	702	0.556721
Pablo Picasso	439	0.890246
Pierre-Auguste Renoir	336	1.163149
Albrecht Dürer	328	1.191519
Paul Gauguin	311	1.256650
Francisco Goya	291	1.343018
Rembrandt	262	1.491672
Alfred Sisley	259	1.508951
Titian	255	1.532620

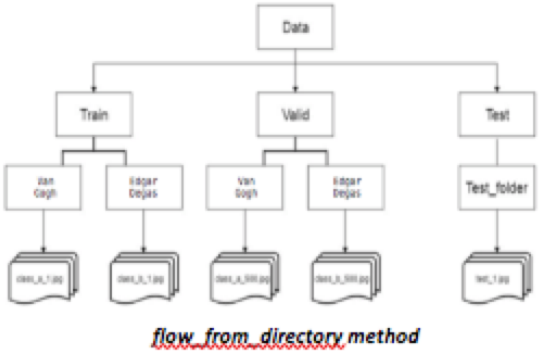
In order to reduce computation and better training, I came up with the idea of only build model and identify the top ten artists, so that each category should have enough sample size. The next step I did is the image augmentation to solve the imbalance issue. The training set is added with more training samples for the under-represented classes. The augmentation was carried out by changes in scale, rotations, shearing, and horizontal & vertical flips etc. The augmented images were created using the ImageDataGenerator API in Keras. It is the in-place data augmentation that we ensure that our network, when trained, sees new variations of our data at each and every epoch. I also created the class_weight for each category for our later models. So, the under-represented classes get more attention.



Data augmentation with Keras

```
datagen = ImageDataGenerator(rescale=1./255.,
                             validation_split=0.2,
                             rotation_range=50,
                             width_shift_range=0.2,
                             height_shift_range=0.2,
                             shear_range=5,
                             horizontal_flip=True,
                             vertical_flip=True)
```

We have also applied the flow_from_directory method in the ImageDataGenerator. It is very convenient that it creates the pipeline to directly take the path to a directory and generates batches of augmented data, preprocess data, and transform to format needed. Through the online tutorial, I know it requires a certain file and folder structure to use the method. I then manipulated and set up the directory to the path of the directory that contains the sub-directories of the respective classes, like the structure below. Each subdirectory is treated as a different class. The name of the class is inferred from the subdirectory name.

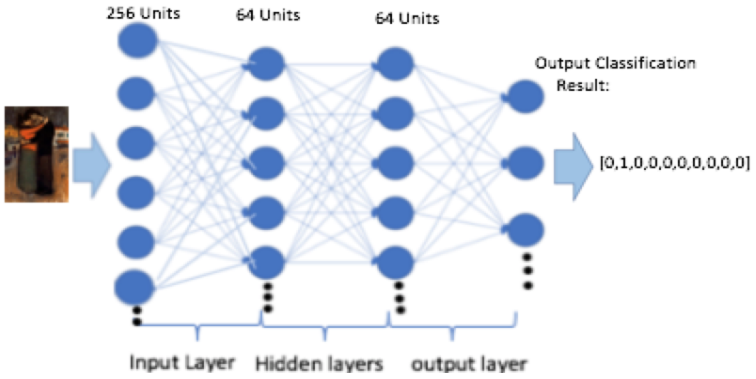


flow_from_directory method

```
train_generator = datagen.flow_from_directory(directory=images_dir,
                                              class_mode='categorical',
                                              target_size=image_size,
                                              batch_size=batch_size,
                                              subset="training",
                                              shuffle=True,
                                              classes=artists_top_name,
                                              color_mode="rgb"
                                             )
```

Multi-layer Perceptron

To begin with the model, I built Multi-layer perceptron as our baseline model. The model we used was a three-layer MLP network. The input dimension is 150528 (224*224*3). The number of neurons for each layer is 125, 64, and 32, respectively. The activation functions used in the hidden layers are Relu and the output layer is used the Softmax activation function, for the purpose of calculating the probability distribution of multi-class outputs. The optimizer is Adam, and the loss function is Categorical Cross Entropy, a function that is specifically targeted at categorical outputs.



