Final Project Report
Deep Learning (DS 6050)
Dr. Sheng Li
1 May 2023

Convolutional Neural Networks for Impressionist Painter Classification

Hyun Ko (npm5ct), Griffin McCauley (kzj5qw), Eric Tria (emt4wf)

I. Motivation

The main motivation and goal for this project was to distinguish the unique styles of different impressionist painters using deep learning. In the 19th century, a famous art movement known as Impressionism emerged, characterized by the use of small, visible brush strokes, open composition, and a focus on depicting the changing qualities of light. This movement also involved using ordinary subject matter, unconventional visual angles, and movement as an important element of human experience. Notable figures who contributed to the development of Impressionism in France included Frederic Bazille, Paul Cezanne, Edgar Degas, Edouard Manet, and Claude Monet [6].

Every artist has his or her own distinct style, and being able to learn which features distinguish one artist from the other will help in learning more about those styles. The motivation can also be split into educational and domain motivations. As an educational motivation, this project gave us an opportunity to apply deep learning techniques to a unique image classification problem. On the other hand, it also gave us a chance to learn more about Impressionist paintings and the styles of the artists from that period. Given the rise of generative models for paintings, being able to classify these paintings is an important task as the need arises for checking the authenticity of paintings.

This project gave our group the unique opportunity to explore different deep learning techniques and models in order to select the right method to classify Impressionist paintings. Since differentiating artist styles involve comparing intricacies between paintings, this project was also an excellent chance to use more advanced methods such as ensembling.

II. Datasets

The dataset called Impressionist Classifier Data was retrieved from Kaggle [1]. This dataset is for multi-class classification of ten impressionist painters: Camille Pissarro, Childe Hassam, Claude Monet, Edgar Degas, Henri Matisse, John Singer-Sargent, Paul Cezanne, Paul Gaugin, Pierre-Augste Renoir, and Vincent van Gogh. The dataset includes a total of 4000 paintings for training (400 for each artist) and a total of 1000 paintings for validation (100 for each artist). All of the painting images are in .jpg format, and, while their resolutions and dimensions vary, they have a size of 2.36 GB in total.

III. Related Work

For this project, we have looked at various research papers that discuss different deep learning techniques used to classify and extract significant features in the context of fine-art paintings since classifying such paintings would need a creative solution. One method proposed by Rodriguez et. al. [3] is based on transfer learning and dividing a painting into subregions where those subregions would then be classified. The proposed method aims to improve classification accuracy compared to other baseline methods used for fine-art paintings. Sandoval et. al. [4] also discusses a process that involves splitting up a painting into subregions. Their paper proposes a two-stage process for classifying fine art paintings. The first stage splits up the image into five patches and uses a deep CNN to classify each patch separately. In the second stage, the output of the five patches are combined in a decision-making module that infers the final labeling decision.

There are other methods such as the one proposed by van Noord et. al. [5] which involves a multi-scale CNN method that addresses both scale-variant and scale-invariant features. In their paper, they show that using both scales is beneficial to image recognition performance.

Banejri et. al. [2] offers a different use-case for CNNs with fine-art paintings. Their paper focuses on exploring the use of pre-trained CNNs as a feature extraction tool, which will then be used for classification. The paper also discusses the experiments they ran to identify which layers work best to classify paintings by their artist and style.

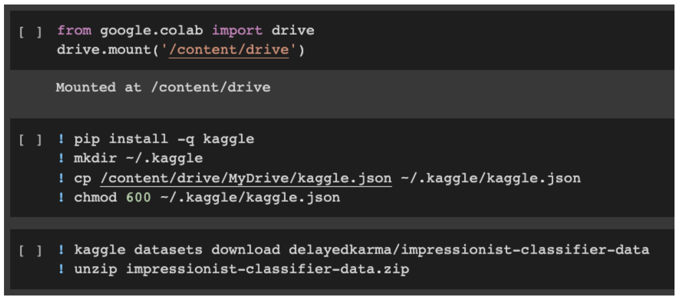
From these related works, our group focused on applying transfer learning on pre-trained CNN models as the foundation of our final model. Although we did not split the training images into subregions like in the related works, our project showed that transfer learning is still effective for this problem given the appropriate hyperparameter tuning and data augmentation techniques.

