Project Report - Introduction To Deep Learning – Vincent Van Gogh & Style Transfer

By: Amit Rabinovich,

Yonatan Ghefter,

Idan Kanat

1st Semester, 2024-2025

Introduction:

In this project, we intend to implement key topics from the course, divided into 2 parts: At 1st – we focused on **transfer learning & fine-tuning, using 2 pre trained CNN models (VGG-19 & AlexNet),** over a dataset of post impressionism paintings, with a particular emphasis on **Van Gogh's** works. This part highly focused on optimizing an accurate classifier capable of distinguishing between post-impressionist paintings and Van Gogh's works, while also extracting meaningful features from those paintings. This part also required a deep understanding of **HP Tuning,** for which we used the **Optuna** package, as well as **W & B, along with K-Fold Cross Validation, regularization & data augmentation techniques**.

Later on, in the 2nd part of the project, we used **Style Transfer**, leveraging the fine-tuned models from part 1 in order to generate beautiful, new paintings in the unique style of Van Gogh. This part's goal has been about exploring how deep learning models can recreate artistic styles while preserving content structure. We further used the fine-tuned CNN models from Part 1 to test how well they could classify these newly generated images as Van Gogh, testing whether the transferred style was strong enough to be recognized as a Van Gogh painting.

Throughout both parts, we intend to consistently analyze, optimize and compare both models (AlexNet & VGG-19), whether they classify Van Gogh paintings or generate stylistic pictures, gaining a better understanding of the performances of both Deep Learning architectures.

Part 1 – Transfer Learning on Van Gogh Paintings:

Post Impressionism Dataset:

The dataset used in this project is a subset of the WikiArt dataset, specifically the "Post-Impressionism" collection. As the name suggests, it consists of 4,670 Post-Impressionist paintings, extracted from the original WikiArt dataset, which reportedly contains 80,042 artworks spanning various artists and genres.

Among these 4,670 paintings, 1,004 are by Vincent van Gogh, making up a roughly 0.214 precent of the dataset.

To structure the dataset, we used a CSV file called "Classes", which provides information such as painting title, style, artist, image dimensions (width & height), the subset to which the painting belongs (train / test), and a binary label indicating whether the

painting was created by Van Gogh. This binary classification will serve as the target output for our neural network models. However, the "Classes" CSV only included around 4,550 paintings, meaning some 120 paintings from the "Post-Impressionism" dataset were excluded. To ensure consistency, we removed these missing entries before proceeding with our experiments.

CNN Models (AlexNet & VGG-19) – Overview:

The 2 relevant models throughout the entire project are specific CNN architectures – AlexNet & VGG-19. We shall briefly explain both of them:

AlexNet:

Introduced in 2012, AlexNet is one of the earliest deep CNN architectures, revolutionizing the world of Deep Learning & Computer Vision. Designed by Alex Krizhevsky, in collaboration with Ilya Sutskever and Geoffry Hinton, it gained international recognition after being the 1st of its kind to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) that year, significantly outperforming previous models with a remarkable error rate of 16.4%, which was substantially lower than the previous best error rate of 25.7%. Another of its key innovations has been its efficient usage of GPUs (Graphics Processing Units), dramatically speeding up training times and enabling it to scale effectively for large datasets.

AlexNet (as well as VGG-19) has been trained on a subset of the ImageNet dataset, which contains around 1.2 million 224*224 RGB images belonging to 1,000 different classes. This dataset has been (and still is) widely used for image classification, and more.

AlexNet consists of 8 layers – 5 Convolutional layers designed to extract meaningful features (with rising complexity) from the images. In between the each of the layers, the ReLu activation function is activated, which is widely used because it avoids problems with vanishing gradients unlike other activation functions. 3*3 Max Pooling (with a stride of 2) is also performed at times between the convolution layers to reduce the input's spatial dimensions each time. After the 5 Convolutional layers, there exist 3 Fully Connected layers followed by a softmax function to normalize the output to a probability distribution vector of 1,000 classes for ImageNet classification.

Initially, AlexNet applies a large 11x11 kernel on the input, followed by 3x3 max-pooling, exactly as stated before. Then, a smaller 5x5 kernel is used, followed by additional 3x3 max-pooling and three consecutive 3x3 convolutional layers. It is important to mention that the stride is always 1 and there's no padding throughout all 5

convolutional layers. Finally, the model concludes with the fully connected layers and the Softmax output.

VGG-19:

Trained for image classification, VGG-19 (Visual-Geometry Group) is another deep CNN architecture, introduced by Karen Simonyan and Andrew Zisserman in 2014. It gained its fame after winning the same ILSVR challenge (like AlexNet 2 years before) in that same year. Known as the deeper variant of the VGG models (deeper than its counterpart: the VGG-16), it has garnered considerable attention due to its relative simplicity and effectiveness. Like AlexNet, it was trained on the ImageNet dataset, taking 224×224 RGB images as input and producing a probability distribution over 1,000 classes.

In this network, there are 16 convolutional layers divided into 5 blocks - 2 convolutional layers in block 1, 3 in block 2, and 4 in each of the remaining blocks. The network's design was characterized by using small 3*3 kernels throughout all convolutional layers, which simplified the network's architecture and improved performance. Like AlexNet, each convolutional layer is followed by a ReLU activation function, preventing vanishing gradient issues. After each convolutional block, 2×2 max pooling (stride = 2) is applied, halving the spatial dimensions and progressively reducing the feature map size. To conclude, the last 3 layers are 3 Fully Connected layers whose input dimensions are 4,096 , followed by a SoftMax layer to produce a 1,000-class probability distribution.

Notably, as spatial dimensions decrease, the number of convolutional filters per layer increases, allowing for deeper hierarchical feature extraction. A key advantage of stacking 3×3 convolutional kernels sequentially is that it achieves the same receptive field as a single 7×7 kernel, but with significantly fewer parameters, leading to better efficiency. It also introduces more non-linearity through repeated ReLU activations, enhancing the network's ability to learn complex patterns.

While VGG-19 is conceptually simple, its' depth comes at a cost – it has notably more parameters (~143 million vs. AlexNet's ~60 million), making it computationally heavier but more powerful in feature extraction. Despite that, VGG-19's structured and deep architecture made it a benchmark for CNN-based models, serving as the foundation for later architectures. While AlexNet was the first to demonstrate CNN superiority and GPU acceleration, VGG-19 refined its concepts, achieving higher accuracy and better feature representation, making it a standard for transfer learning.

Note on Transformations: Since both VGG-19 and AlexNet were trained on the ImageNet dataset, the input paintings had to be adjusted accordingly. Each painting's dimensions were resized to match the required ImageNet input size of 224×224 pixels

(in RGB format), ensuring compatibility with both CNN models. Additionally, before being fed into the models, the pixel values were normalized using the mean and standard deviation of the ImageNet dataset: mean = [0.485, 0.456, 0.406] and s.d = [0.229, 0.224, 0.225]. This normalization aligns the input distribution with what the models were originally trained on, improving both models' performances.

Preprocessing and Optimizations

To optimize the training pipeline and improve data loading efficiency, we developed a preprocessing and serialization strategy that transforms raw datasets into a compact, reusable format. This process began with the implementation of the optimize_dataset function, which iterates through a given PyTorch-compatible dataset, converts each image and its corresponding label into NumPy arrays, and saves the entire dataset into a compressed .npz file using np.savez_compressed. This reduces disk space usage and accelerates future data access by eliminating the need for repeated preprocessing or transformation steps.

To support efficient integration into PyTorch workflows, we also created the NumPyDataset class. This custom dataset class is designed to directly load images and labels from the optimized .npz file and convert them into PyTorch tensors on-the-fly. It seamlessly integrates with standard PyTorch data loaders, enabling fast and memory-efficient data handling during training and evaluation.

