Painting Style Recognition

A university project for the

Artificial Intelligence course

Developed by:

> Alireza Nobakht

> Arshia Abedin

> Moien Talebi

Supervisor: Dr. Pishgoo

In KNTU university of tech

Table of Contents

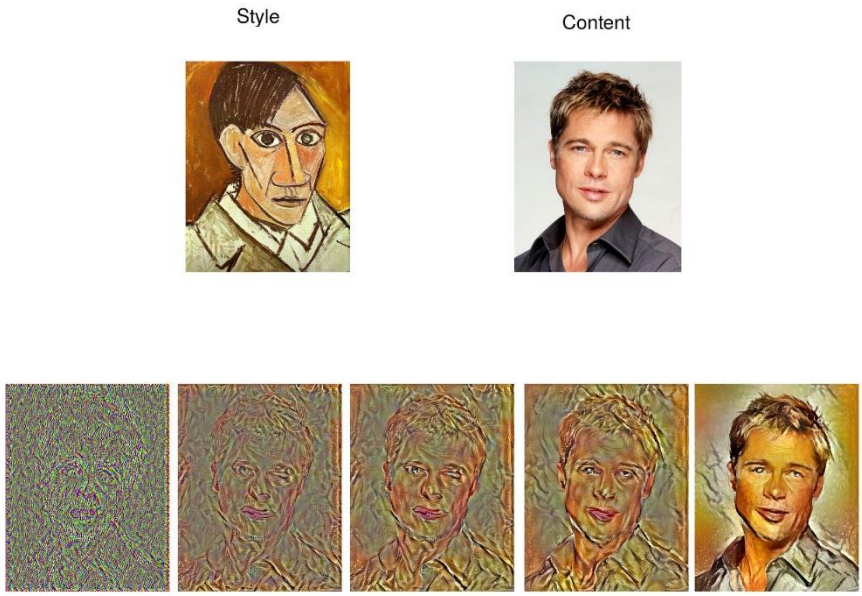
Introduction: Artificial Intelligence in Art

Art has consistently evolved in response to technological innovation, reflecting the intellectual and cultural progress of human societies. With the rapid advancement of Artificial Intelligence (AI), computational methods have become increasingly significant in analyzing and understanding artistic works. Among these methods, Deep Learning has emerged as a powerful approach for automatically recognizing and classifying painting styles.

Painting style recognition is a complex task due to the abstract and high-dimensional nature of visual art. Artistic styles are defined by subtle characteristics such as brush strokes, color palettes, composition patterns, texture, and visual rhythm. These features are often difficult to formalize using traditional rule-based or handcrafted feature extraction methods. As a result, conventional machine learning approaches may struggle to achieve high accuracy and generalization across diverse artistic movements.

Deep Learning addresses these challenges by enabling models to automatically learn hierarchical representations directly from image data. Convolutional Neural Networks (CNNs) have demonstrated strong performance in image classification tasks, including artistic style recognition. By training on large datasets of labeled paintings, CNN-based models can identify complex visual patterns and distinguish between styles such as Impressionism, Cubism, Baroque, Surrealism, and others with significant accuracy.

However, several challenges remain in this domain. The availability and quality of labeled art datasets can significantly affect model performance. Variations in image resolution, lighting conditions, digitization quality, and overlapping stylistic characteristics between art movements may introduce classification ambiguity. Additionally, deep models require substantial computational resources and large amounts of training data to achieve optimal results.



Despite these challenges, the application of Deep Learning in painting style recognition offers substantial benefits. It contributes to automated art analysis, digital archiving, educational tools, museum curation systems, and recommendation engines in online art platforms. Furthermore, it provides researchers with quantitative methods to study stylistic evolution and artistic influence across historical periods.

Project Introduction and Overview

This project focuses on **painting style recognition using Deep Learning**, aiming to automatically identify the artistic style of a given painting image. The task is formulated as a **multi-class image classification problem**, where each input image is assigned to one of several predefined art movements, such as Impressionism, Cubism, Renaissance, Surrealism, and others.

The core objective of the project is to design, train, and evaluate a deep learning–based model capable of learning discriminative visual features that characterize different painting styles. Due to the abstract nature of art, stylistic elements such as brush strokes, color composition, texture, and structural patterns are often subtle and highly variable, which makes this problem particularly challenging for traditional image processing and rule-based methods.

To address these challenges, the project employs **Convolutional Neural Networks (CNNs)**, which are well-suited for visual pattern recognition. The model is trained on the **WikiArt dataset**, a large-scale and widely used art dataset containing paintings categorized by artistic style. Given the original dataset's large size and class imbalance, data curation and sampling strategies are applied to create a balanced and manageable subset suitable for efficient training and evaluation.

This project is structured as an academic Artificial Intelligence course project and is divided into multiple phases. The first phase establishes a baseline CNN model, defines the data preprocessing pipeline, and evaluates initial performance. Subsequent phases are designed to explore more advanced architectures and techniques, such as transfer learning with deep pre-trained models, to improve classification accuracy and robustness.

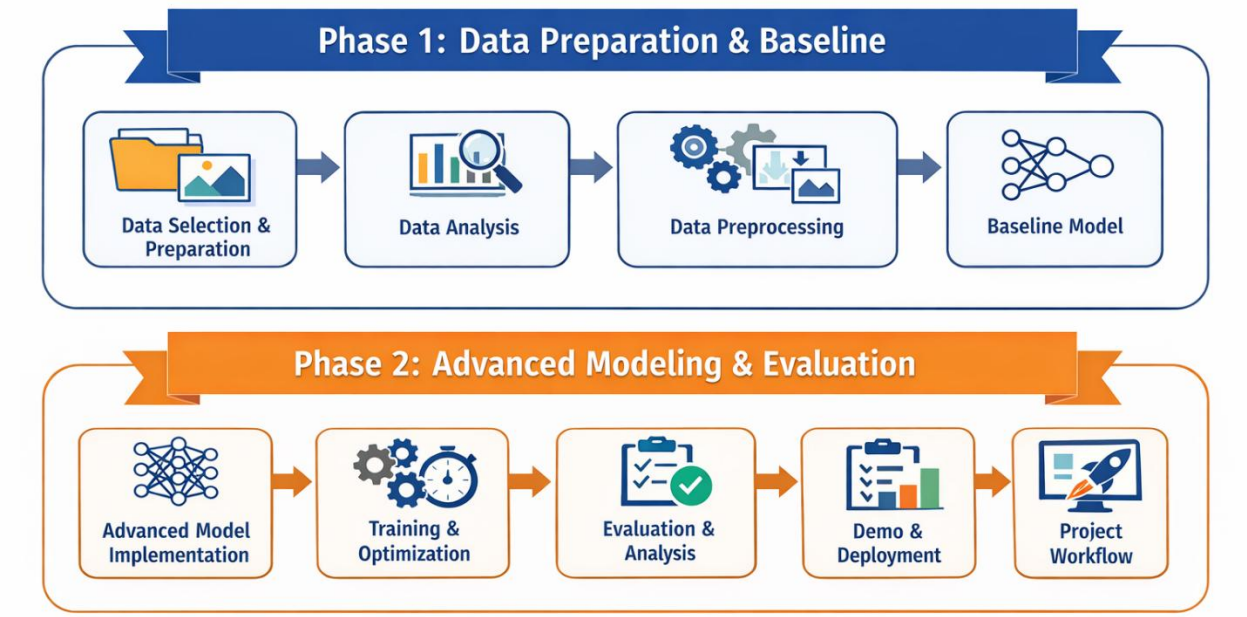
Overall, this work demonstrates the practical application of **Deep Learning in computational art analysis**, highlighting both its effectiveness and its challenges in recognizing complex artistic styles from visual data.

Workflow

This project is divided into two main phases: **theoretical** and **practical**.

In the first phase, the focus was on data preparation, analysis, and designing the overall modeling strategy. In the second phase, the proposed design was implemented, tested, and systematically improved to enhance performance and evaluation results.

Dividing the project into two structured phases provides several advantages. By separating the **theoretical planning stage** from the **practical implementation stage**, the development process becomes more organized, goal-oriented, and analytically grounded.



Project Workflow Overview

In the first phase, careful data preparation and strategy design reduce the risk of architectural or experimental mistakes during implementation. This structured planning ensures that model development is based on informed decisions rather than trial-and-error. In the second phase, the focus shifts to execution, evaluation, and systematic improvement, allowing performance optimization based on measurable results.

In the following sections, a detailed explanation of each phase will be presented, including the specific steps performed and the results obtained throughout the project.

Phase #1

Documentation

Data Preparation and Baseline Model

• Dataset Overview

The WikiArt dataset was used as the primary data source for this project. The dataset contains paintings categorized into 27 different artistic styles.

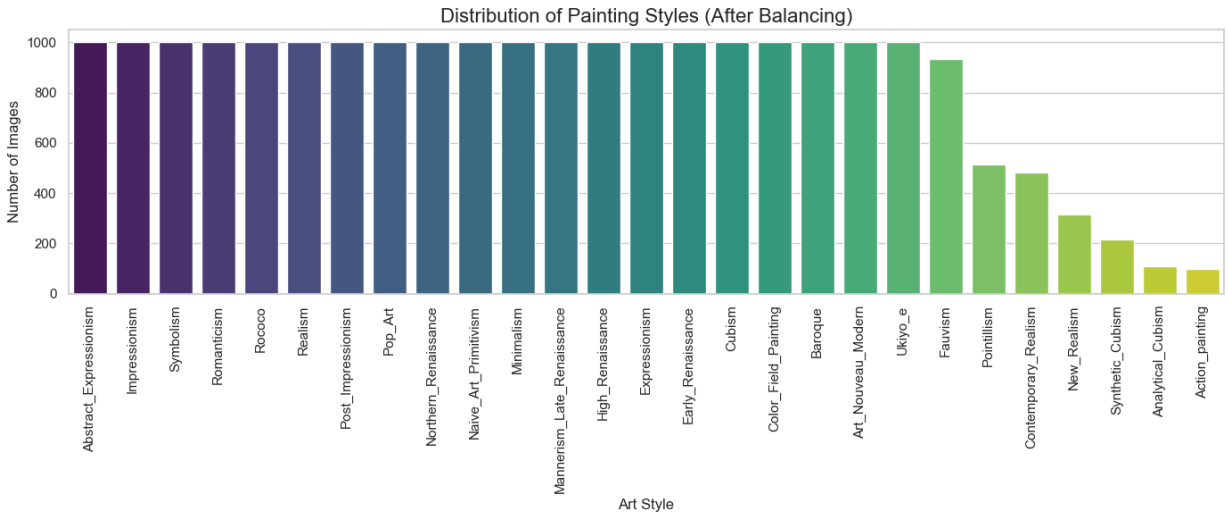
Due to class imbalance in the original dataset, an under-sampling strategy was applied. For each style, 4000 images were randomly selected to create a balanced dataset suitable for training.