IV. Technical Approach

While the overarching mission of our project was to design and implement a model which would be able to swiftly and accurately evaluate the visual composition and style of a given piece of artwork and ascribe its authorship to a specific artist, there were a number of essential components which had to be constructed to support this. The four major elements that needed to be established are as follows: a method to load the data directly into Google Colab from Kaggle, a pipeline to process and convert the raw data into a form that can be input to a PyTorch multilayer perceptron model, a set of model architectures to be fine tuned and trained for our specific problem domain, and an evaluation schema that ensembles and aggregates the individual model predictions to produce a final accuracy metric. To accomplish these tasks, we primarily leveraged the libraries of numpy, pandas, torch, torchvision, matplotlib, and seaborn for various data manipulation, model building, and visualization purposes. To elaborate on our approach and methodology further, however, let us first begin with the data loading and subsequent processing.

a. Data Loading

Although datasets can easily be downloaded locally from Kaggle through its user interface, it can be tedious and time consuming to manually organize and reupload files to Google Colab each time a new runtime is instantiated, especially if the size of the files is large. Therefore, given that the dataset we were working with was almost 2.5 GB in size, we decided it would be best to configure our notebook such that the data would be automatically loaded into Google Colab from Kaggle each time the runtime restarted and the notebook was run. Thankfully, Kaggle's API strongly supports integration with the Google Colab ecosystem, and, with only the adjacent eight lines of code, we were able to download the data.



```
[ ] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive

[ ] ! pip install -q kaggle
    ! mkdir ~/.kaggle
    ! cp /content/drive/MyDrive/kaggle.json ~/.kaggle/kaggle.json
    ! chmod 600 ~/.kaggle/kaggle.json

[ ] ! kaggle datasets download delayedkarma/impressionist-classifier-data
    ! unzip impressionist-classifier-data.zip
```

b. Data Processing

Having successfully been able to implement automatic and direct data loading, we then turned our attention to developing a framework which would handle preliminary processing and conversion to a model consumable format. Since we intended to perform data augmentation as one of the final steps in our overall plan, this region of code also needed to be robust enough to support this later addition. To accomplish this, we opted to rely on the ImageFolder and DataLoader functions from torchvision.datasets and torch.utils.data, respectively. The ImageFolder function enabled us both to define a path from which to stream the data to the DataLoader object from and to apply and compose deliberately selected transformations from torchvision.transforms to the data as it is loaded into the model.

This technique afforded the desired level of flexibility and efficiency in an elegant manner, but one additional aspect of our data which had to be addressed here was the fact that it was only initially partitioned into testing and training sets on Kaggle; therefore, we needed to create our own test set for final model performance evaluation. There were a number of ways in which we could have approached this, but, due to the fact that data being streamed through the training data folder would later on be experiencing additional data augmentation that would not be appropriate to apply to the testing or validation sets, we decided to simply split our validation data in half using random_splits from torch.utils.data, resulting in a final training/validation/testing split of 80/10/10%. Now that the DataLoaders for the three sets were in place, we could begin the model construction and tuning process.

c. Model Architectures

Our model architectures and hyperparameter tuning is thoroughly discussed in the following Experiments section, but the general approach we took for this portion of the project was to identify and use transfer learning for a few prominent and proven pre-trained architectures and then fine-tune both models and the hyperparameters. All training was conducted using an Adam optimizer and cross entropy loss, and the base models were chosen primarily based on their impressive performances in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), their coverage during course lectures, and their availability through torchvision.models. Rather than simply incorporating the completely pre-trained parameterizations for these models, however, we made a few meaningful alterations which would allow for enhanced performance for our specific problem domain. For each model, we used the pre-trained convolutional layers as the base body of our architecture upon which we attached a fully connected multilayer perceptron (MLP) head which would calculate and produce the final multi-class classification output. The fully connected head consisted of two hidden layers of decreasing dimensionality prior to the final output layer, and each layer also included batch normalization, ReLU activation, and an optional dropout. All of these layer modules were imported from torch.nn, and were wrapped in a Sequential container.