This approach brings several advantages to the learning process:

- **Reduced preprocessing overhead**: All data transformations are done once and stored, avoiding repeated work during each training epoch.
- **Faster I/O**: Compressed binary storage significantly reduces read times compared to raw image formats or on-the-fly transformations.
- **Reproducibility and portability**: Saving the dataset in a consistent format ensures the same data is used across experiments, and makes it easier to share or deploy.
- **Seamless PyTorch integration**: By wrapping the .npz files in a PyTorch-style dataset class, we preserved compatibility with DataLoader and all PyTorch training utilities.

The impact of this optimization was clearly measurable: running a full pass over the dataset without training (i.e., one epoch of pure data loading) initially took **around 2 minutes** on the lab's GPUs. After converting the dataset using the optimize_dataset pipeline, that same operation took only **0.3 seconds**—an almost **400x speedup**. This optimization dramatically reduced training overhead and improved overall experiment

turnaround time (The procedure is fully described and implemented on preprocessing.py)

Data Augmentation Techniques:

To improve each model's generalization ability – to better classify unseen data, we artificially expanded the training dataset we had before by applying a variety of transformations over the paintings, allowing each model to be trained on a wider array of training data, helping it learn more robust features and improve its ability to distinguish between paintings during classification. This was all done in addition to the relevant transformations associated with the ImageNet inputs as mentioned before.

Augmentations:

We used only Offline augmentations, meaning that the augmentations were applied prior to the training process and were saved into a file and later merged with the original dataset to help improve generalization. We used several different types of augmentations to achieve that:

- "Flip Transform" :

```
flip_transform = transforms.Compose([
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(degrees=30),
    transforms.RandomPerspective(distortion_scale=0.3, p=0.5),
])
```

The flip_transform applies three augmentations: a vertical flip with 50% probability, a random rotation up to ±30 degrees, and a perspective distortion with 50% probability and a distortion scale of 0.3. The motivation behind this transformation is to introduce geometric variability during training, helping the model become more robust to orientation and perspective changes. This is especially useful in tasks like artwork classification, where the subject may appear in different angles, positions, or camera viewpoints.

- "Dropout Transform"

```
n_times = 25
dropout_transform = transforms.Compose([
    transforms.ToTensor(),
    *([transforms.RandomErasing(p=0.5, scale=(0.01, 0.01), ratio=(1, 1))]*n_times),
    transforms.Grayscale(),
])
```

The dropout_transform applies a series of augmentations including conversion to tensor, 25 applications of RandomErasing with a 50% chance each (targeting very small patches of fixed size), and finally conversion to grayscale. The purpose of this transformation is to simulate visual occlusions

and reduce reliance on specific local features by forcing the model to learn more robust, global patterns. Applying RandomErasing multiple times introduces scattered pixel-level noise, mimicking imperfections or damage in images, while converting to grayscale encourages the model to focus on texture and structure rather than color.

- "Affine Transformation"

```
affine_transform = transforms.Compose([
    transforms.RandomAffine(degrees=180, translate=(0.1, 0.1), scale=(0.8, 1.2), shear=(10,10)),
    transforms.RandomEqualize(p=0.5),
])
```

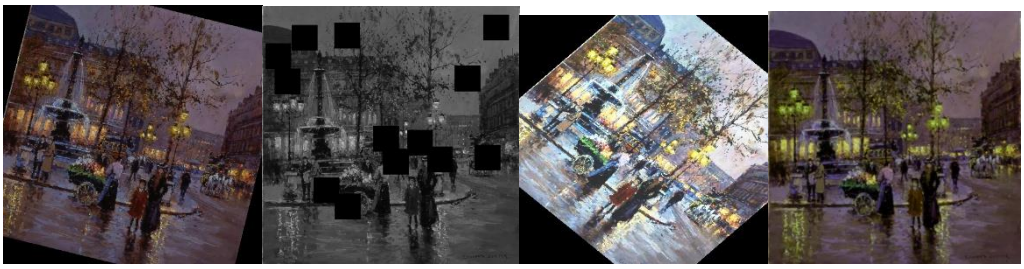
The affine_transform applies a RandomAffine transformation with rotation up to 180 degrees, slight translation, scaling, and shearing, followed by RandomEqualize with a 50% probability. This transformation introduces geometric distortions such as rotation, shifting, resizing, and skewing to simulate variations in how an image might be positioned or captured. The optional equalization adjusts image contrast by redistributing pixel intensities, mimicking differences in lighting or scanning quality. Together, these augmentations help the model become more robust to spatial and contrast variations commonly encountered in real-world data.

- "Blur Transformation"

```
blur_transform = transforms.Compose([
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.GaussianBlur(kernel_size=(3,3), sigma=(0.1, 2))
])
```

The blur_transform applies ColorJitter to randomly adjust brightness, contrast, saturation, and hue, followed by GaussianBlur with a kernel size of 3×3 and a variable sigma. This transformation simulates variations in lighting conditions and color balance, as well as image softness caused by camera focus or motion blur. The motivation behind it is to help the model become resilient to visual noise, color inconsistencies, and blurriness, enabling more stable performance under diverse real-world imaging conditions.

Show case of all the augmentations we used:



Flip Dropout

Affine Blur

original picture

To prevent data leakage and ensure the integrity of the evaluation process, all data augmentation was applied exclusively to a designated subset of the training data (731 samples). This subset was selected using stratified sampling to maintain the original class distribution. Furthermore, during cross-validation, we ensured that augmentations were strictly confined to the training split of each fold. Specifically, no augmented version of an image from the validation set in a given fold was included in the corresponding training set. This careful separation was essential to preserve the validity of performance metrics and prevent the model from inadvertently learning from validation data during training.

Training Process:

Transfer Learning – the key change to the models' architectures:

As emphasized throughout the course, training deep neural networks from scratch is both time-consuming and computationally expensive. Therefore, this is rarely done in practice. In our case, we were also instructed to fine-tune 2 pre-trained CNN models — AlexNet and VGG-19 — both originally trained on a 1,000-class ImageNet classification task. Since our task was a binary classification problem (predicting whether a painting was created by Van Gogh or not), we modified **only** the output layer, i.e: the last fully connected layer of the classifier, of each model to produce 2 output values instead of 1,000, representing probabilities of a certain painting / input to be either painted or not by Van Gogh. This is contrary to the other classical implementation, in which we throw away the entire classifier and train a new head from scratch. Both approaches keep the feature extractor (convolutional layers) and so, take advantage of the rich representations it learned throughout pre-training. However, the proposed approach, implemented in our project, retains the earlier FC layers too, which already capture more useful high-level representations learned from ImageNet. In this approach only a part of the FC layers – specifically the final one, will indeed be trained from scratch, while the others won't. We took into consideration that replacing only the last layer allows us to take advantage of these representations while adapting the output to our

specific task. In contrast, removing the entire classifier and training a new head from scratch could risk losing valuable learned features, reducing the benefit from the structure of the pre trained model, as well as consume more time which was a constraint.

Note that while a single output unit with a sigmoid activation could have also been suitable for this task, due to technical constraints in our implementation, the models were configured to output a two-dimensional vector. Just like sigmoid, performing a SoftMax activation at the output would produce a valid probability distribution over the two possible classes — 'Van Gogh' or 'Not Van Gogh'.

Fine-Tuning:

In addition to the slight modifications in the pre trained models' structures, we also unfreezed the final layers of the feature extractor of each model, as we were required to do in the project. This allowed us to fine-tune these layers' weights to better suit our binary classification task, helping to improve the models' performance. The earlier layers of the feature extractor, however, were kept frozen, as they already captured useful features during pre-training on ImageNet. The extent of which we unfreezed the networks' weights (i.e: number of layers to be unfrozen & fine-tuned) is a hyperparameter which will be discussed later on (in the HP-Tuning section).