The final processed dataset characteristics are:

- Number of classes: 27
- Images per class: 4000
- Data split strategy: Stratified split
- Training set: 70%
- Validation set: 15%
- Test set: 15%

• Class Distribution After Balancing

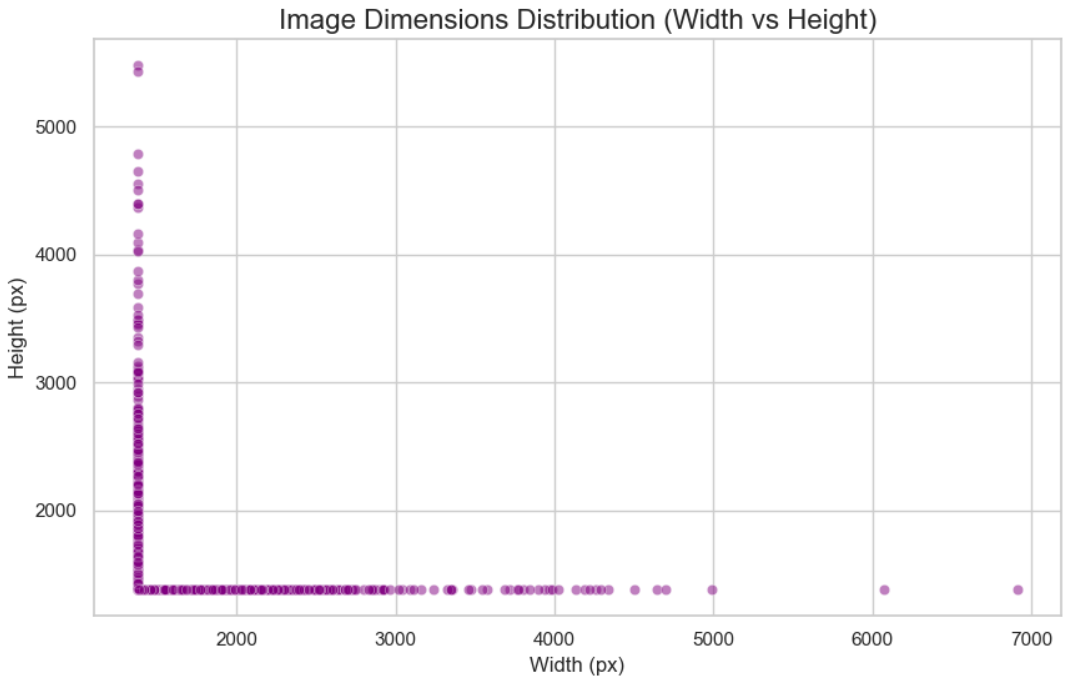
The following figure shows the class distribution after applying the sampling strategy.



The distribution confirms that all classes are balanced, reducing the risk of model bias toward majority classes.

• Image Dimension Analysis

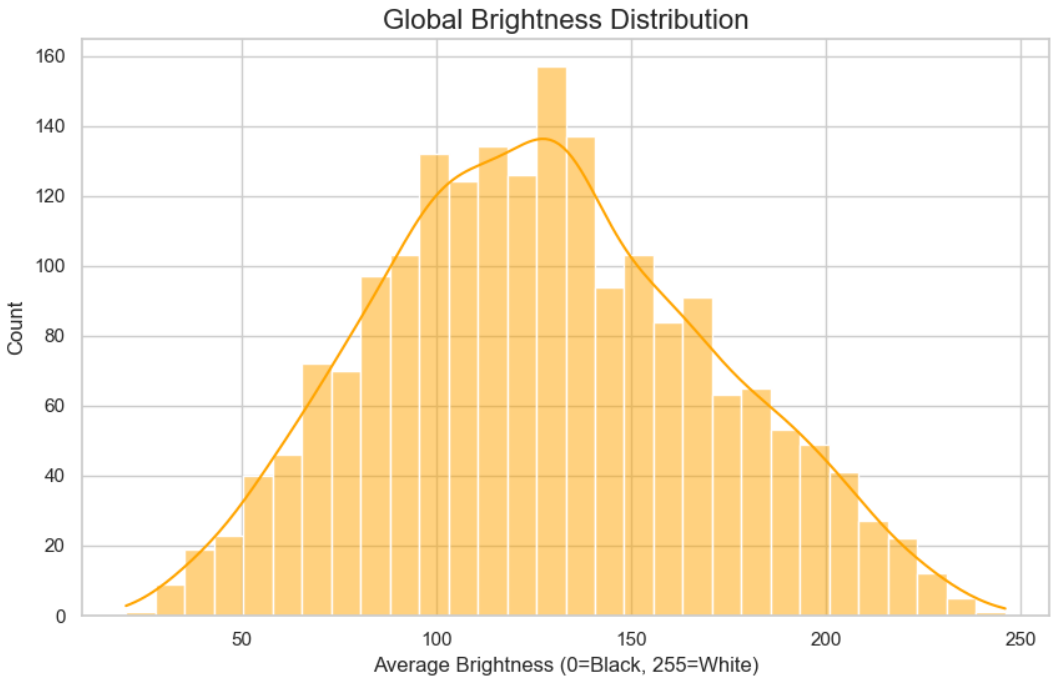
An analysis of image dimensions was performed to understand variability across the dataset.



The results indicate significant variation in image width and height, which justifies the need for resizing during preprocessing.

• Brightness Distribution

The brightness distribution of images was analyzed to detect potential lighting inconsistencies.



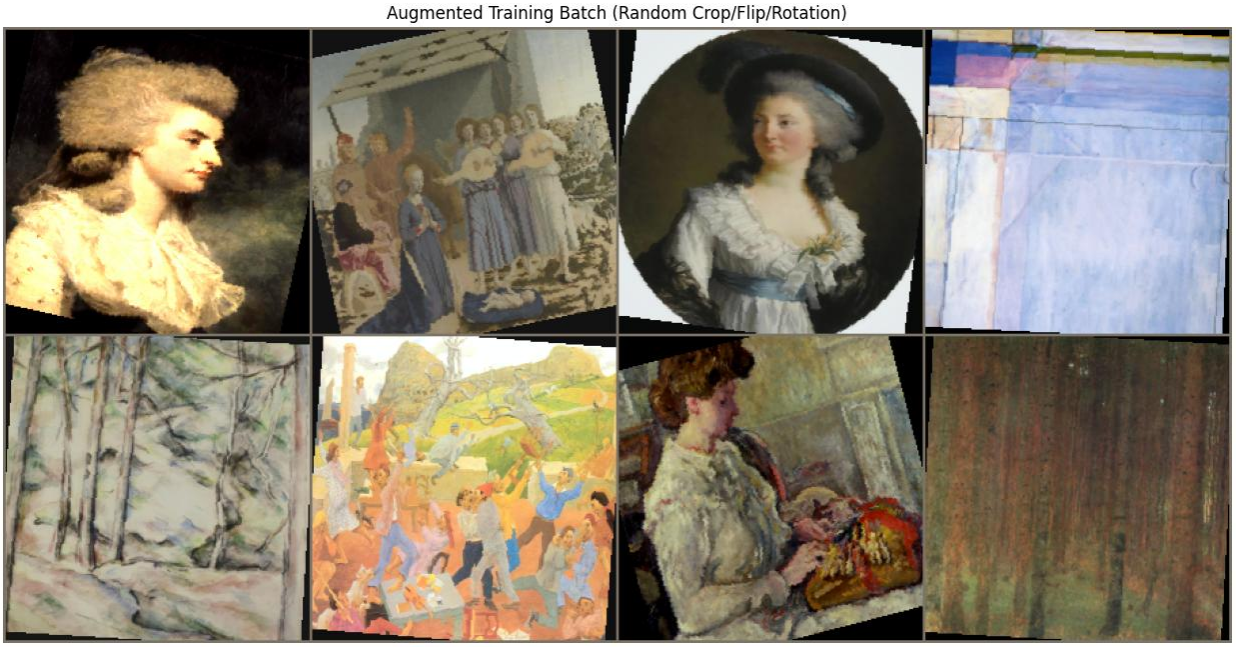
The histogram shows a near-normal distribution of brightness values, indicating that no extreme contrast correction was required.

• Data Transformation Steps

All images were processed through a standardized preprocessing pipeline before being fed into the neural network.