In addition to the fully connected heads, we also attempted to fine-tune the more high-level and abstract representations embedded in the pre-trained convolutional layers of the base models by unfreezing just the final layers or blocks in those architectures while leaving the initial convolutional layers frozen. By freezing the early layers and unfreezing only the final layers, we enabled the model to retain its pre-trained low-level and highly generalizable representations such as vertical and horizontal lines and colors while allowing the more high level and problem specific representations captured by the final layers to be relearned and tuned for our specific dataset. This technique produced far better results than either fully freezing or fully unfreezing the pre-trained base model, and seemed to be the most appropriate approach to apply in practice. Once the architectures were determined and fixed, we were then able to parallelize the hyperparameter tuning process by having each member of our group perform the manual, pseudo-grid search discussed in the next section for each of our three selected base models.

d. Hyperparameter Tuning & Data Augmentation

Our hyperparameter tuning methodology involved searching over a range of values for learning rate, batch size, dropout rate, and weight decay (L2 regularization), and, once this process had converged to an optimal model configuration, we were able to add data augmentation to the training set to boost generalization, train each model one final time using these hyperparameters, and then store the .state_dict() attribute of the trained model to Google Drive using torch.save(). (Note that our training loop was implemented in such a way as to track the epoch parameterization that produced the best validation accuracy and output that parameterization for the resulting model, thus ensuring that the model being produced was indeed the highest performing model across the epochs.) At this point, having stored all of the tuned and trained models as .pt files in Google Drive, we were able to load them all into a single notebook and perform the final ensembling.

e. Ensembling

To execute the final ensemble and evaluation, we wrote a simple function that consumed the test set, our three models, and the cross entropy loss function. Passing these five arguments in, we then internally generated the 10 dimensional predicted class probability vectors for each model and implemented a soft voting policy of averaging the three vectors to get the final prediction vectors. The decision to use ensembling to tie all of our models together was made not only due to the "wisdom of the crowds" generally leading to better aggregate performance when many models are included in a final prediction but also because this procedure allowed us to meaningfully incorporate a variety of architectures and design schemes into our model in an implicit way without having to modify too much of the convolutional bodies ourselves or having to retrain every single layer from scratch. While there were many subjective choices to be made along the way to developing an optimal painting authorship classifier model, the overall strategy and approach we implemented followed a natural and logical flow and progression that we believed would lead to strong performance results while also afford us the best opportunity to explore a wide range of valuable techniques within a deep learning pipeline. For these reasons, we felt as though this was the ideal technical approach for our situation and are pleased with the results it produced.

V. Experiments

In order to get to the final ensemble model, the first step that our group did was to select the pre-trained models to use as baseline models. We started with the common models used for image classification: ResNet-18, ResNet-50, VGG-16, VGG-19, EfficientNet-B0, and GoogLeNet. Since there are available pre-trained weights for these models on PyTorch, we used those to do our initial experiments. Using training, validation, and test sets from our dataset, our group evaluated each model based on the trends of their accuracy and loss curves. The pre-trained models were trained using 10 epochs and had initial accuracies ranging from 60% to 80%. Based on these initial results, we chose the three models that had good accuracies and showed good training curves. The three models chosen were ResNet-50, EfficientNet-B0, and GoogLeNet. The performance of these models before any architecture modification and hyperparameter tuning can be seen in Table 1.

	ResNet-50	EfficientNet-B0	GoogLeNet
Validation Accuracy	81.62	63.64	68.88

Table 1: Base Validation Accuracy using Pre-trained Model Weights

After selecting these base models, our group worked on conducting transfer learning so that we can take these pre-trained models and adapt them to the specific dataset that our project is using. As a first step in transfer learning, we wanted to find the right way to freeze and unfreeze layers of the pre-trained models. Freezing layers means that weights for these layers are not updated during training which is helpful when the lower layers of a model are already performing well for our dataset so that we can then focus on updating the higher layers of the model. Unfreezing, on the other hand, allows weights to be updated during training, which is useful for tuning the lower layers of pre-trained models. Since unfreezing too many layers can cause overfitting because the weights may be too specialized for the training data, our group focused on unfreezing only the named layers from the lower blocks of the pre-trained models. This allowed the low level learned features to be fixed while the more abstract ones to be learned for our specific domain. After doing this, our group observed boosts in performance for both the training and validation accuracies.