Optimizer & Loss Function:

Since both models are fine-tuned on a binary classification problem, we chose the **Cross Entropy Loss** - a popular loss function for such problems, which penalizes large errors – deviations of the predicted probability, and the real output. The penalty is on a log scale, but since probabilities are fractions, it ensures a very large penalty for very large errors (e.g., incorrect predictions with high confidence). This helps the model to learn more effectively. In addition, to optimize the model's convergence during training, we chose **Adam** as our optimizer. As discussed throughout the course, Adam offers several advantages: it uses an **adaptive learning rate**, which allows it to adapt more efficiently to different parts of the optimization landscape and helps it converge faster, especially when gradients vary in magnitude. Adam is also **based on Gradient Descent**, ensuring it moves in the right direction to minimize the loss. Additionally, it **incorporates momentum**, which accumulates a weighted moving average of the gradients' means, helping to smooth updates and reduce oscillations, leading to more stable and faster convergence. The combination of these efficient and effective convergence properties allows Adam to navigate the optimization process efficiently, making it one of the most popular and commonly used optimizers for training deep learning models. For these reasons, we chose Adam to help optimize our models effectively and efficiently.

HP Tuning & K-Fold Cross Validation:

To transfer-learn & fine-tune both models, they had to be trained, and for that, an optimal combination of hyperparameters had to be found. To find them, we used the **Optuna** package as we saw throughout the course, which would create a **study** for each CNN model, which consists of multiple **trials** – each trial is analogous to a certain combination of hyperparameters. Unlike model parameters, hyperparameters are not learned during training but are chosen manually or through optimization techniques like this one. In the implementation we saw in class, in each trial, we trained a model in the standard manner: we iterated over the training dataset (the iterations are called epochs), measured performance metrics (such as training loss, accuracy, AUC, etc..), and tracked them using the **Weights & Biases** (Wandb) API. In class, we also wanted to monitor the model performance over "unseen" validation data, to better assess the model's generalization ability. We had to **split the data into equally sized batches** – whose size was determined by us as a hyperparameter, which will be discussed later & also helped speed up & stabilize the training process.

This time, however, we introduced a twist: In each trial, using its corresponding hyperparameters, instead of using a simple train/validation split (the Holdout Method) we first extracted 20% of the data from the "Classes" dataset designated as "train" to create a new validation set (common practice, leaves a large enough validation set to estimate models' generalization ability without compromising on the amount of data for training. This is done as a formality too for the sake of model selection), and then performed **K-Fold Cross Validation** over the rest of the training data. This technique is more robust because it ensures that every data point is used both for training and validation across different folds. However, this method is more computationally expensive - since this time, a model has to be trained K times on K-1 folds each time, and metrics are measured on the remaining fold for validation, and we average the metrics over the folds to get an unbiased estimate of the model's performance on "unseen" validation data. Since K-Fold Cross Validation can prove to be computationally expensive, especially for training deep CNN models, we decided to stay with a low **K = 4** – so that the models won't be fine-tuned as much, reducing the computational complexity, and so that each fold will have a relatively large amount of data. We were also advised to do so in the project. The hyperparameters we experimented with have been the:

- **Learning Rate = LR:** Kept low in the range of [1e-5, 1e-3], so that the change from fine-tuning the model's parameters will be gradual & controlled, ensuring a more precise adjustment to the pre-trained weights without overfitting.

- **Number of Epochs:** Set as an **integer between 10 & 30**. We saw that the models' performances were good enough & stabilized in that range of epochs, therefore we decided to stay with that range.
- **Weight Decay:** A regularization hyperparameter which was kept low in the range of [1e-6, 1e-4], the larger the Weight Decay – the more stringent the regularization, and vice versa. Concluding our experimentations, a low weight decay in that range strikes a balance, allowing for effective regularization without overly restricting the model's ability to learn complex patterns.
- **Number of Layers to Fine-Tune:** Set as an **integer between 0 & 3** to give the model flexibility in how many layers would be most optimal for fine-tuning. We included 0 as a possibility as in some cases the model could perform best even without any fine-tuning. The range was limited to lower values due to computational constraints and time limitations to tweak this many parameters, even though after the data augmentations left us with no shortage of data. Fine-tuning the later layers of the feature extractor specifically, has been an especially important consideration, since those layers are generally responsible for detecting more specific features which can be highly relevant to our Post Impressionism dataset.
- **Batch Size:** Set as a 2 to the power of various numbers – such as 32 = 2^5, 64 = 2^6, 128 = 2^7 & 256 = 2^8. This is a convention in the Deep Learning world, seen throughout the course as well, which is done to stabilize & speed the training process by lowering memory requirements which results in speeding up processing of the data and gradient updating.
- **Patience & Early Stopping:** While optimizing both CNN models, we used the Early Stopping regularization technique in order to prevent overfitting. The technique **stops training the model if the Validation AUC doesn't improve** (i.e: doesn't strictly increase) **over a defined number of consecutive epochs**. That number is defined as the patience, which we set to **5**. It's crucial that the 5 = patience < Number of Epochs (set by us between 10 & 30), since otherwise, the model can't ever satisfy the early stopping condition even if it's necessary and its validation AUC / performance doesn't improve. We initially considered to bound the patience as an integer between 5 & 15 but since it can vastly increase the HP optimization space and running trials is computationally expensive & time consuming, we eventually decided to keep it as a "reasonable" value which isn't ever larger than the number of epochs. Furthermore, a patience value of 5 allows the model sufficient opportunities to improve its validation AUC before early stopping is triggered, even with a lower number of epochs (e.g: 10 epochs). After some trial & error we realized this specific patience value improves performance on both training, validation & test data.

A few other Notes about the Procedure:

- We set the **Number of Trials** to **15** due to the substantial computational resources and time required to perform K-Fold Cross-Validation across numerous trials. Given our constraints on both time and computational capacity, running an extensive hyperparameter tuning process was impractical. And so, following the recommendations provided in the project, we limited the number of trials to 15.
- Similar to the implementation seen throughout the course, we logged & tracked all metrics into Weights & Biases - **W & B** – measuring the relevant model performance metrics. For each of the 15 trials, across all 4 folds, and for each CNN model, we recorded key metrics over the trial's defined number of epochs.
  The metrics:
  Training: Loss & Accuracy.
  Validation: Loss, AUC, Accuracy, Recall, Precision, F1-Score, Specificity.
  We also measured the average validation AUC over all its corresponding 4 folds.
- Performing standard K-Fold CV was insufficient, as it did not guarantee the same target class balance across folds. To address this, we used **Stratified K-Fold CV** to ensure that each fold maintained the same class distribution as the overall dataset.
- **The best trial would be selected based on the maximization of the Average Validation AUC** (across K = 4 folds). Practically speaking, during each trial, the Average Validation AUC over the folds was tracked, and the trial with the largest value was chosen for the next steps. It was essential to choose a trial based on optimizing an **average** metric on the **Validation** sets across the folds, since it was the very reason we performed K-Fold CV: to get an unbiased estimate of the model performance on a validation set, from the average validation metrics. This should test the model's ability to generalize well. The specific metric we chose to look at was the **AUC**. We chose it over other metrics like Loss or Accuracy because it provides a more comprehensive measure of model performance, especially in the context of class imbalance. Unlike Accuracy, AUC is robust to imbalanced data and evaluates the model's ability to rank predictions across all thresholds, not just classify them correctly. This makes the Average Validation AUC a better indicator of the model's generalization ability and its capacity to distinguish between classes, which was critical for our model evaluation.
- **After finding the best trial** (i.e: best combination of hyperparameters) for each CNN model, **we trained each model on the larger training dataset** - 80% of the original training dataset from the "Classes" file as explained before. This ensured that the model wasn't trained on (K-1)/K = ¾ of the smaller training dataset in the K-Fold CV. Given that the best hyperparameters were already found, we could exploit it to find the optimal parameters. This time, we used a

larger training dataset for that procedure for several reasons: we had the flexibility to do so without introducing any explicit issues. Besides, having more training data could help the model learn better, so if we can use more training data, it's better to do so. After training, we evaluated the models' performances over the unseen validation data, and then again tested their performances on the truly unseen test data.