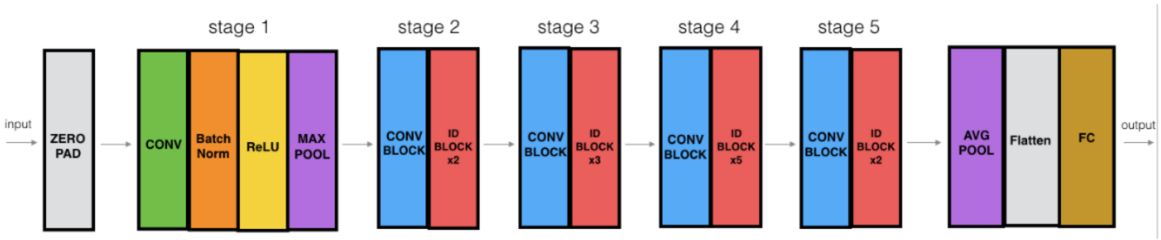
Below is my code for the model part. I added "dropout" to randomly ignore pertain amount of neurons during the training phase, for the purpose of preventing the over-fitting, which increase the accuracy for about 5%.

```
model = Sequential([
    Dense(N_NEURONS[0], input_dim=150528, kernel_initializer='uniform'),
    Activation("relu"),
    Dropout(DROPOUT),
    BatchNormalization()])
for n_neurons in N_NEURONS[1:]:
    model.add(Dense(n_neurons, activation="relu", kernel_initializer='uniform'))
    model.add(Dropout(DROPOUT, seed=SEED))
    model.add(BatchNormalization())
model.add(Dense(10, activation="softmax", kernel_initializer=weight_init))
model.compile(optimizer=Adam(lr=LR), loss="categorical_crossentropy", metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=N_EPOCHS, validation_data=(x_test, y_test))
```

After the hyperparameter tuning, the best accuracy I got for MLP is 58%. This is obvious not a very good score and I turned to improve this number by using the pre-trained models.

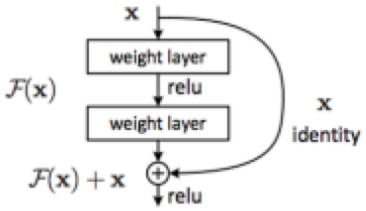
Pretrained Model

I did the online research and tried multiple pretrained architectures. I found out ResNet50 and VGG pretty on the similar topic of project. Here, I majorly highlight my work with ResNet50.

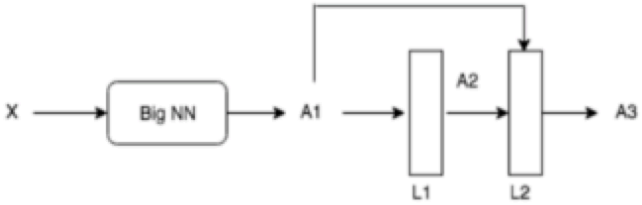


ResNet-50 Model

About the ResNet50, one unique feature I learned is the function of "skip connection", which allows the model to stack additional layers and build a deeper network. Like below left, the residual block helps to skip over layers to avoid the problem of vanishing gradients (gradients get really small). As result, we can build the network deep and extract more useful information.



Logical scheme of base building block



Term A1 will be passed to L2.

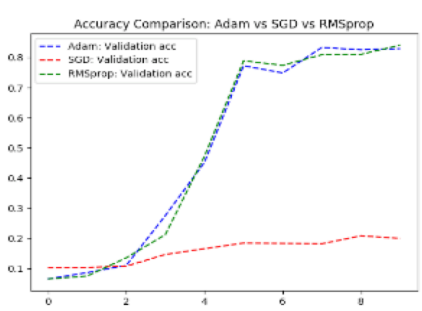
Like it exhibited on the right, with skip connection, term A1 will be passed to L2. The equation of A3 is modified as: A3 = relu (W2 *A2+b2 +A1).