The next step was to construct the fully connected head of each model which will receive the flattened convolutional output. After testing different dimensions, it appeared that the architecture that goes 2048, 512, 128, output in the fully connected layers worked really well for our dataset. The number of dimensions for each model were slightly different when it was flattened, but that general scheme showed to be very effective. During the construction of the fully connected head, our group evaluated the order of operations with batch normalization, ReLU activation, and dropout because it can have an impact on the performance and stability of a neural network during training. Since batch normalization is used to normalize inputs to a layer and improve gradient flow during backpropagation, we decided to apply batch normalization before ReLU to help maintain non-linearity and ensure that it has enough information to work with, because applying it after ReLU can limit its effectiveness. Dropout was then applied after the ReLU activation since it acts on the outputs of the activation function, and to avoid the risk of dropping out important information before ReLU.

After choosing which layers to freeze and unfreeze in the pre-trained models and constructing the fully connected heads, the next step in our experiments was to tune the hyperparameters in order to get more accurate models for all three which would have significantly improved performances. Our group focused on tuning four hyperparameters: learning rate, batch size, weight decay, and dropout rate. We chose to tune the learning rate because it determines the step size of the optimizer in updating the parameters during backpropagation, which controls how much the model adjusts its parameters. For the learning rate, we tuned using values of 1e-2, 1e-3, 5e-4, and 1e-4. The learning rate was selected based on the best validation accuracy for each of the three models as seen in Table 2.

Learning Rate	ResNet-50	EfficientNet-B0	GoogLeNet
1e-5	75.24	80.81	61.61
1e-4	83.45	82.42	79.39
5e-4	**88.08**	**82.62**	**83.60**

1e-3	85.20	78.18	81.81

Table 2: Tuning Learning Rate with Batch Size, Weight Decay, and Dropout Rate held constant

Once we determined the best learning rate for each pre-trained model, our group proceeded to try out different batch sizes. It is important to tune the batch size in order to find the right balance between convergence speed, memory usage, and generalization. We started with a batch size of 16 and tested 32, 48, and 64. As seen in Table 3, all the models generally performed better with a batch size of 32.

Batch Size	ResNet-50	EfficientNet-B0	GoogLeNet
16	88.08	82.62	81.81
32	**88.48**	**82.82**	**83.60**
48	**88.48**	82.62	81.61
64	**88.48**	82.63	81.61

Table 3: Tuning Batch Size with Learning Rate, Weight Decay, and Dropout Rate held constant

After selecting the batch size, we moved on to tuning the dropout rate, which is important in order to achieve the best trade-off between overfitting and underfitting. The dropout rate values we tried were 0.4, 0.2, 0.1, and 0 with the best ones for each model listed in Table 4.

Dropout Rate	ResNet-50	EfficientNet-B0	GoogLeNet
0.0	**88.48**	82.82	**83.60**
0.1	86.87	82.42	80.40
0.2	87.47	**84.04**	79.79
0.4	87.07	83.23	80.40

Table 4: Tuning Dropout Rate with Learning Rate, Batch Size, and Weight Decay held constant

Finally, our group also tuned weight decay, which helps prevent overfitting and improves the models' generalization performances. The weight decay options we looped through were 0, 1e-5, 1e-4, and 1e-3 with the best ones listed in Table 5.

Weight Decay	ResNet-50	EfficientNet-B0	GoogLeNet
0.0	**88.48**	84.04	**83.60**
1e-5	**88.48**	**85.05**	**83.60**
1e-4	87.88	82.22	81.41
1e-3	87.07	81.62	80.80

Table 5: Tuning Weight Decay with Learning Rate, Batch Size, and Dropout Rate held constant

After conducting hyperparameter tuning on all 3 models, it was interesting to note that the best versions of all the models had mostly the same set of hyperparameters as seen in Table 6. The similar final hyperparameters across the three models were a learning rate of 5e-4 and batch size of 32. For dropout rate and weight decay, there were only slight differences between the models. It is possible that this is due to the specific dataset and problem on-hand for this project.