Evaluating & Comparing the Models + Key Insights:

K-Fold CV & HP Tuning:

In the graphs (in the appendices), we can see each trial's average performance metrics over K = 4 folds, which were logged into W&B. Below is an analysis of the results:

AlexNet:

- Metrics: Many if not most trials achieved a **near-perfect average validation accuracy, precision, recall & AUC of around 1**, with the worst performing trial maintaining ~0.978 across these metrics. Specificity was also consistently high - ~0.99 at worst. Nearly all trials achieved a validation loss of approximately 0 with a clear outlier with a validation loss of 0.24. This indicates that even on "unseen" Validation sets, the AlexNet model performed amazingly well considering all aspects / metrics as it distinguished between "unseen" Post Impressionist paintings.
- Batch Size: was varied – many trials got values of 64 & 128, and some got other values such as 32 & 256.
- Number of Layers to Fine-Tune: No clear winner: some well performing trials (w.r.t to validation loss) fine-tuned 0-2 layers, others achieved similar performance by fine-tuning 3–4 layers. This suggests that the specific number of unfrozen layers in the feature extractor has a negligible impact over model performance.
- Number of Epochs: There appears a concentration of trials with values ranged 25-30, but for other trials it appeared scattered across 10-25. This dispersion suggests that although longer training may be beneficial, performance was not strictly tied to a specific epoch count.
- Learning Rate & Weight Decay: Most trials (including the purples – "good ones") appeared to have a very small LR & Weight Decay of around 0 for both, while a sizable portion of them had a slightly higher LR & Weight Decay, which were still very small (0.0008 at most for the LR & 0.0001 at most for the Weight Decay). This indicates that our model requires little regularization & to take baby steps (LR) towards optimal convergence.

VGG-19:

- Metrics: Many if not most trials achieved **near-perfect average validation accuracy, precision, recall & AUC of around 1**, with the worst performing trial having each average metric at ~0.95 at worst, which is lower than AlexNet's "lowest bar" of ~0.978. Same trend with the specificity, however, the worst performing trial had a specificity value of ~0.98 at worst, which indeed indicates very good performance across all trials, which is also lower than AlexNet's "lowest bar" of ~0.99. Virtually all trials achieved a validation loss of approximately 0 with a clear outlier with a validation loss of 0.5, also worse than AlexNet's worst trial in that regard (with the Validation Loss of around 0.24). This indicates that even on "unseen" Validation sets, the VGG-19 model performed very well considering all aspects / metrics as it distinguished between "unseen" Post Impressionist paintings. However, in general, VGG-19's trials appear to have some noticeably worse performance than the corresponding AlexNet trials. The difference between them is still negligible but it's notable across all metrics.
- Batch Size: was highly concentrated at 256, with some other worse performing trials having a lower value of either 32, 64, 128..
- Number of Layers to Fine-Tune: There was no clear optimal number of layers to fine-tune, as the distribution of trials across different values appeared uniform. Some high-performing trials fine-tuned as few as 0–2 layers, while others fine-tuned 3–4. The even distribution of results across different settings suggests that the number of unfrozen layers had minimal effect on final performance.
- Number of Epochs: There appears a concentration of trials with values ranged 14-24, but for other trials it appeared scattered across 10-30. As with AlexNet, this indicates no strong dependency on a specific number of epochs.
- Learning Rate & Weight Decay: Most trials (including the purples – "good ones") appeared to have a very small LR & Weight Decay of around 0 for both, while a sizable portion of them had a slightly higher LR & Weight Decay, which were still very small (0.0008 at most for both the LR & Weight Decay). Like AlexNet, those values indicate that our model requires little regularization & to take baby steps (LR) towards optimal convergence.

To conclude, the vast majority of both AlexNet & VGG-19's trials appeared to perform exceptionally well on average on the Validation set, indicating that their performances on "unseen" data is remarkable. However, generally, AlexNet's trials appeared to have better metrics – higher AUC, Accuracy, lower loss, etc.. (and therefore, perform better) than VGG-19's, on all regards, indicating better average performance on "unseen" data.

Final Training + Testing the Models:

Set	Model	AlexNet	VGG-19
Validation	AUC	0.9845	0.9999
	Loss	0.1305	0.05
	Accuracy	0.9343	0.9357
	Precision	0.8021	0.8
	Recall	0.9317	0.9441
	Specificity	0.9351	0.9333
	F1 Score	0.8621	0.8661
Test	AUC	0.9845	0.9875
	Loss	0.1705	0.0902
	Accuracy	0.8986	0.9242
	Precision	0.7103	0.7745
	Recall	0.9086	0.9239
	Specificity	0.8957	0.9243
	F1 Score	0.7973	0.8426

We can see both fine-tuned CNN models provided good results even on unseen data. Both the fine-tuned AlexNet & VGG-19 performed impressively well on both the validation & test sets, with remarkable metrics over the validation set – large AUC (>0.98 for both models), Accuracy (~0.93 for both), Precision (~0.80 for both), Recall, Specificity (~0.93 for both) & F1 Score (~0.86 for both). The difference between the models were very minor on the validation set as seen in the table. VGG-19's Validation AUC is still larger than its corresponding AlexNet's Validation AUC, but only by 0.01. The validation loss is rather much smaller looking at VGG-19 – 0.05, compared to AlexNet's 0.13. The confusion matrices for the validation set confirm this comparably good performance, showing that AlexNet correctly classified the majority of paintings - 533 non-Van Gogh and 150 Van Gogh paintings (with 37 and 11 misclassifications respectively), while VGG-19 correctly identified 532 non-Van Gogh and 152 Van Gogh works (with 38 and 9 misclassifications). This striking similarity suggests both architectures reach comparable performance plateaus on the validation set.

In the Test Set, things differ. Both models still perform well, providing Test AUCs of ~0.98 and accuracies ranging at 0.9 for example, as well as high recall & specificity, proving that both capture well Van Gogh's intricate style, and distinguishing between Van Gogh & other paintings. We can see that each model performs slightly worse as we look at a test metric of such model and compare it to its corresponding validation metric. For example, the loss increased from ~0.13 to ~0.17 for AlexNet and from ~0.05 to ~0.09 to VGG-19 for example. However, looking at all test metrics, VGG-19 performs better than AlexNet in **all of them**, by some wide gaps, indicating better model performance on such unseen data. The test confusion matrices show AlexNet correctly classified most of the paintings - 627 non-Van Gogh and 179 Van Gogh paintings while misclassifying 73 and 18 samples. In contrast, VGG19 correctly identified most of the paintings as well, and even more - 647 non-Van Gogh and 182 Van Gogh works, with only 53 and 15 misclassifications. Both models experience performance degradation when transitioning to truly unseen data, but AlexNet's more significant precision drop (from 0.8021 to 0.7103) suggests VGG19's far deeper architecture provides better robustness. Comparing VGG-19 & AlexNet w.r.t to the confusion matrices, we can see that as we compare them upon the same set (test & validation), we can see that VGG-19 has a consistently larger amount of True Positives & True Negatives than AlexNet, while maintaining a consistently lower amount of corresponding misclassifications (False Negatives & False Positives), but these gaps are not as wide – they're up to single paintings. The consistent pattern of higher false negatives across both models indicates they tend to be overly conservative in classifying authentic Van Gogh works, a critical consideration for practical art authentication applications.

Analysis of Examples:

Disclaimer: We provided here potential explanations to the way both models predict images & assign them as Van Gogh or not. However, it's crucial to keep in mind that both models are deep CNN architectures, infamous for the difficulty of interpreting & understanding the reasons they classify images as such. Therefore, it's very important to note that these explanations are deep within the realm of speculation and we can't be sure.