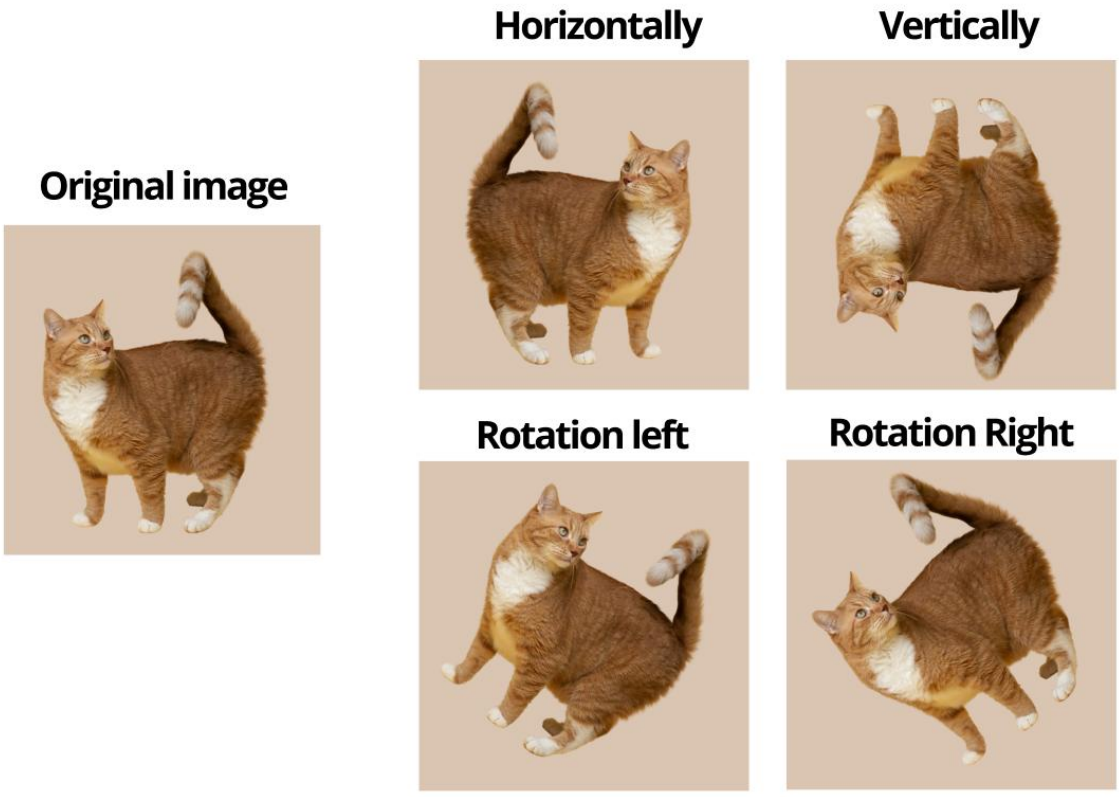
The applied transformations include:

- Resize to 384×384 pixels
- Random Rotation (±15 degrees)
- Random Resized Crop
- Horizontal Flip (training set only)
- Normalization using ImageNet mean and standard deviation



• Augmentation Samples

To improve generalization and robustness, data augmentation was applied to the training set.



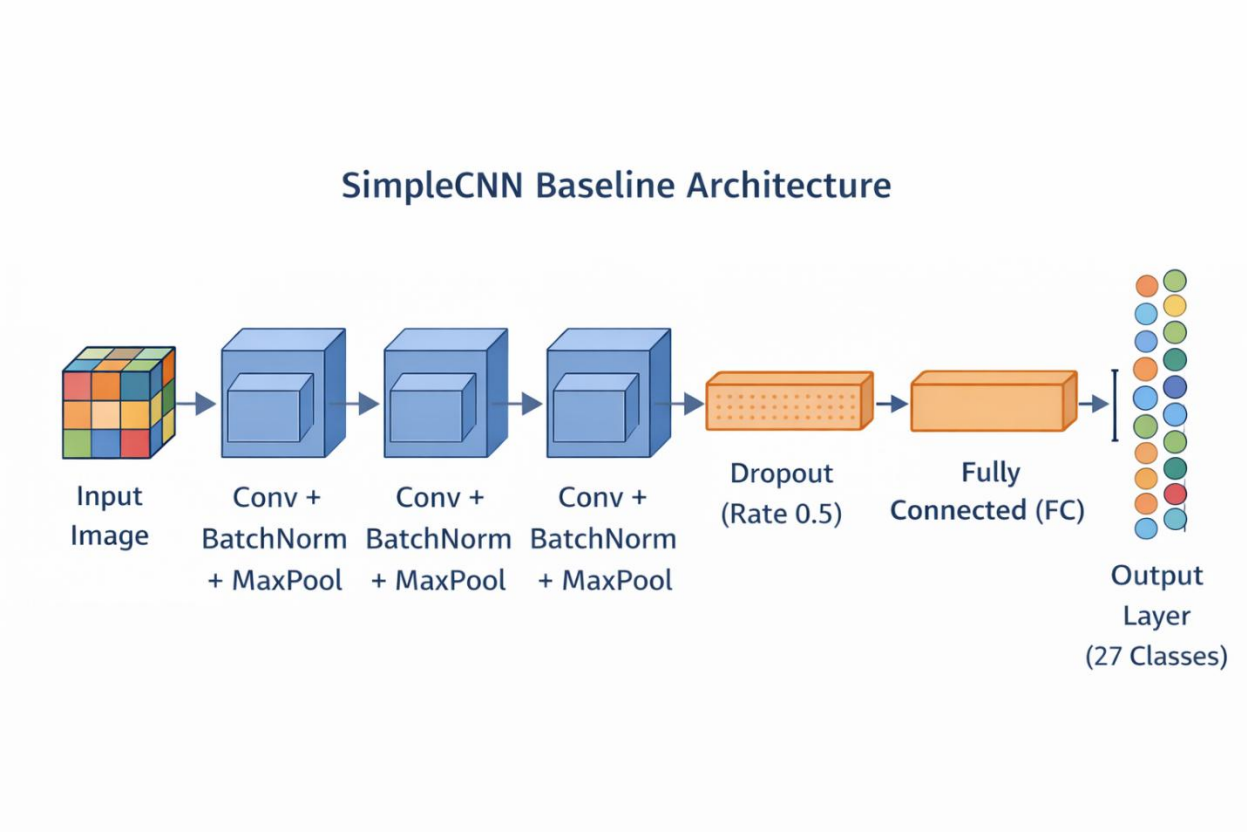
The augmented samples demonstrate variability introduced into the dataset to prevent overfitting.

• Model Architecture

A SimpleCNN model was designed as the baseline architecture.

The architecture consists of:

- Four convolutional layers
- Batch normalization layers
- Max pooling layers
- Two fully connected layers
- Dropout layer (rate = 0.5)



This baseline model serves as a performance benchmark for future improvements.

• Training Configuration

The model was trained using the following configuration:

- Loss Function: CrossEntropyLoss
- Optimizer: Adam
- Learning Rate: 0.000009
- Batch Size: 32

Phase 1 Summary

Phase 1 successfully established:

- A balanced and curated dataset
- A standardized preprocessing pipeline
- A baseline CNN architecture
- Quantitative performance benchmarks

These results provide a structured foundation for Phase 2, where advanced architectures and optimization strategies will be implemented to improve classification performance.

Phase #2

Documentation

Advanced Modeling, Evaluation, and Explainability

Professional Software Refactoring

In the second phase of the Painting Style Recognition project, the workflow transitioned from experimental research to a **Modular Production Pipeline**. This shift ensures that the training, evaluation, and deployment processes are decoupled, reproducible, and scalable.

1. From Research Prototypes to Production Architecture

The primary milestone of Phase 2 was the transition from fragmented Jupyter Notebooks to a **Structured Python Package**. This architectural overhaul was designed to fulfill the industry standards of **Separation of Concerns (SoC)**, ensuring that data processing, model orchestration, and evaluation logic exist as independent, reusable modules.

2. The src Core: A Component-Based Design

The project was refactored into a centralized src directory. This transformation allows for a streamlined pipeline where different components interact through defined interfaces rather than global notebook variables. model to the intricate textures of paintings.

2.1. The Model Factory Pattern (src/models/models.py)

To handle the complexity of testing multiple deep learning backbones (ResNet50, EfficientNet-B0, EfficientNet-B3), we implemented a **Factory Pattern**. This centralizes the model instantiation logic and ensures that architecture modifications—such as custom classifier heads and dropout layers—are applied consistently.

```
elif model_name == 'efficientnet_b0':
    self.base_model = efficientnet_b0(weights=EfficientNet_B0_Weights.IMAGENET1K_V1)

    if not fine_tune:
        for param in self.base_model.parameters():
            param.requires_grad = False

    num_ftrs = self.base_model.classifier[1].in_features
    self.base_model.classifier[1] = nn.Linear(num_ftrs, num_classes)
```

- **Implementation Detail:** The TransferLearningModel class acts as a wrapper that dynamically injects the appropriate classification layer based on the backbone's feature map size.

2.2. Advanced Pipeline Orchestration (src/training/)

Training logic was decoupled into two specialized scripts to handle different stages of the model lifecycle:

- **train.py (The Engine):** Manages the standard training loop, incorporating **Early Stopping** and **ReduceLROnPlateau** schedulers. It ensures reproducibility by utilizing a centralized set_seed utility.
- **fine_tune.py (The Optimizer):** Designed for high-performance refinement. It introduces **Mixed Precision Training (AMP)** to optimize GPU memory and speed and implements a selective **Layer Unfreezing** strategy to adapt pre-trained weights to specific artistic textures.

2.3. Automated Evaluation & Analytics (src/evaluation/evaluate.py)

The evaluation phase was automated to generate a "Scientific Report" for every trained model. This module eliminates manual testing by:

1. **Metric Serialization:** Saving classification reports (F1, Precision, Recall) into JSON files for version tracking.
2. **Diagnostic Visualization:** Automatically generating Confusion Matrices and **Error Analysis Grids** (False Positives) to provide immediate feedback on model weaknesses.

2.4. Interface Decoupling (src/app.py)

The final system is exposed via a deployment layer that is completely independent of the training environment. The app.py script leverages the src modules to load the best-performing weights into a **Gradio-based Web UI**, providing an end-to-end user experience from image upload to style inference.