	ResNet-50	EfficientNet-B0	GoogLeNet
Learning Rate	5e-4	5e-4	5e-4
Batch Size	32	32	32
Dropout	0.0	0.2	0.0
Weight Decay	0.0	1e-5	1e-5

Table 6: Final Hyperparameters for each Model

Following the determination of these optimal hyperparameters, we also explored various forms of data augmentation and found that incorporating random subtle changes to the hue and saturation, horizontal flips, and slight rotations were able to boost the validation accuracy by a couple percentage points. With these hyperparameters and this data augmentation, ResNet-50 had the best validation accuracy at 88% while EfficientNet-B0 and GoogLeNet had validation accuracies of 85% and 83% respectively. Figures 1, 2, and 3 show the training and loss curves for these 3 models.

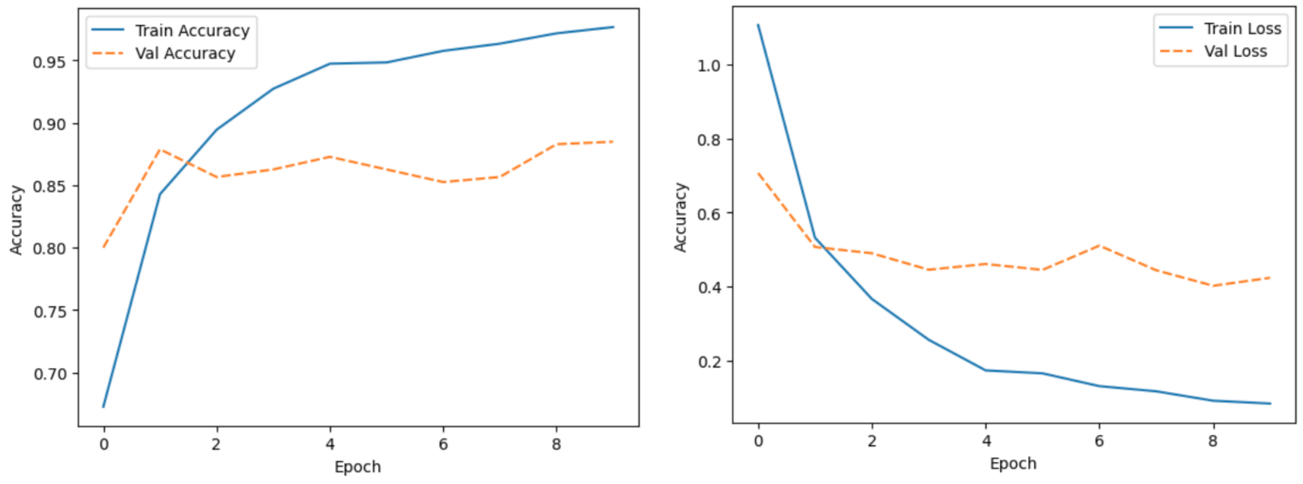


Figure 1: Training and Loss Curves for Best ResNet Model Configuration

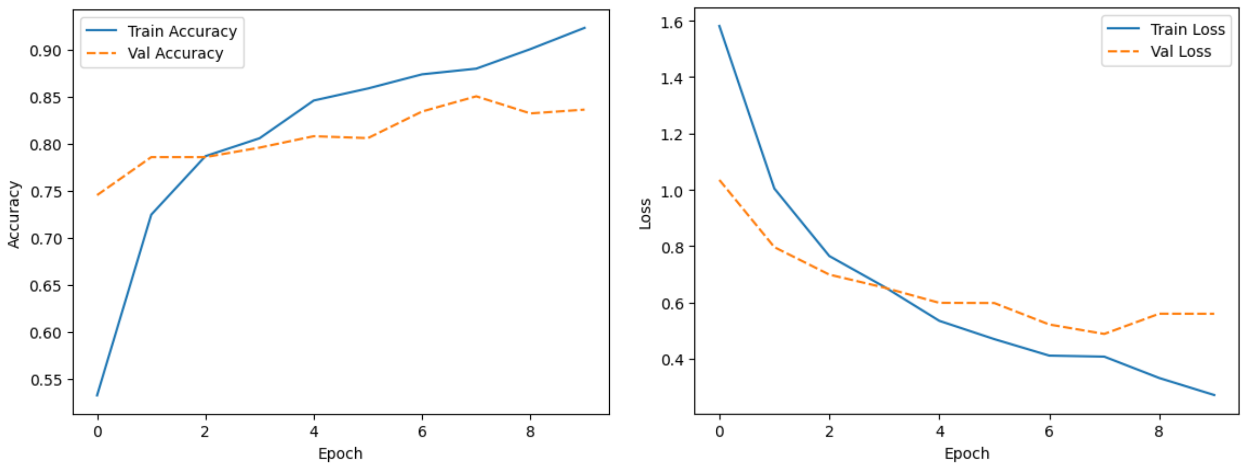