AlexNet:

True Positives:

In these cases, we can speculate that Van Gogh's distinctive color palette, which are typically saturated & vibrant, as well as his thick swirling brushtrokes, could've been extracted by AlexNet so that it'd classify them as Van Gogh's.

True Negatives:

In both cases which aren't Van Gogh paintings, a potential explanation could be that either the paintings' colors weren't as typically saturated as Van Gogh (as seen on the left) / the portraits seemed too abstract to be associated with Van Gogh, who typically painted landscapes. The painting to the right has very strong & warm colors as well as thick swirling brushtrokes which is very typical of Van Gogh. However, AlexNet succeeded classifying it correctly as Not Van Gogh.

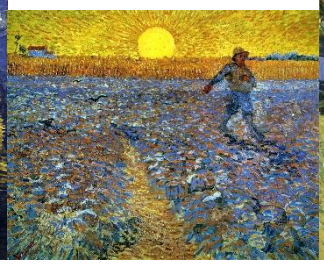
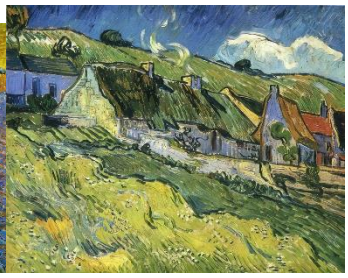
False Negatives:

False Positives:

Surprisingly, AlexNet classified this painting as Van Gogh, possibly due to its apparent brushtrokes of the river which are reminiscent of Starry Night, even though the colors appear less warm & saturated than typical Van Gogh paintings

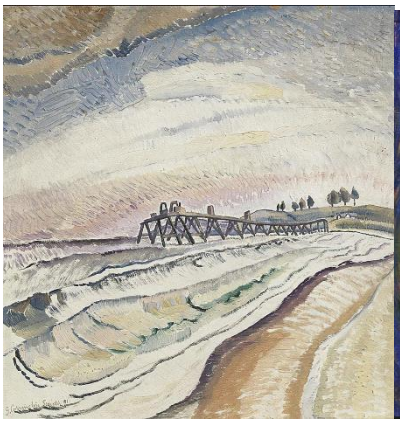
VGG-19:

True Positives:

In these cases, we can speculate that Van Gogh's distinctive color palette, which are typically saturated & vibrant, as well as his thick swirling brushtrokes, could've been extracted by VGG-19 so that it'd classify them as Van Gogh's.

True Negatives:




In both cases which aren't Van Gogh paintings, a potential explanation could be that either the paintings' colors weren't as typically saturated as Van Gogh ones (as seen on the left) / the portraits seemed too abstract to be associated with Van Gogh, who typically painted landscapes. The painting to the right has very strong & warm colors as well as thick swirling brushtrokes which is very typical of Van Gogh. And so, VGG-19 succeeded classifying it correctly as Not Van Gogh.

False Negatives:

Since both models produced very few misclassifications, especially False Negatives (as the Recall shows on both models), it was particularly hard to understand what made them get the classification wrong, especially with such few mistakes and no clear pattern which could help explain it. This is exacerbated by the fact that despite the accuracy, precision & recall gaps between 2 the models it's still reflected in few paintings which showed disagreement between them in our data, so it could be hard to understand on what AlexNet is more focused at while extracting features, compared to VGG-19. Their performances appeared to have some similarity with a lead to VGG-19 across all test metrics, indicating better performance.

False Positives:

Surprisingly, VGG-19 classified this painting as Van Gogh, possibly due to its apparent brushtrokes of the river which are reminiscent of Starry Night, even though the colors

appear less warm & saturated than typical Van Gogh paintings. It also agreed with AlexNet on this mistaken prediction.

Part 2 – Style Transfer:

Style Transfer Explained:

Style Transfer is a process which blends the content of an image (the Content Image) with the artistic style of another image (the Style Image) to generate a new image which inherits both the content of the 1st image as well as the style of the other image. It does so by using certain layers from either pre-trained or fine-tuned CNN models, to extract 2 distinct types of features: Style Features – which represent the image's style – brushstrokes, colors, texture, stylistic shapes etc. from the style image, and Content Features – which capture the image's content - overall structure, shapes, objects, dimensions etc. from the content image. Style Features are typically the features extracted from the earlier layers of the CNN models, while Content Features are typically extracted in later, intermediate layers of the network which are not too deep either.

The target image to be generated by the end of this process begins as either noise or a copy of the content image and is iteratively optimized to minimize a total loss function - a weighted sum of content and style losses so that it increasingly resembles the content image in structure and the style image in appearance. In our project, we applied this process using 20 original content images and produced at least 5 stylized images per pair of content image and style painting, combining the images to produce stylized outputs that reflect Van Gogh's distinctive artistic style. By leveraging both AlexNet and VGG-19, we explored how different architectures and layer selections influence the effectiveness and visual quality of style transfer. Over all we used 25 style images and 17 of them were painted by Van Gogh

Steps:

1. Choosing Style & Content Images
2. Choosing pre-trained / fine-tuned CNN models for feature extraction: before beginning the style transferring we set the models to evaluation mode and freeze all layers of the convolutional part of the model (the classifier part of the models is not used). Since the purpose of style transferring is essentially training our input to fit the loss, training the model during the process is not needed and can impact negatively on the results by changing the useful qualities learned by the model during the process
3. Processing the images: all of the images used in the algorithm need to be processed and go through a transformation that sends the images to the

distribution space of the data on which the original model was trained on, that is because it's weights are tuned to that distribution's characteristics and inputting data from a different space will affect negatively on the model's evaluation of the images.

4. Selecting layers for feature extraction – style & content + weighting them: Content Features are the activation maps extracted from a single intermediate layer, as these layers best capture the semantic structure and spatial layout of the content image without being overly abstract or too localized. Style Features are extracted from multiple shallower. These early layers retain more fine-grained textures, color patterns, and brushstroke-level details, which are essential for capturing style at multiple levels of abstraction. Each selected style layer is normalized to ensure that no single layer dominates the style representation due to having a higher raw loss value. Normalization balances the contribution of each layer and leads to more visually coherent and artistically pleasing results. The extracted features are used to compute the Content Loss and Style Loss, which are weighted using coefficients α and β, respectively (for example: $\alpha = 1$, $\beta = 1000$) to control their influence on the final output.

5. Optimizing the weighted loss (Style Loss & Content Loss) and updating the target picture accordingly: The algorithm starts with a target image, initialized either as random noise or a copy of the content image. This target image is the one that gets optimized. The Content Loss is calculated as a Sum of Squared Error (SSE) between the feature map of the content image and that of the target image at the selected content layer:

   This ensures the final image maintains the structure and layout of the content image.

   The Style Loss is based on MSE, but it measures the difference between the **Gram matrices** of the style image and the target image across multiple layers & their feature maps: The Gram matrix of a feature map is a matrix of inner products between vectorized filter responses. It captures spatially invariant correlations—i.e., the co-occurrence statistics of features (like brushstrokes or textures) without being sensitive to their exact location in the image. By matching Gram matrices, we ensure that the target image replicates the global style patterns of the style image (e.g., color distributions, textures, artistic strokes), not just pixel-level details. Each style loss layer is normalized (typically by the square of the number of features and size of the layer) to prevent any single layer from disproportionately impacting the total style loss. The Total Loss is the sum of weighted Content and Style losses.

Using backpropagation, the gradients of this loss are computed with respect to the pixels of the target image. The image is then updated using the Adam optimizer, which adaptively adjusts learning rates for faster and smoother convergence. This iterative optimization runs over several epochs, gradually updating the target image so it increasingly resembles the structure of the content image and the style of the style image.