3. Standardized Project Directory Structure

The following tree represents the final engineered state of the project, ensuring a professional hand-off and scalability:

```
├── data/
│   └── processed/          # Versioned data splits (train/val/test)
│   └── raw/                # raw dataset
├── models/                 # Model Registry (best_model_efficientnet_b3.pth)
├── notebooks/              # Research, EDA, and Interpretability (Grad-CAM)
├── results/
│   ├── figures/            # Automatically generated Confusion Matrices
│   └── metrics/            # JSON serialized performance reports
└── src/                    # The Production Source Code
    ├── models/             # Architecture Factory
    ├── training/           # Training & Fine-tuning Orchestrators
    ├── evaluation/         # Automated Diagnostic Tools
    ├── preprocessing/      # Dataset & Transform definitions
    ├── utils/              # Helper functions (Logging, Seed, Metrics)
    └── app.py              # Deployment Interface (Gradio)
```

Scientific Training Protocol and Experiment Tracking

1. Controlled Training Methodology

The core of Phase 2 involved transitioning to a highly controlled, scientific training environment. To achieve high accuracy across 27 distinct art styles, we moved beyond basic training into a multi-stage **Fine-Tuning** strategy.

1.1. Strategy: Selective Layer Unfreezing

Rather than training the entire network from scratch, which often leads to catastrophic forgetting of pre-trained features, we implemented a selective unfreezing protocol in fine_tune.py.

- **Logic:** We first freeze the entire backbone and then specifically unfreeze the deepest blocks (e.g., the last 3 stages of EfficientNet-B3). These layers are responsible for high-level semantic features—such as brushstroke patterns and complex textures—which are critical for artistic style recognition.

```
elif model_type in ['efficientnet_b0', 'efficientnet_b3', 'efficientnet_b4']:
    # Unfreeze Classifier
    for param in model.base_model.classifier.parameters():
        param.requires_grad = True

    # Unfreeze the last 3 blocks of features (Deepest layers)
    for param in model.base_model.features[-3:].parameters():
        param.requires_grad = True
```

2. Optimization and Hyperparameter Rationale

The selection of hyperparameters was not arbitrary; it was based on the convergence behavior of deep architectures like EfficientNet.

2.1. Learning Rate Selection (2e-4)

We selected a learning rate of 2e-4

- **The "Why":** Standard transfer learning often uses 10^-3, but for fine-tuning pre-trained weights, a lower rate is essential to prevent the "explosion" of gradients that

would destroy the pre-trained ImageNet features. This specific rate provides a balance between rapid convergence and stability.

2.2. Advanced Optimizers and Schedulers

• AdamW **Optimizer:** Used with a weight decay of 0.01 to provide better regularization than standard Adam.

• **Cosine Annealing Warm Restarts:** This scheduler (CosineAnnealingWarmRestarts) cyclically varies the learning rate. This allows the model to "escape" local minima and find flatter, more generalizable optima in the loss landscape.

2.3. Regularization: Label Smoothing

To handle the inherent ambiguity between similar styles (e.g., Baroque vs. Renaissance), we utilized **Label Smoothing (0.1)** in our Cross-Entropy loss. This prevents the model from becoming overconfident in its predictions, leading to better generalization on the Test Set.

3. Monitoring and Reproducibility

A key requirement of Phase 2 was the ability to track experiments and ensure every result can be reproduced.

3.1. Experiment Tracking with Weights & Biases (WandB)

We integrated **WandB** as our primary monitoring tool. Unlike standard logs, WandB allowed us to visualize the training dynamics in real-time.

- **System Metrics:** Tracked GPU memory usage and compute efficiency.
- **Gradient Monitoring:** Used wandb.watch(model) to log gradients and histograms of weights, ensuring that the unfreezing process was not causing vanishing gradients.
- **Metric Logging:** Every epoch, training/validation loss and accuracy were synced to the cloud dashboard.

```
elif model_type in ['efficientnet_b0', 'efficientnet_b3', 'efficientnet_b4']:
    # Unfreeze Classifier
    for param in model.base_model.classifier.parameters():
        param.requires_grad = True

    # Unfreeze the last 3 blocks of features (Deepest layers)
    for param in model.base_model.features[-3:].parameters():
        param.requires_grad = True
```

3.2. Validation Strategy and Checkpointing

Training was governed by the performance on the **Validation Set**, which acted as a proxy for the unseen Test Set.

- **Best Model Checkpointing:** We implemented a "Best-Save" logic. The model state is only saved to models/best_model_efficientnet_b3.pth if the current validation accuracy exceeds the previous maximum.
- **Reproducibility:** Every run is initialized via the set_seed(42) function in utils.py, which fixes the seeds for Python, NumPy, and PyTorch (including CuDNN determinism), ensuring that results are consistent across different machines.

4. Performance Optimization: Mixed Precision (AMP)

To accelerate training on modern GPUs, we utilized **Automatic Mixed Precision (AMP)**. By performing forward and backward passes in float16 while maintaining master weights in float32, we achieved significantly faster epoch times without sacrificing model precision.

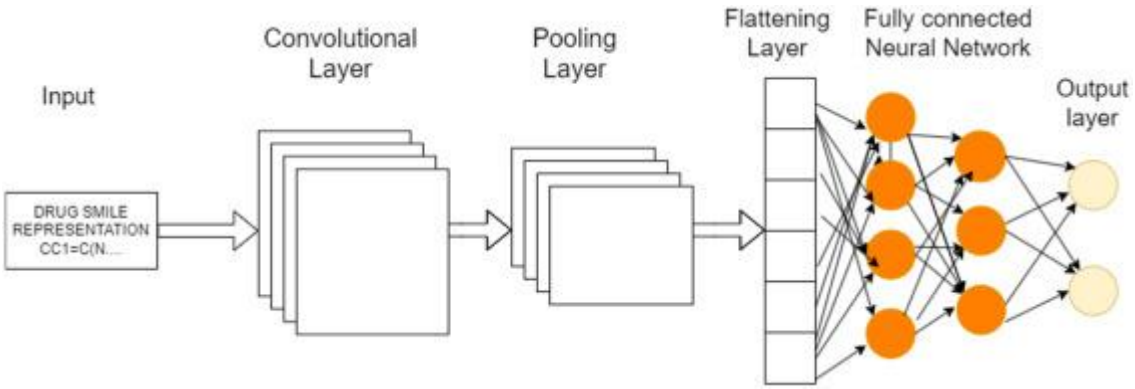
```
optimizer.zero_grad(set_to_none=True)

with torch.amp.autocast(device_type='cuda', dtype=torch.float16):
    outputs = model(inputs)
    loss = criterion(outputs, labels)

scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

Neural Architecture Design & Implementation Logic

In this stage, the project transitioned from a high-level conceptual design to a rigorous, code-centric implementation. We focused on three main pillars: **Structural Stability**, **Parameter Efficiency**, and **Task Adaptation**.

1. The Custom Baseline: SimpleCNN Anatomy

The SimpleCNN class in src/models/models.py was engineered to establish a performance floor. The implementation emphasizes manual feature extraction through sequential convolutional blocks.



1.1. Structural Stability with BatchNorm

To ensure numerical stability during the forward pass, we integrated nn.BatchNorm2d after every convolution. This is critical for art style recognition where color distributions can vary wildly between styles.

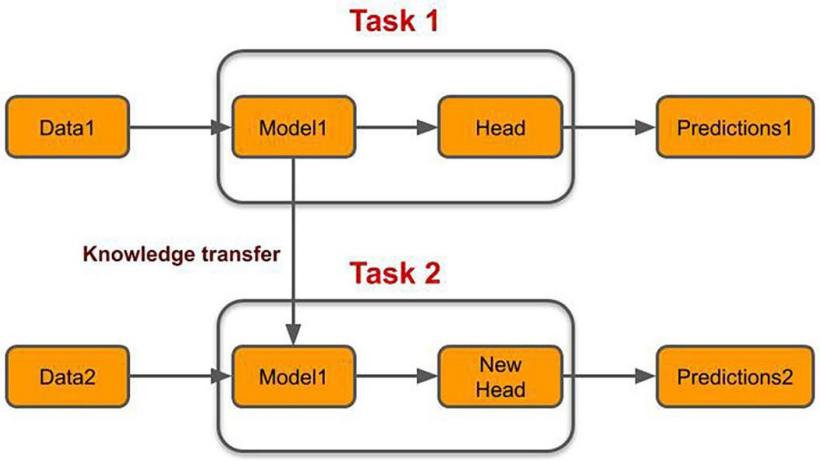
1.2. The Flattening Strategy

One of the most significant engineering challenges in the Baseline was the transition from spatial features to class probabilities. We calculated a fixed flatten_dim based on the 384x384 input resolution after three pooling operations.

```python
class SimpleCNN(nn.Module):
    """The baseline model from Phase 1."""
    def __init__(self, num_classes=27):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        self.flatten_dim = 128 * 28 * 28
        self.fc1 = nn.Linear(self.flatten_dim, 512)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, num_classes)
```

2. Advanced Backbone Surgery: TransferLearningModel

The TransferLearningModel class represents our move toward "Production-Grade" AI. We utilized architectural surgery to adapt SOTA backbones like **ResNet50** and **EfficientNet** to our specific 27-class problem.



2.1 EfficientNet-B3: Compound Scaling in Practice

Unlike the manual depth of SimpleCNN, **EfficientNet-B3** uses compound scaling. In our implementation, we "hijacked" the pre-trained classifier to map ImageNet features to artistic styles:

```
elif model_name == 'efficientnet_b3':
    # EfficientNet-B3: Higher resolution (300-384px) & Deeper
    self.base_model = efficientnet_b3(weights=EfficientNet_B3_Weights.IMAGENET1K_V1)

    # Classifier modification
    num_ftrs = self.base_model.classifier[1].in_features
    self.base_model.classifier[1] = nn.Linear(num_ftrs, num_classes)
```

2.2 Regularization through Sequential Heads (ResNet50)

For the **ResNet50** variant, we implemented a more aggressive regularization head to combat the tendency of deep networks to overfit smaller datasets:

```
if model_name == 'resnet50':
    self.base_model = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)
    if not fine_tune:
        for param in self.base_model.parameters():
            param.requires_grad = False

    num_ftrs = self.base_model.fc.in_features
    self.base_model.fc = nn.Sequential(
        nn.Dropout(0.5),
        nn.Linear(num_ftrs, num_classes)
    )
```

3. Data-Centric Regularization: The Transform Pipeline

Architecture alone does not solve overfitting. In src/preprocessing/transforms.py, we implemented a "Heavy Augmentation" strategy that acts as a stochastic regularization layer.

We replaced standard rotations with TrivialAugmentWide and RandomErasing. The latter forces the network to learn global artistic context rather than relying on local "signatures" (like an artist's signature in a corner):

```
return transforms.Compose([
    transforms.RandomResizedCrop(IMG_SIZE, scale=(0.7, 1.0)), # Increased scale range
    transforms.RandomHorizontalFlip(p=0.5),

    transforms.TrivialAugmentWide(),

    transforms.ToTensor(),
    transforms.Normalize(mean=MEAN, std=STD),

    transforms.RandomErasing(p=0.2, scale=(0.02, 0.2)),
])
```

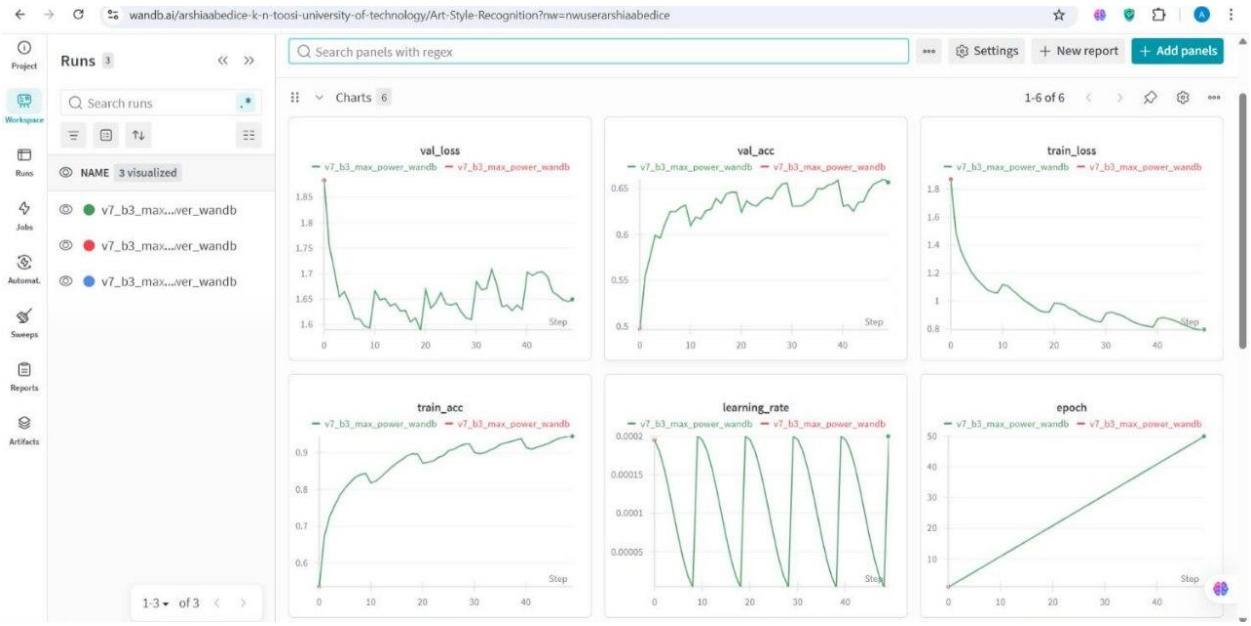
4. Engineering for Scalability: The get_model Factory

To ensure the code is clean and scalable, we abstracted the instantiation logic into a factory function. This allows the training script to remain agnostic of the specific model architecture.

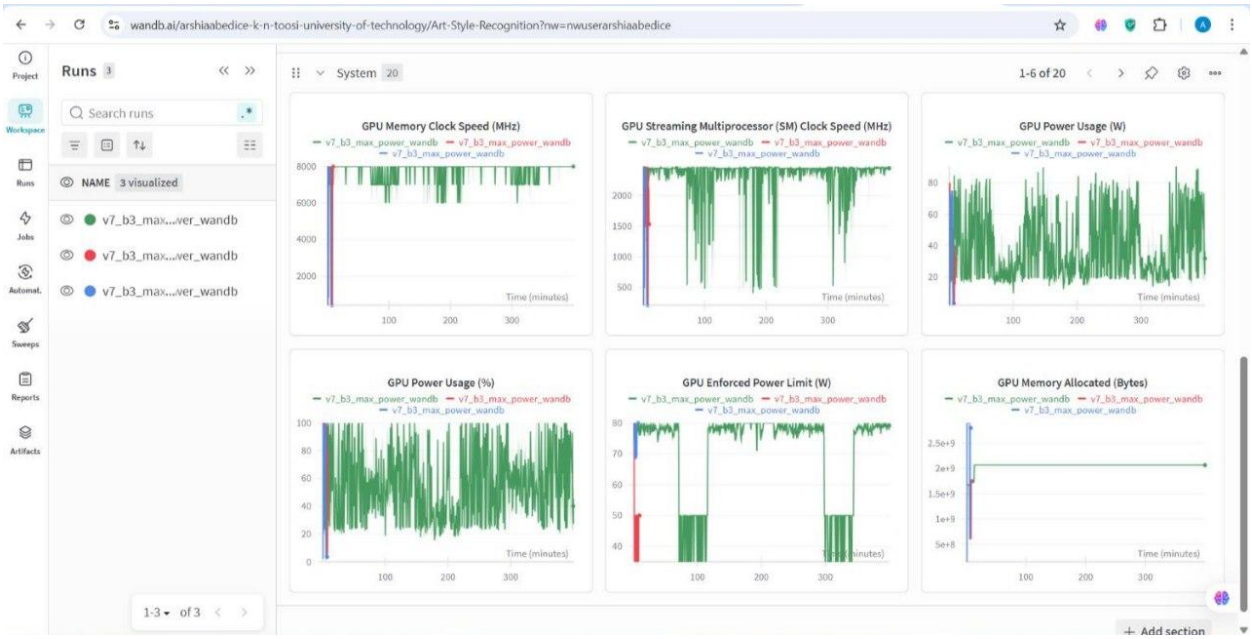
```
def get_model(model_type, num_classes=27, device='cpu'):
    if model_type == 'baseline':
        model = SimpleCNN(num_classes)
    elif model_type in ['resnet50', 'efficientnet_b0', 'efficientnet_b3']:
        model = TransferLearningModel(num_classes, model_name=model_type, fine_tune=True)
    else:
        raise ValueError(f"Unknown model type: {model_type}")
```

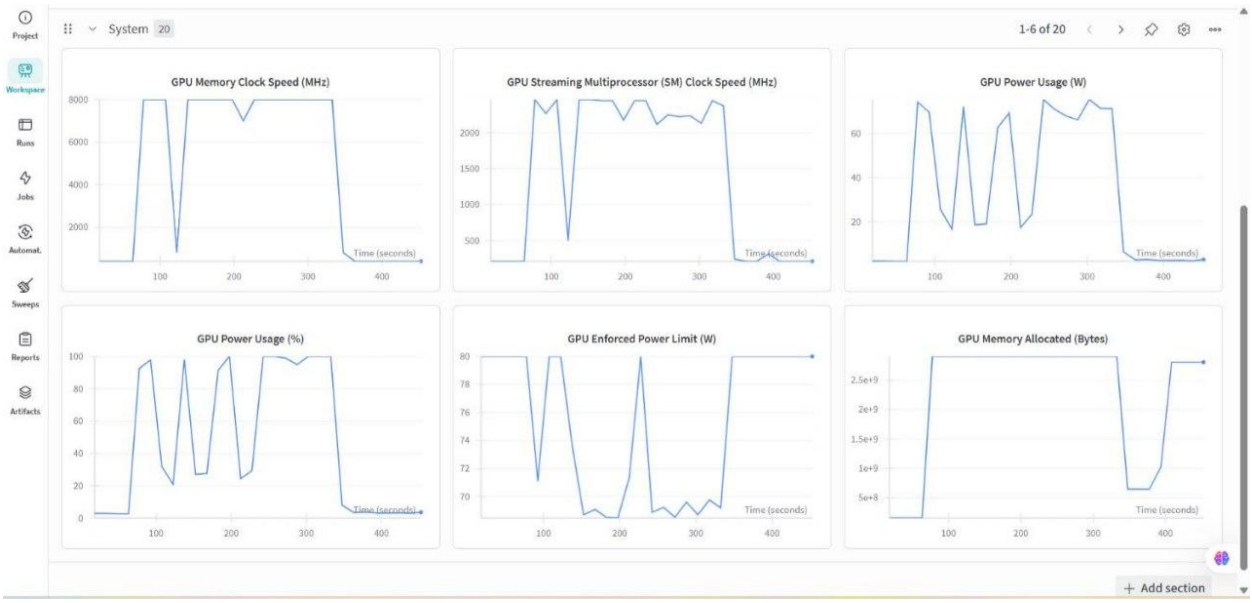
Results:

(Training & Optimization)



(Hardware & Perfprmance)





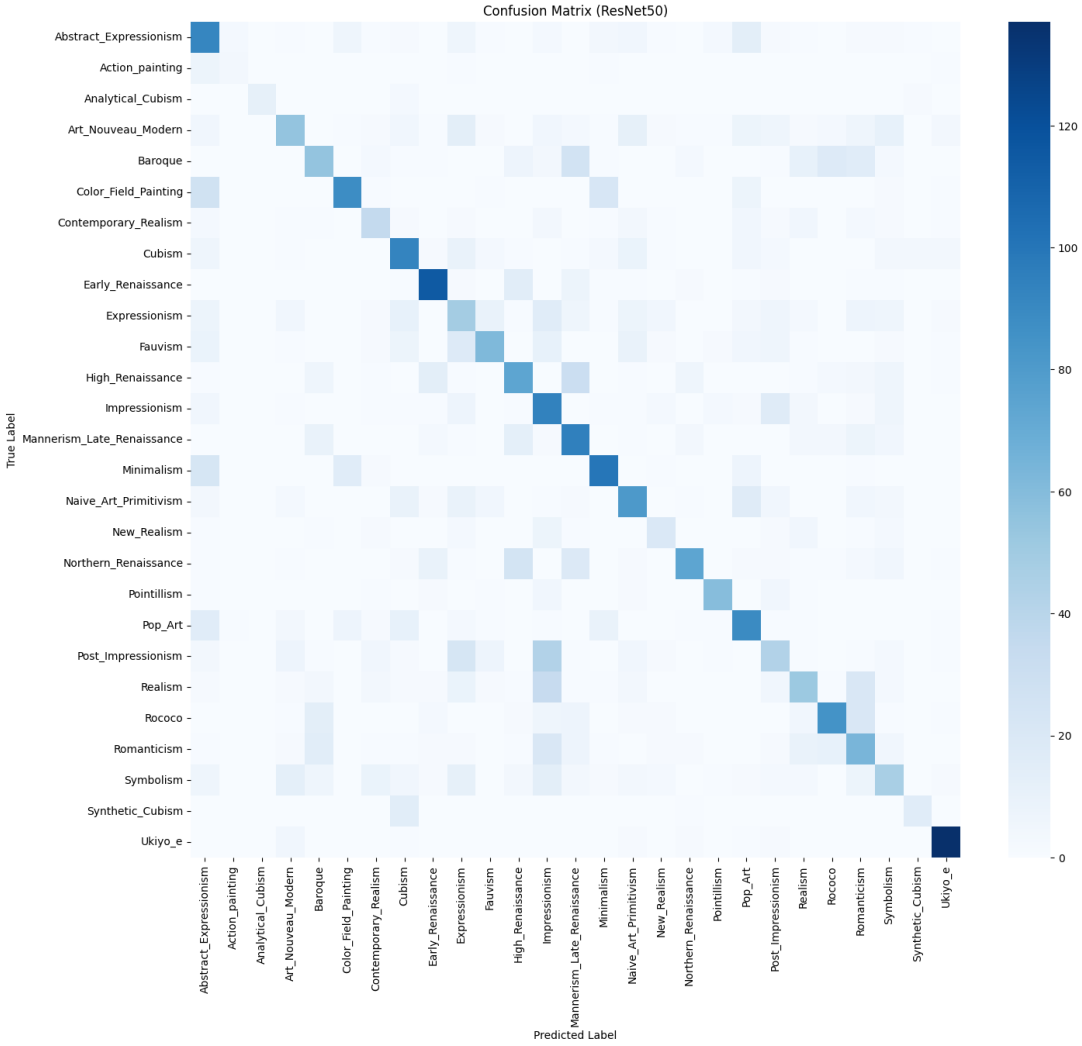
(Quantitative Analysis)

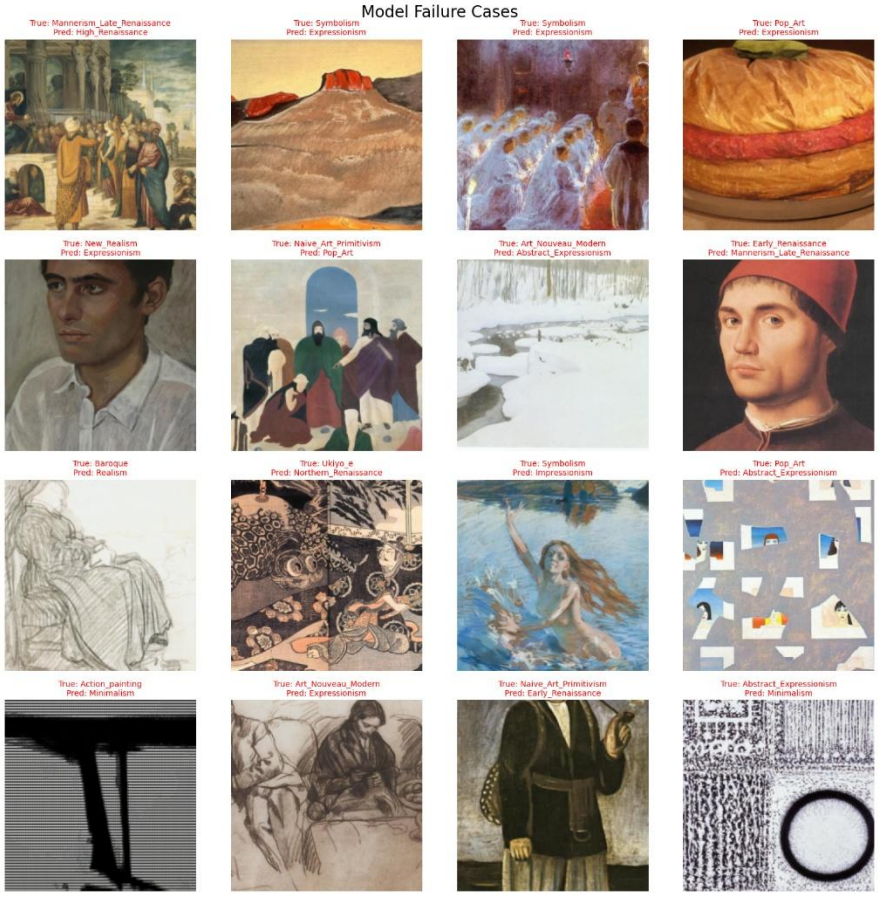
(model NOT Improved)

🏆 Test Accuracy: 0.5265