Figure 2: Training and Loss Curves for Best EfficientNet Model Configuration

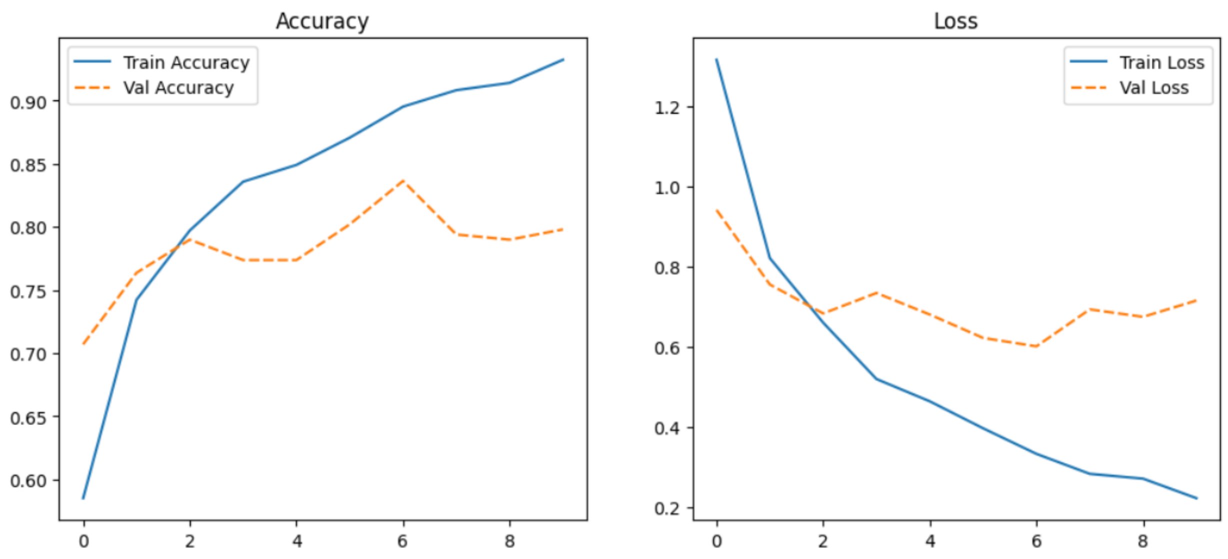


Figure 3: Training and Loss Curves for Best GoogLeNet Model Configuration

These models were then used to build the final ensemble model which produced a test accuracy of 88.5%. While this test accuracy is slightly below the highest validation accuracy achieved by a single model, it does represent a notable improvement of nearly 2% from the highest test accuracy achieved by any of the individual models and demonstrates the aggregate performance strength of our three unique model architectures. The results of this process demonstrate how powerful leveraging pre-trained models can be and how nuanced techniques such as fine tuning, hyperparameter tuning, data augmentation, and ensembling can meaningfully improve the performance of contemporary convolutional neural network models.