6. Presenting and saving the results: the final step in the process although it might seem miniscule includes an important concept to be mindful of, after obtaining the results of the style transfer optimization process our result is not yet ready for viewing. Considering that before evaluating the images through the model they needed to go through a processing function to align their pixels with the distribution of data on which the models were trained, the stylized images at the end of the optimization process remain within that distribution space. Hence to save and view the generated images in a way which can be understood and appreciated by us the results need to be "deprocessed" by that we mean, go through the inverse of the processing function to go back to the image's original distribution space.

```python
def style_transfer_multi_style(model, style_img_paths, style_weights, content_img_path, layers, style_layer_weights, content_layer, content_weight=1, st
    model = model.features  # Gives us access to the layers of features

    # Prepare model for evaluation, disabling gradient computation
    model.to(device).eval()
    for param in model.parameters():
        param.requires_grad_(False)

    # Load and preprocess the content image
    content = load_image(content_img_path, shape).to(device)
    content_features = get_features(content, model, layers)

    # Load and preprocess the style images
    style_features_list = []
    for style_img_path in style_img_paths:
        style = load_image(style_img_path, shape).to(device)
        style_features = get_features(style, model, layers)
        style_features_list.append(style_features)

    # Combine style features by weighted averaging
    combined_style_features = {}
    for layer in layers.values():
        combined_style_features[layer] = torch.zeros_like(style_features_list[0][layer])
        for i, style_features in enumerate(style_features_list):
            combined_style_features[layer] += style_weights[i] * style_features[layer]
        combined_style_features[layer] /= sum(style_weights)

    target = content.clone().requires_grad_(True).to(device)
    style_grams = {layer: gram_matrix(combined_style_features[layer]) for layer in combined_style_features}
    optimizer = optim.Adam([target], lr=0.003)

    # Initialize list to track losses
    losses = []
    # Style transfer loop
    with tqdm.tqdm(total=num_steps, desc="Style Transfer Progress") as pbar:
        # Style transfer loop
        for ii in range(1, num_steps + 1):
            # Extract features from target image
            target_features = get_features(target, model, layers)
            # Compute content loss
            content_loss = torch.mean((target_features[content_layer] - content_features[content_layer])**2)

            # Compute style loss by comparing Gram matrices for each layer
            style_loss = 0
            for layer in style_layer_weights:
                target_feature = target_features[layer]
                target_gram = gram_matrix(target_feature)
                _, d, h, w = target_feature.shape
                style_gram = style_grams[layer]
                layer_style_loss = style_layer_weights[layer] * torch.mean((target_gram - style_gram)**2)
                style_loss += layer_style_loss / (d * h * w)

            # Calculate total loss and update target image
            total_loss = content_weight * content_loss + style_weight * style_loss
            optimizer.zero_grad()
            total_loss.backward()
            optimizer.step()

            pbar.update(1)

            # Track the loss
            if ii % 1000 == 0:
                losses.append(total_loss.item())
                print(f"Step {ii}, Total loss: {total_loss.item()}")
                if show_progress:
                    # Deprocess and display the intermediate result
                    intermediate_result = deprocess(target)
                    plt.imshow(intermediate_result)
                    plt.title(f"Step {ii}")
                    plt.axis('off')
                    plt.show()

    return target
```

This function was created to be a generic function that is fit to use with standard model architectures like AlexNet and VGG-19 (compared to inception's model unorthodox architecture). The function based on the function provided to us in the instructional presentation, takes in the same arguments such as style weight, content weight and number of optimizer steps to perform and adds more arguments for better more specific control over the process such as which layers to use, individual style layer weights, which of the chosen layers will be used for the content loss, also this function adds the ability to combine the styles of more than one style image which also requires a set of weights between the style images.

The first part of the function after the input arguments is steps No.2 and No.3 as described above taking only the convolutional part of the model, freezing it, calling utility functions for loading and processing the images and then extracting the features of each image according to step No.4. afterwards comes a different step not mentioned in the algorithm above in which all of the selected style images' extracted features are combined in a weighted average before calculating each layer's gram matrix.

The next step is the optimization loop as explained in step No.5, the content's MSE loss is calculated first, then the gram matrixes of the activation of each specified layer of the model are calculated before calculating it's MSE loss compared to the combined style images' gram matrixes, the results are normalized and added according to their weights And the content loss is also added. Then the standard optimization process follows.

A snippet is added for better process tracking such as loss printing and intermediate result printing. In the end the result of the optimization is given as an output

```
def style_transfer_multi_wrapper(model, style_img_paths, style_weights_list, content_img_path, layers, style_layer_weights_list, content_layer, content_w
    results = []
    folder_name = generate_folder_name(style_img_paths, content_img_path, model_name)
    folder_path = create_directory(output_dir, folder_name)

    for i, (style_weights, style_layer_weights, content_weight, style_weight) in enumerate(zip(style_weights_list, style_layer_weights_list, content_weig
        # Call the multi-style transfer function
        target = style_transfer_multi_style(model, style_img_paths, style_weights, content_img_path, layers, style_layer_weights, content_layer, content_
        img = deprocess(target)
        results.append(img)

        # Save the result immediately
        filename = generate_filename(style_weights, style_layer_weights, content_weight, style_weight, layers, content_layer, i)
        output_path = os.path.join(folder_path, filename)
        img.save(output_path)

        # Copy original style and content images to the directory
        for style_img_path in style_img_paths:
            shutil.copy(style_img_path, folder_path)
        shutil.copy(content_img_path, folder_path)

    return results
```

This is a wrapper function for the style transfer function above, it can take as an input multiple sets of individual layer weights, style images weights, style and content weights in addition to other inputs. The purpose of this function is to make experimentation with style transfer easier and more streamlined. In this function step No.6 is implemented in order for us to be able to view the results as we expect.

In addition the function calls for utility functions we made for saving the resulting images in a convenient manner.

Note that an argument to both functions is the shape of the results this is because the nature of CNNs allows us to do the style transfer process on images of every size (the results are affected by target size of the images).

Importance of Normalization in the Style Loss:

Normalization is essential in style loss calculation during style transfer as it establishes scale-invariance across feature maps with varying dimensionality. Without normalization, the optimization process would be disproportionately influenced by feature maps with larger dimensions, creating an implicit weighting bias that misrepresents the true stylistic elements. This normalization becomes particularly critical when considering the different contributions from various network layers, as each layer captures distinct stylistic attributes—lower layers represent finer textures and brush strokes, while deeper layers encode more abstract patterns and compositional elements. The style loss calculation relies on Gram matrices, which are essentially feature correlation matrices that capture the statistical relationships between different feature map activations, revealing how style features co-occur within an image. The dimensions of these Gram matrices vary significantly across network layers, and without proper normalization by $1/(4 \times N_l^2 \times M_l^2)$, where N_l represents the number of feature channels and M_l^2 the spatial dimensions, larger Gram matrices would dominate the loss function. By normalizing each feature representation by its total number of elements, the style loss becomes dimension-agnostic, enabling consistent comparison between feature representations at different network depths and resolutions, and ensuring a balanced transfer of both fine-grained textural elements and broader compositional patterns from the style image to the content image during the optimization process.

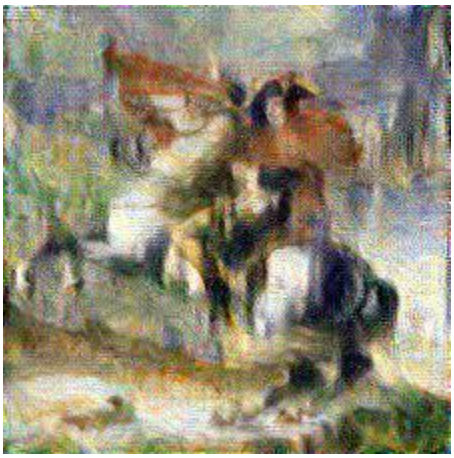
Analysis of Examples:

Cases where both models classify as Van Gogh:

It appears that both models agree on predicting certain Style Transfer images as Van Gogh when it comes to a combination of factors: natural, still landscapes, as well as his

distinctive color palette which is highly saturated & vibrant. He also has some distinct, thick swirling brushstrokes which give his paintings energy, dynamism & emotion.