```
------------------------------
                          precision    recall  f1-score   support

    Abstract_Expressionism       0.41      0.61      0.49       150
           Action_painting       0.50      0.27      0.35        15
         Analytical_Cubism       1.00      0.71      0.83        17
       Art_Nouveau_Modern       0.53      0.37      0.43       150
                  Baroque       0.49      0.37      0.42       150
     Color_Field_Painting       0.75      0.59      0.66       150
     Contemporary_Realism       0.47      0.49      0.48        72
                   Cubism       0.53      0.62      0.57       150
        Early_Renaissance       0.72      0.77      0.74       150
            Expressionism       0.28      0.33      0.30       150
                  Fauvism       0.65      0.44      0.53       140
         High_Renaissance       0.51      0.49      0.50       150
            Impressionism       0.35      0.63      0.45       150
Mannerism_Late_Renaissance       0.45      0.63      0.52       150
                Minimalism       0.67      0.67      0.67       150
     Naive_Art_Primitivism       0.53      0.54      0.53       150
              New_Realism       0.44      0.43      0.43        47
      Northern_Renaissance       0.76      0.49      0.60       150
               Pointillism       0.83      0.77      0.80        77
                   Pop_Art       0.52      0.59      0.56       150
       Post_Impressionism       0.37      0.29      0.32       150
                  Realism       0.47      0.35      0.40       150
                   Rococo       0.67      0.57      0.62       150
              Romanticism       0.37      0.43      0.40       150
                Symbolism       0.39      0.31      0.35       150
         Synthetic_Cubism       0.70      0.50      0.58        32
                  Ukiyo_e       0.87      0.91      0.89       150

                 accuracy                           0.53      3400
                macro avg       0.56      0.52      0.53      3400
             weighted avg       0.54      0.53      0.53      3400
```

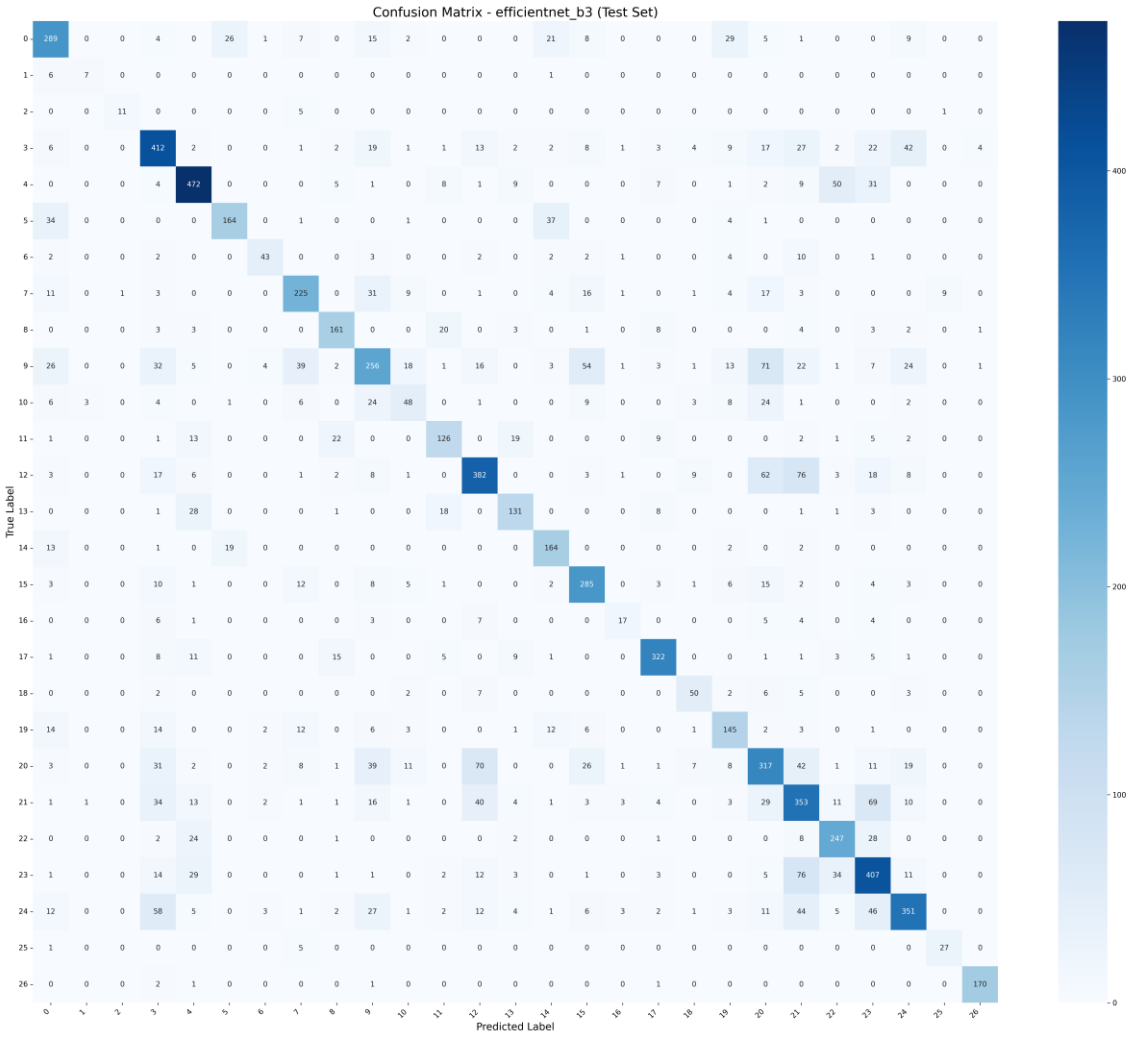
Full history saved in metrics/runs/*.json files!!!

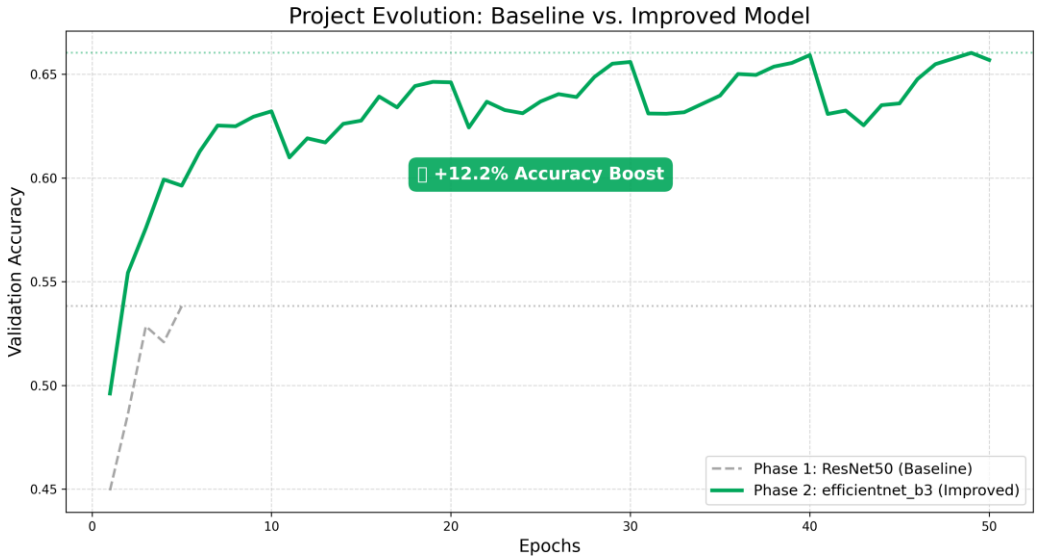

Confusion Matrix (ResNet50)
True Label
Predicted Label
Abstract_Expressionism
Action_painting
Analytical_Cubism
Art_Nouveau_Modern
Baroque
Color_Field_Painting
Contemporary_Realism
Cubism
Early_Renaissance
Expressionism
Fauvism
High_Renaissance
Impressionism
Mannerism_Late_Renaissance
Minimalism
Naive_Art_Primitivism
New_Realism
Northern_Renaissance
Pointillism
Pop_Art
Post_Impressionism
Realism
Rococo
Romanticism
Symbolism
Synthetic_Cubism
Ukiyo_e
0
20
40
60
80
100
120


Model Failure Cases
True: Mannerism_Late_Renaissance
Pred: High_Renaissance
True: Symbolism
Pred: Expressionism
True: Symbolism
Pred: Expressionism
True: Pop_Art
Pred: Expressionism
True: New_Realism
Pred: Expressionism
True: Naive_Art_Primitivism
Pred: Pop_Art
True: Art_Nouveau_Modern
Pred: Abstract_Expressionism
True: Early_Renaissance
Pred: Mannerism_Late_Renaissance
True: Baroque
Pred: Realism
True: Ukiyo_e
Pred: Northern_Renaissance
True: Symbolism
Pred: Impressionism
True: Pop_Art
Pred: Abstract_Expressionism
True: Action_painting
Pred: Minimalism
True: Art_Nouveau_Modern
Pred: Expressionism
True: Naive_Art_Primitivism
Pred: Early_Renaissance
True: Abstract_Expressionism
Pred: Minimalism

(Model Improvements)

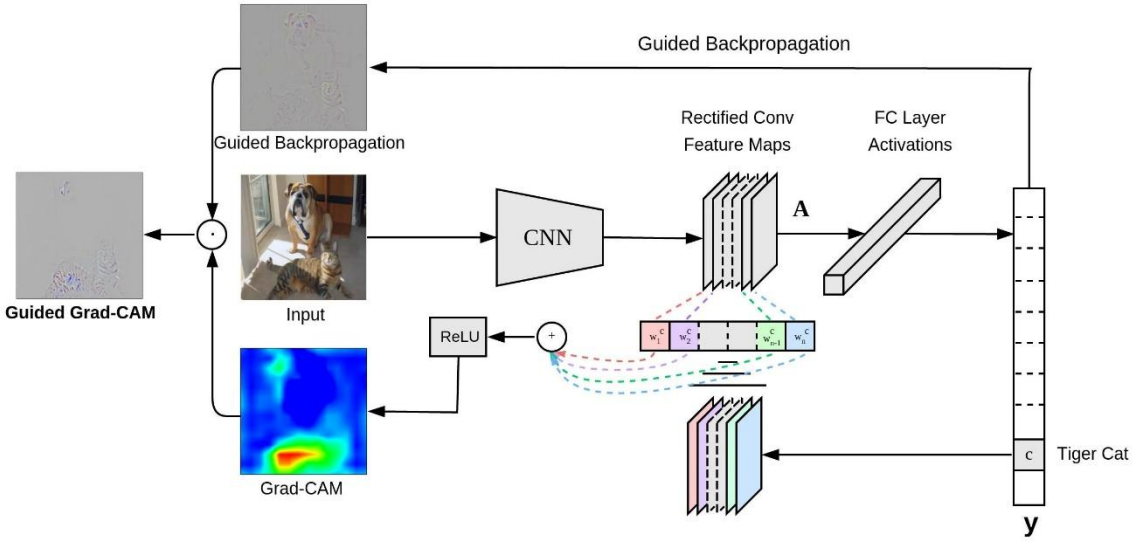
class	precision	recall	f1-score	support
0	0.67	0.69	0.68	417
1	0.64	0.50	0.56	14
2	0.92	0.65	0.76	17
3	0.62	0.69	0.65	600
4	0.77	0.79	0.78	600
5	0.78	0.68	0.73	242
6	0.75	0.60	0.67	72
7	0.69	0.67	0.68	336
8	0.75	0.77	0.76	209
9	0.56	0.43	0.48	600
10	0.47	0.34	0.40	140
11	0.68	0.63	0.65	201
12	0.68	0.64	0.66	600
13	0.70	0.68	0.69	192
14	0.65	0.82	0.73	201
15	0.67	0.79	0.72	361
16	0.59	0.36	0.45	47
17	0.86	0.84	0.85	383
18	0.64	0.65	0.65	77
19	0.60	0.65	0.63	222
20	0.54	0.53	0.53	600
21	0.51	0.59	0.54	600
22	0.69	0.79	0.74	313
23	0.61	0.68	0.64	600
24	0.72	0.58	0.65	600
25	0.73	0.82	0.77	33
26	0.97	0.97	0.97	175
accuracy			0.66	8452
macro avg	0.68	0.66	0.67	8452
weighted avg	0.66	0.66	0.66	8452


Confusion Matrix - efficientnet_b3 (Test Set)
True Label
Predicted Label


Project Evolution: Baseline vs. Improved Model
Validation Accuracy
Epochs
+12.2% Accuracy Boost
Phase 1: ResNet50 (Baseline)
Phase 2: efficientnet_b3 (Improved)

Interpretability with Grad-CAM

The final milestone of this project is **Transparency**. We implemented **Grad-CAM** (Gradient-weighted Class Activation Mapping) to visualize the decision-making process of our CNN. This ensures the model is looking at artistic textures rather than background noise.



1. How Grad-CAM Works

Grad-CAM uses the gradients of any target concept (like 'Cubism') flowing into the final convolutional layer to produce a localization map.

- **Target Layer:** We targeted the last convolutional block of **EfficientNet-B3** (the most semantic layer).
- **Heatmap Generation:** Areas with the highest intensity (red/yellow) indicate the specific brushstrokes or patterns that influenced the model's prediction.

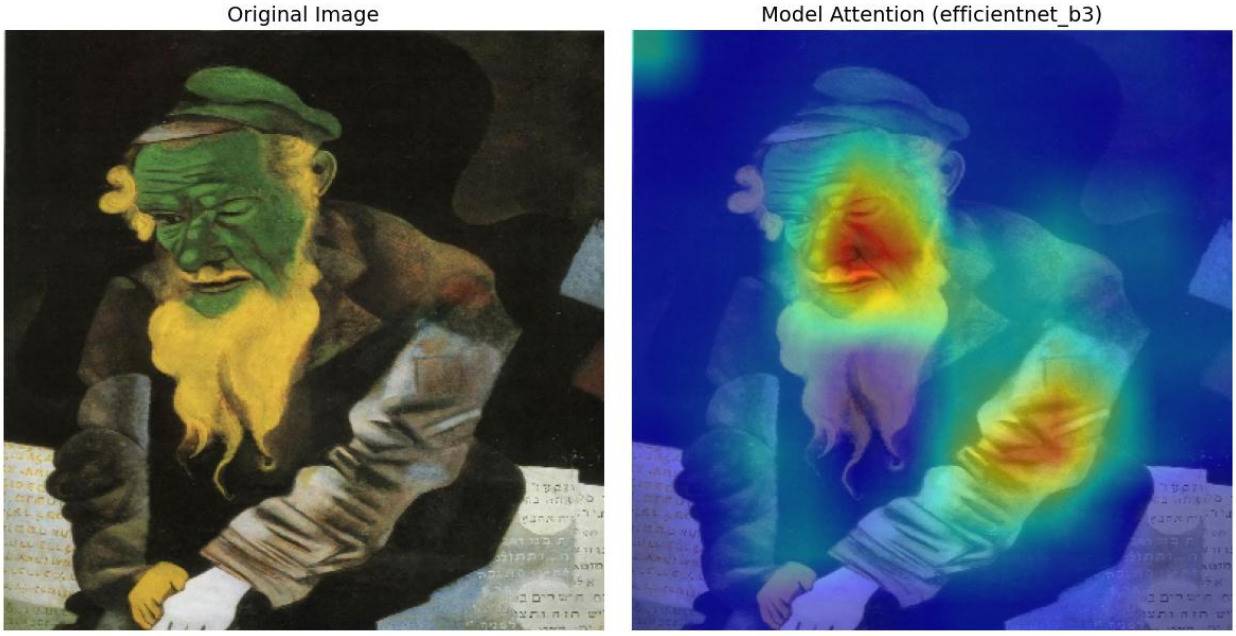
2. Qualitative Insights

By analyzing the Grad-CAM outputs in notebooks/6.0-Explainability-GradCAM.ipynb, we observed:

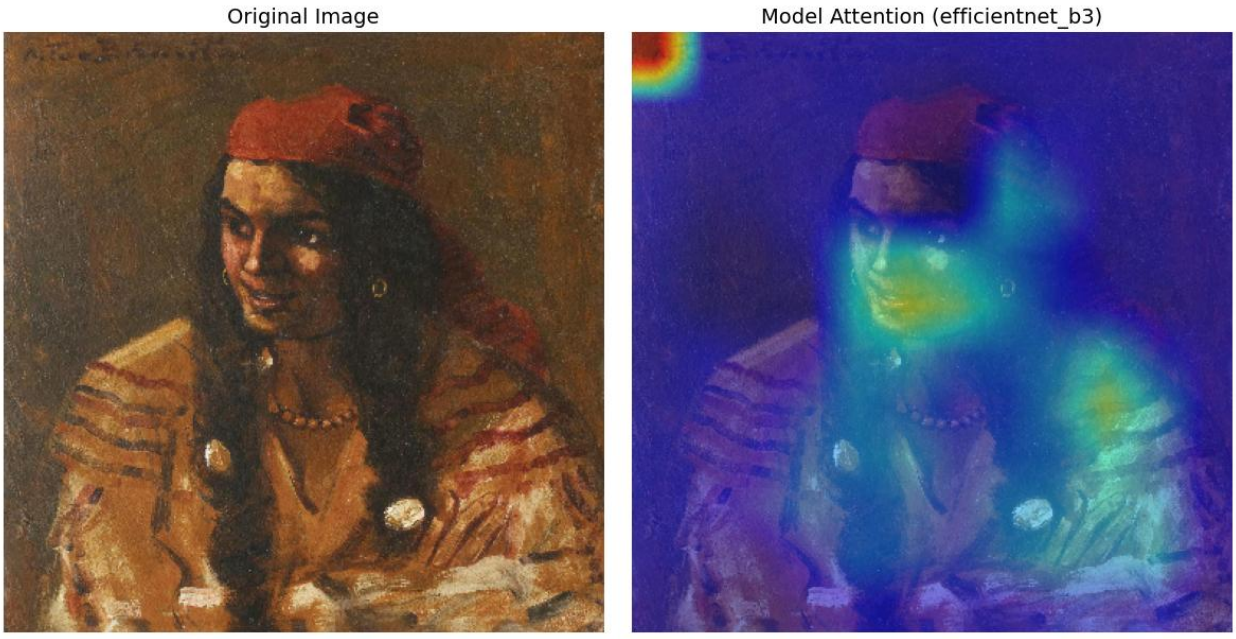
- **Impressionism:** The model focuses on small, disconnected color dabs.
- **Cubism:** The attention shifts to sharp geometric edges and fragmented planes.
- **Abstract Art:** The model analyzes the overall composition and color contrast rather than specific objects.

This step validates that our **Data Augmentation** (like Random Erasing) worked effectively. The model ignores occluded parts and still finds the relevant stylistic "signatures" in the visible sections of the canvas.

Testing style: Cubism | Confidence: 87.12%



Testing style: Impressionism | Confidence: 68.57%



Deployment Architecture — From Tensors to Production

The final stage of the project focused on **Operationalization**. We transformed the trained PyTorch models into a portable, user-facing system using a microservices approach, ensuring that the complex logic of art recognition is accessible through a clean interface and scalable via containerization.

1. The Inference Engine & Web Interface (src/app.py)

We utilized **Gradio** to build an interactive bridge between the user and the **EfficientNet-B3** backbone. The implementation focuses on real-time processing and probabilistic transparency.

1.1 Predictive Logic and Top-K Output

Artistic styles are often nuanced and overlapping. Our "predict" function does not just return a single label; it provides a probability distribution. Using torch.topk, the UI displays the **Top-3 most likely styles**, allowing users to see the model's confidence across similar genres.

1.2 Data Consistency (Preprocessing Pipeline)

To prevent "Distribution Shift," the application strictly mirrors the training transforms. Images are resized to **384px** and normalized using standard ImageNet statistics before being injected into the model.

2. Containerization: The Microservices Strategy

To manage the complexity of dependencies (Python, PyTorch, Node.js, and system libraries like libgl1), we implemented a **Multi-Container Docker Strategy**.

2.1 Backend Isolation (Dockerfile.api)

The backend is built on a python:3.10-slim image to keep the footprint small while including necessary computer vision libraries.

We separated the installation of torch to target the CPU index, ensuring the container remains lightweight and runs on servers without dedicated GPUs.

2.2 Frontend Portability (Dockerfile.ui)

The user interface uses a **Multi-Stage Build** process.

1. **Stage 1 (Node.js):** Compiles the React/HTML/CSS assets.
2. **Stage 2 (Nginx):** Discards the heavy Node environment and serves the static production files through an ultra-lightweight Nginx server. This results in a frontend container that is exceptionally fast and secure.

3. System Portability and Port Mapping

By decoupling the UI (Port 80) from the API (Port 5000), we created a modular system. This architecture allows us to update the model weights in the backend without ever taking the frontend offline, representing a professional **DevOps** approach to AI.

Automation & CI/CD Pipeline

The project's maturity is solidified by a **GitHub Actions** workflow, transforming the repository into an automated production pipeline. This ensures that every update is verified through a rigorous "Smoke Test."

1. Automated Validation (CI)

Our CI pipeline (ci.yml) automates the entire lifecycle on every push. It uses an **ubuntu-latest** runner to execute a high-speed simulation of the project:

- **Environment Setup:** Uses actions/setup-python@v5 and implements **Pip Caching** to reduce build times.
- **Synthetic Data Generation:** A built-in script (create_dummy.py) creates balanced, randomized datasets on the fly, allowing for full pipeline testing without downloading the massive WikiArt dataset.

- **Execution Testing:** 1. **Training:** Runs fine_tune.py (ResNet50) for 1 epoch. 2. **Evaluation:** Executes evaluate.py to verify metric generation. 3. **Integrity:** Validates model loading for the API.

2. Infrastructure as Code

The CI ensures **"Code-to-Model" integrity**. If a change in the model architecture breaks the training loop, the CI fails, protecting the main branch.

- **Artifacts:** Successful evaluation results (JSON & Plots) are automatically uploaded as **GitHub Artifacts** via actions/upload-artifact@v4.
- **Reproducibility:** The workflow forces a clean install of dependencies (PyTorch CPU-version), mirroring the **Docker** environment.

Conclusion: The Style-Content Dilemma & Future Horizon

The Core Challenge: Semantic vs. Stylistic Bias

The fundamental difficulty in Art Style Recognition lies in the way **Convolutional Neural Networks (CNNs)** perceive images. CNNs are naturally biased toward **Semantic Content** (objects) rather than **Stylistic Patterns** (textures).

The "Dog vs. Building" Paradox:

- **Scenario A:** Image 1 is a "Dog" in *Style A*. Image 2 is a "Dog" in *Style B*.
- **Scenario B:** Image 1 is a "Dog" in *Style A*. Image 3 is a "Building" in *Style A*.

Because the model looks for shapes and features, it is mathematically "tempted" to group Image 1 and Image 2 together because they both contain a "Dog". However, our goal is to group Image 1 and Image 3 together because they share the same *Artistic Style*.

Breaking this "Object Bias" is what makes this project complex. Achieving **over 60% accuracy across 27 classes**—where random guessing would yield only **3.7%**—proves that our model has successfully learned to prioritize **textures, brushstrokes, and color palettes** over the literal objects in the paintings.

Solution: Hybrid Content-Aware Architectures

If we were to extend this research, the next step would be to move toward a **Hybrid Contrastive Learning** model.

The goal would be to develop a system that explicitly understands when two images are semantically similar (e.g., both are dogs) but stylistically different.

- **Dual-Stream Inference:** A model that extracts "Content Embeddings" and "Style Embeddings" separately.
- **Differential Focus:** If the model detects two images are close in *Content*, it should automatically "ignore" those shared features and pivot its attention to the subtle differences in texture and stroke.

By forcing the model to maneuver through these "Content Traps," we can push the accuracy even further, creating a system that truly understands the soul of the art, not just the subject of the painting.

Final Project Verdict

This project demonstrates that through **Surgical Fine-Tuning** and **Advanced Regularization**, we can train a machine to perceive the "how" (Style) rather than just the "what" (Content), marking a significant step in the intersection of Artificial Intelligence and Art History.