(It is also important to note, however, that these training processes are stochastic in nature, and, thus, may produce slightly different results on different runs. Therefore, we must recognize the fact that, while these may have been the optimal hyperparameters and results for our specific training runs, there may be other instances where slightly different configurations could produce even more optimal results. This does not impact the validity of our process or the significance of our final results, but it is a feature of deep learning that must be acknowledged appropriately.)

VI. Conclusions

As a result of this project, our group was able to successfully design, construct, and tune a model capable of accurately identifying Impressionist paintings' authorship based on the abstract stylistic elements associated with each specific artist. By harnessing the knowledge embedded in the pre-trained layers of existing and well known model architectures, we were able to retain the low-level feature extraction while adding enhanced discriminative power for our specific problem domain through fine tuning, hyperparameter tuning, data augmentation, and ensembling. The final performance result of our ensemble, an 88.5% test accuracy, is a significant accomplishment since this dataset is a multi-class classification problem with 10 possible outcomes. This performance is nearly 9 times better than random guessing and shows how robust neural networks can be in their high-level representations since we are not simply demonstrating the ability to learn the features associated with an object of a specific class but are truly displaying how these deep learning models have the capacity to develop high-dimensional, embedded representations of notions such as style and composition and can ascribe these elements to specific artists. This is an impressive feat these models are able to accomplish, and through this project, we have been able to meaningfully explore how specific techniques can assist in these models' impressive discriminative abilities.

While the process and results associated with our approach were generally quite successful, there are some limitations to our project. Our initial motivation was to identify the unique styles and techniques of each Impressionist artist, and, although our model has internally iterated to a parameterization that encapsulates this information, what we have developed is simply a robust classifier that does not implement object detection or saliency map, which are methodologies potentially more directly related to achieving our original goal. Additionally, this current model does not go beyond the inherent transparency and understandability limitations of deep learning models and still suffers from the "black-box" dilemma. Therefore, incorporating additional techniques that focus on the object level and depict the most salient regions for classification would be ideal for enhancing our understanding of Impressionist artists. Despite these minor limitations, however, our work has shown the versatility and remarkable potential of the convolutional neural networks and associated tuning techniques and has laid the foundation for future work in understanding the abstract relationship between Impressionist artists and the styles embodied in their paintings.

All code for this project can be found in our GitHub repository:
https://github.com/hyunsuk-ko/ImpressonistCNN

VII. References

[1] Banerjee, P. (2020). “Impressionist Classifier Data”. Kaggle. Retrieved Mar. 13, 2023. https://www.kaggle.com/datasets/delayedkarma/impressionist-classifier-data

[2] Banerji, S., Sinha, A. (2017). Painting Classification Using a Pre-trained Convolutional Neural Network. In: , *et al.* Computer Vision, Graphics, and Image Processing. ICVGIP 2016. Lecture Notes in Computer Science(), vol 10481. Springer, Cham. https://doi.org/10.1007/978-3-319-68124-5_15

[3] Rodriguez, C., Lech, M., Pirogova, E. (2018). "Classification of Style in Fine-Art Paintings Using Transfer Learning and Weighted Image Patches," 2018 12th International Conference on Signal Processing and Communication Systems (ICSPCS), Cairns, QLD, Australia, 2018, pp. 1-7, doi: https://doi.org/10.1109/ICSPCS.2018.8631731

[4] Sandoval, C., Pirogova, E., Lech, M. (2019). "Two-Stage Deep Learning Approach to the Classification of Fine-Art Paintings," in IEEE Access, vol. 7, pp. 41770-41781, 2019, doi: https://doi.org/10.1109/ACCESS.2019.2907986

[5] van Noord, N., Postma, E. (2017) “Learning scale-variant and scale-invariant features for deep image classification,” *Pattern Recognition*, vol. 61, pp. 583–592, Jan. 2017, doi: https://doi.org/10.1016/j.patcog.2016.06.005

[6] Wikipedia Contributors. (2019) “Impressionism,” *Wikipedia*, Mar. 09, 2019. https://en.wikipedia.org/wiki/Impressionism