Cases where both models classify as not Van Gogh:



We can see that both models agreed that this style transfer Napoleon image shouldn't be associated with Van Gogh. We didn't find as much style transfer images in which both models agreed to negatively classify a style transfer image.

Cases where VGG-19 classifies as Van Gogh while AlexNet doesn't:

In both cases, the content is simply a picture of a dog which isn't outside so it's not a typical landscape picture with a diverse saturated color palette Van Gogh typically uses, so that could be a potential reason that AlexNet classified both style transfer images as not Van Gogh. However, we can see thick swirling brushtrokes which resemble Van Gogh's style. That could be a reason why VGG-19 classifies those Style Transfer images as Van Gogh's while AlexNet, in which in both cases colors aren't as distinctive to Van Gogh, disagrees.

Cases where AlexNet classifies as Van Gogh while VGG-19 doesn't:

In both cases, a potential explanation to the disagreement between both models could be that the colors are very saturated & vibrant, which is typical to Van Gogh's style, and so, AlexNet classified them as Van Gogh. However, the brushtrokes in both images didn't have the unique thick swirling signature so common in Van Gogh paintings, and so, VGG-19 apparently could recognize the style transfer paintings as Van Gogh's. We can speculate that the brushtrokes of the right painting appear less as clear as the brushtrokes in the similar right painting where both models agreed to classify the image as Van Gogh's, and that's why VGG-19 "changed its mind" and classified the image differently although the style & content were practically the same, but the weights given to the layers differed, even though AlexNet remained consistent & classified both images as Van Gogh's. The same trend could be argued w.r.t to the image to the left. The style & content remained the same but the weights to each layer differed and so that could explain as well why this time VGG-19 changed its mind while AlexNet didn't. However, we have to keep in mind that both models are deep CNN architectures, infamous for the difficulty of interpreting & understanding the reasons they classify images as such. Therefore, it's very important to note that these explanations are deep within the realm of speculation and we can't be sure.

Layer Selection:

AlexNet:

While using AlexNet in the style transfer process, we utilized all of the model's available convolutional layers which are layers 0,3,6,8,10. Given that AlexNet is a relatively shallow network with only five convolutional layers, employing all layers is essential for capturing the full range of features in the images. The earlier layers were used for style loss calculation, while the last layer was designated for content loss. This choice is based on the nature of the network's receptive fields: earlier layers capture fine-grained textures and colors, whereas deeper layers, with larger receptive fields, capture broader shapes and structural elements of the style painting. Therefore, it is crucial to incorporate both early and deep layers for effective style loss computation.

Selecting an appropriately deep layer for content loss ensures that the model perceives the entire content image coherently (despite any compression introduced by pooling layers) and accurately represents it in the content loss.

VGG-19:

VGG-19 is a deeper model than AlexNet, which afforded us greater flexibility in selecting layers for the style transfer process. We experimented with various combinations of layers and found results that corroborate our earlier observations regarding AlexNet. Specifically, when we used only the last five convolutional layers of VGG-19, the results exhibited minimal variation in coloration compared to the original content image, as well as very little stylistic influence. This outcome can be attributed to the fact that the early layers are responsible for activating the deeper layers' structural style recognition through a complex feedback loop during updates, which is essential for effective style transfer. Omitting the early layers disrupts this coloration step, significantly hindering the overall process.

Another factor that can disrupt the formation of style patterns is assigning excessively large weights to certain layers. For example, if a deep layer is assigned a weight of 1e6 while other layers have much smaller weights, the optimization process may focus primarily on minimizing the loss from this layer. This can lead to insufficient minimization of losses from other layers, as the optimization process only considers the current step without accounting for how optimizing one layer's loss affects others. Conversely, assigning too large weights to early layers can result in an emphasis on color variation, causing the loss of larger patterns in the image.

Although we did not arrive at a definitive selection of layers, as we explored a variety of combinations during our style transfer experiments, we found that the layer choices presented in the instructional presentation which were layers 0,5,10,19,21 were satisfactory for achieving effective style transfer. We additionally used the sets of [5,10,30,32,34] and [25,28,30,32,34] in experimentation, the former worked well relatively to our tastes while the latter showed minimal results as can be seen below

Examples of the ineffectiveness of style transfer using only deeper layers, these images were produced using uniform weights for each layer while using layers 25,28,30,32 in the VGG-19 model (the four convolutional layers before the last convolutional layer which was used for content).

Classification Model Choice:

The following is a table with selected model performance metrics. We chose to evaluate the models by recall, specificity, precision and F1 score as they represent a comprehensive view of the model's classification abilities. The evaluations were done only on the stylized images and their content image base (evaluating the style paintings is redundant because the paintings would be evaluated by the models at some point during training, validation or test), absolute positive values were considered images style transferred with a Van Gogh painting and absolute negatives were either content images or style transferred images without a Van Gogh style.

Stylized by	classifier	recall	specificity	precision	F1 score
AlexNet	VGG19 finetuned	0.211765	1	1	0.349515
	AlexNet finetuned	0.58824	0.852941	0.909091	0.714286
VGG19	VGG19 finetuned	0.157303	1	1	0.271845
	AlexNet finetuned	0.29213	1	1	0.452174

As we can see in the table both the precision and the specificity are very high (1 for

both models for the style transfer, and for almost all models as classifiers, apart from 1 case when the fine-tuned AlexNet delivered high precision ~0.85 & specificity ~0.9). $precision = \frac{TP}{TP+FP}$, $specificity = \frac{TN}{N}$,This is because the models almost did not evaluate any non-Van Gogh styled images as drawn by him which indicates our models' excellent ability to identify non-Van Gogh styled images.

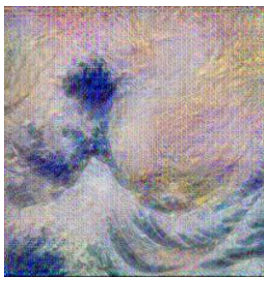
However, the recall of the models is very low (~0.15-0.3 for all models apart from 1 case when AlexNet was to classify the stylized images, as well as the pre-trained AlexNet which performed style transfer). In fact, it seems that no matter which model is used for Style Transfer purposes, AlexNet appears to deliver much better recall & F1 Score (which depends on the recall!) than VGG-19, despite providing at times slightly worse results in other metrics such as precision & specificity. This means a lot of images didn't pass our models' discrimination, even though they were styled as Van Gogh. We can interpret this as a strength of the model - indeed, the styled transferred images are not truly of Van Gogh's making and so our models technically achieved their purpose.

While both models' performance is similar, we choose the fine-tuned **VGG-19** model because it is more successful in discriminating against our stylized images, as noted by having lower recall and higher specificity, regardless of which pre trained model generated the Style Transfer images. It can be also attributed to its much deeper architecture, which can more flexibly extract even more abstract & meaningful features than shallower AlexNet.