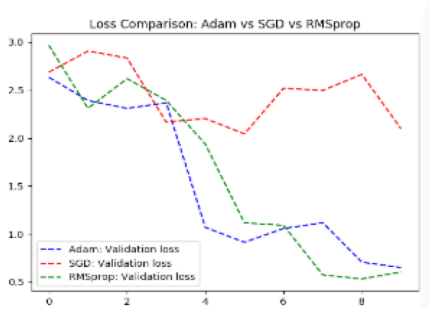
The model is built in Keras. Below is the highlight of codes for the model. I first choose to load with the pretrained ImageNet weights. I set include_top=False to not include the final pooling and fully connected layer in the original model, so that my new output layer can be added and trained. The model is followed by another two layer of neurons and the output layer.

```
# Load pre-trained model
pre_trained_model = ResNet50(weights='imagenet', include_top=False, input_shape=model_input_shape)
# Add layers at the end
ResNet50_model = pre_trained_model.output
ResNet50_model = Flatten()(ResNet50_model)
N_NEURONS=(512,16)
for n_neurons in N_NEURONS[1:]:
    ResNet50_model = Dense(n_neurons, kernel_initializer='uniform')(ResNet50_model)
    ResNet50_model = BatchNormalization()(ResNet50_model)
    ResNet50_model = Activation('relu')(ResNet50_model)
output = Dense(n_classes, activation='softmax')(ResNet50_model)
model = Model(inputs=pre_trained_model.input, outputs=output)
optimizer = Adam(lr=0.0001)
# Compile the model
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

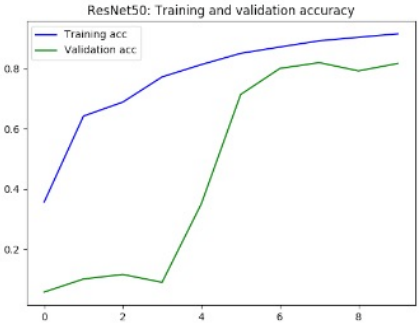
Hyperparameter Tuning

Optimizer: I did some analysis on the different performance of different optimizer. By comparing the three optimizations, Adam learns the fastest and it is more stable than the other optimizers, it doesn't suffer any major decreases in accuracy.

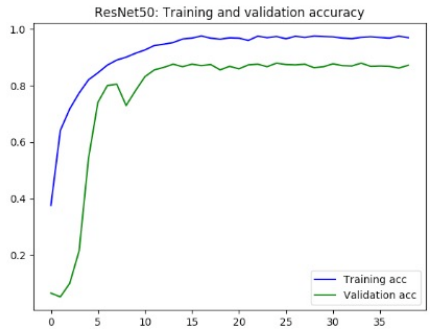




Epoch: Training the model for more iterations might improve the performance, at the cost of computation resource.



Training with 10 Epoch



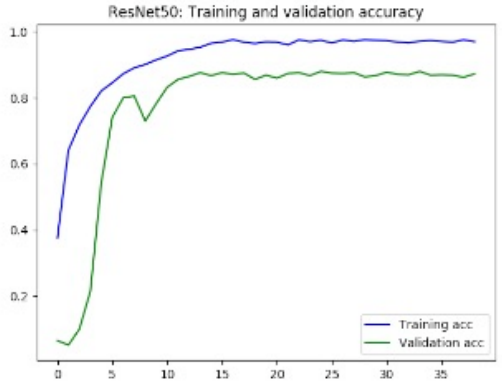
Training with 40 Epoch

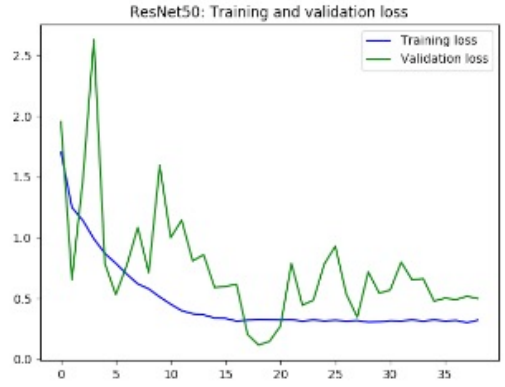
Callback Feature: I added the early stop and ReduceLROnPlateau, which helps reduce learning rate when a metric has stopped improving. ReduceLROnPlateau is designed to reduce the learning rate after the model stops improving with the hope of fine-tuning model weights.

```
early_stop = EarlyStopping(monitor='val_loss', patience=20, verbose=1,
                           mode='auto', restore_best_weights=True)

# Reduce learning rate when a metric has stopped improving.
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=4,
                              verbose=1, mode='auto')
```

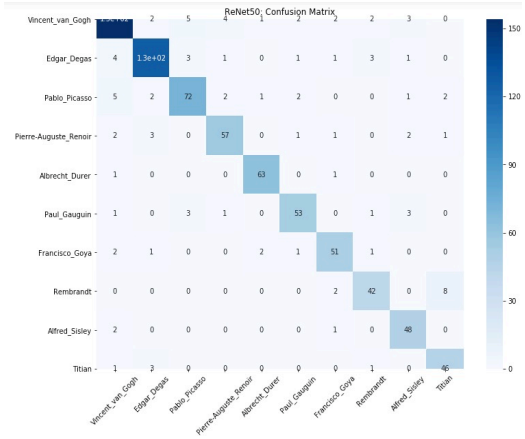
Training Loss and Validation Loss Over Time:





4. Result

The best accuracy rate I got is about 87%. I created a confusion matrix and classification report to describe the performance of models on the test dataset.



Classification Report

	precision	recall	f1-score	support
Vincent_van_Gogh	0.91	0.85	0.88	175
Edgar_Degas	0.92	0.93	0.92	140
Pablo_Picasso	0.87	0.83	0.85	87
Pierre-Auguste_Renoir	0.91	0.88	0.89	67
Albrecht_Dürer	0.88	0.98	0.93	65
Paul_Gauguin	0.87	0.85	0.86	62
Francisco_Goya	0.79	0.78	0.78	58
Rembrandt	0.81	0.81	0.81	52
Alfred_Sisley	0.80	0.96	0.88	51
Titian	0.82	0.82	0.82	51
accuracy			0.87	808
macro avg	0.86	0.87	0.86	808
weighted avg	0.87	0.87	0.87	808

5. Summary and Conclusion

In this project, we successfully built four networks that help us to recognize the artist of a painting picture based on the colors used and geometric patterns. Overall, pretrained models have significantly better performance than the self-designed network. Personally, I am able to practice and apply the deep learning framework we have learned in the class. It helps me to better understand the model and its parameter. By doing the research online, I am also to learn the new model such as the pretrain model like Resnet and VGG.

To further improve model performance, we would like to implement more epochs to train our models. And, we also hope to apply model ensemble techniques to incorporate the strength from our models.

6. Percentage of code

Lines of Code copied from the internet: 140
Line of Code modified 120
Line of Code added 80
(140-120)/ (140+80) = 10%

7. Reference

https://d2l.ai/chapter_convolutional-modern/resnet.html

https://medium.com/@vijayabhaskar96/tutorial-image-classification-with-keras-flow-from-directory-and-generators-95f75ebe5720

https://www.learnopencv.com/keras-tutorial-fine-tuning-using-pre-trained-models/

https://machinelearningmastery.com/how-to-use-transfer-learning-when-developing-convolutional-neural-network-models/

https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d