Model Validation & Evaluation:

We can see that the fine-tuned models' results sometimes confirm and sometimes refute the differences we observed among the stylized paintings, using the naked eye. However, it's highly speculative, so we can never be 100% sure and our explanations here will be based more on intuition. From the stylized images mentioned above as examples, we can see that when there are very noticeable patterns in the paintings associated with Van Gogh, such as a combination of thick swirling brushstrokes, as well as his distinctively saturated & warm color palette, both models manage to successfully assign those newly stylized painting as Van Gogh. However, from the naked eye it appeared to us that AlexNet could classify such paintings as Van Gogh more easily, whereas VGG-19 was more stringent and didn't classify images as Van Gogh when it wasn't particularly confident about them. It also appeared to us that AlexNet is more sensitive to the specific color palette of the images, and if it can associate the colors with Van Gogh then it may prove significantly easier (compared to the same scenario under VGG-19). On the other hand, it also appeared to us that VGG-19 is more sensitive to the extent of the thick swirling brushstrokes of the images (typical sign of Van Gogh's

artwork), and if it can associate these with Van Gogh then it may prove significantly easier (compared to the same scenario under AlexNet). This explains well the nature of the predictions we observed initially in this part of the project, shown above. Here's another stylized image with warm colors and thick swirling brushstrokes which both models agrees to classify it as Van Gogh:



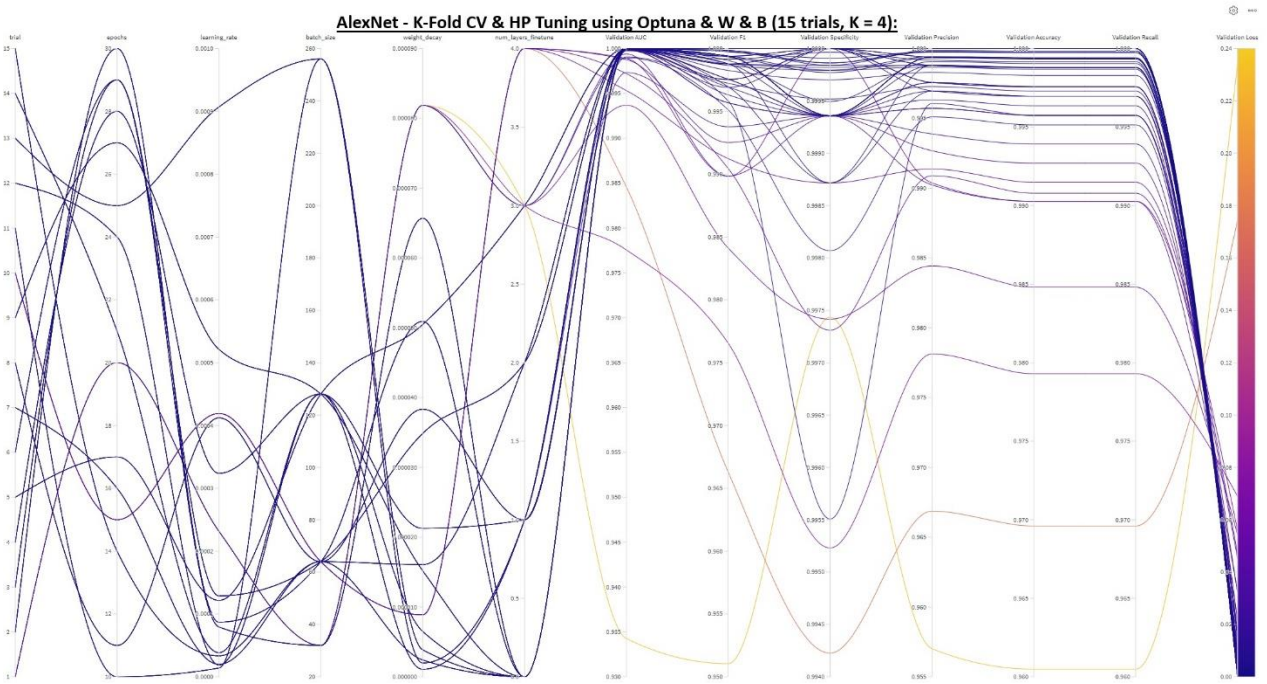
Note that this is not a rule of thumb and both models got predictions "wrong", even though it seemed like the stylized image satisfied the rules of thumb. A look at this picture of a dog would show clearly warm, saturated colors, typical to Van Gogh, as well as thick swirling brushstrokes, but both models didn't assign the image as Van Gogh. We thought it seemed very appropriate to assign it as Van Gogh but that wasn't the case:

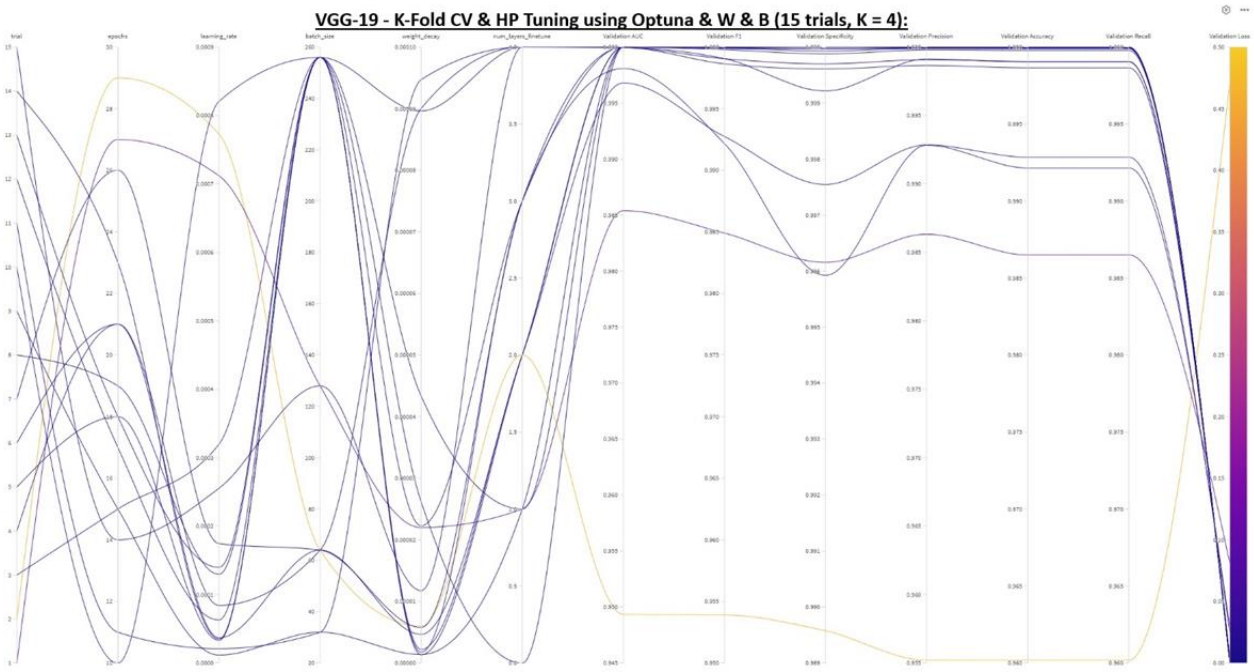
Results Comparison with Part 1:

As explained before, the fine-tuned CNN models delivered very impressive results in part 1, indicating that both of them manage to well distinguish between post impressionist paintings and correctly assign them to Van Gogh / not, with a slight lead on performance on unseen data going to VGG-19. However, during the Style Transfer procedure – part 2, we saw a sharp drop in both models' performances, especially with AlexNet, and specifically by looking at their recall & F1 values, indicating they are having a slight difficulty classifying newly generated images, stylized by Van Gogh, but have never been truly painted by him. Just like in part 1, by looking at specific metrics such precision & specificity, VGG-19 appeared to still have marginally better performance than AlexNet distinguishing the newly stylized images, regardless of the pre-trained model which generated those images. However, contrary to part 1, this time,

AlexNet has beaten VGG-19 with better precision & F1 score, regardless of the pre-trained model which generated those images. This is a speculation but the higher recall & lesser-equal precision could make sense if AlexNet assigns more stylized paintings as Van Gogh's (it's easier to be "positively assigned" ~ Van Gogh) compared to VGG-19, which could explain that there are more positively predicted stylized images and so it's easier to get some of them right and so recall may go up.

Appendices (Graphs & more!):





Final Training + Testing both CNN Models – Part 1: