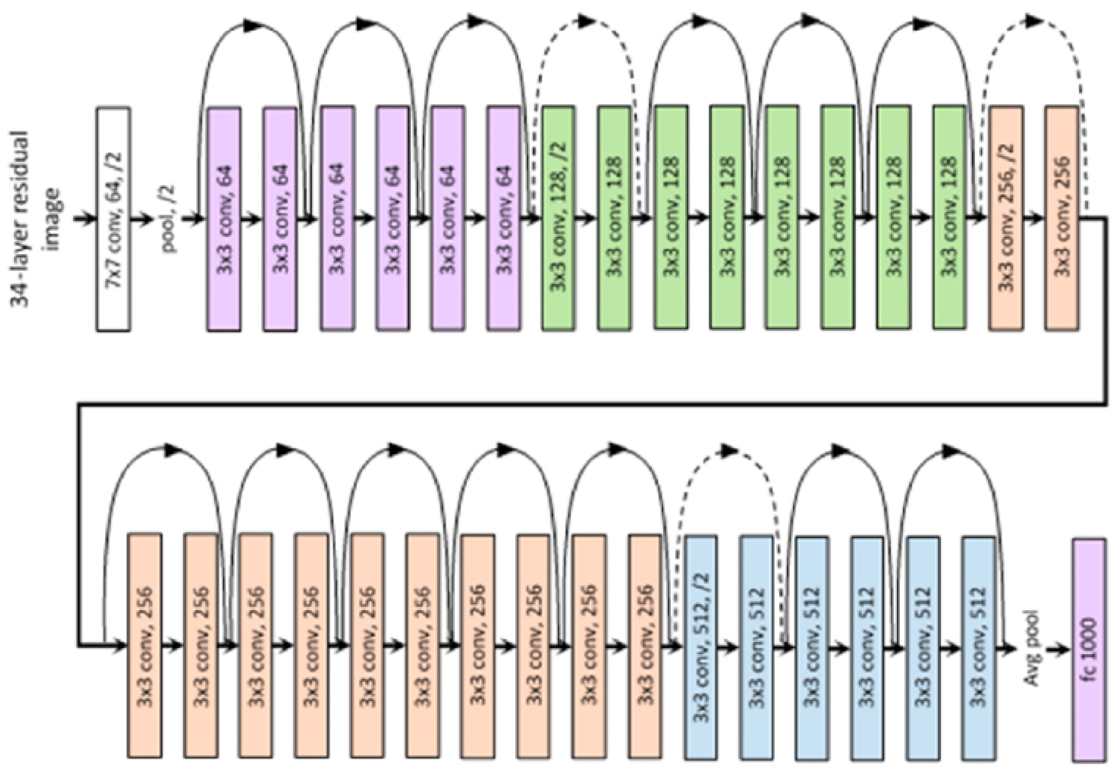
Neural network and deep learning

1-ResNet Implementation

Architecture:

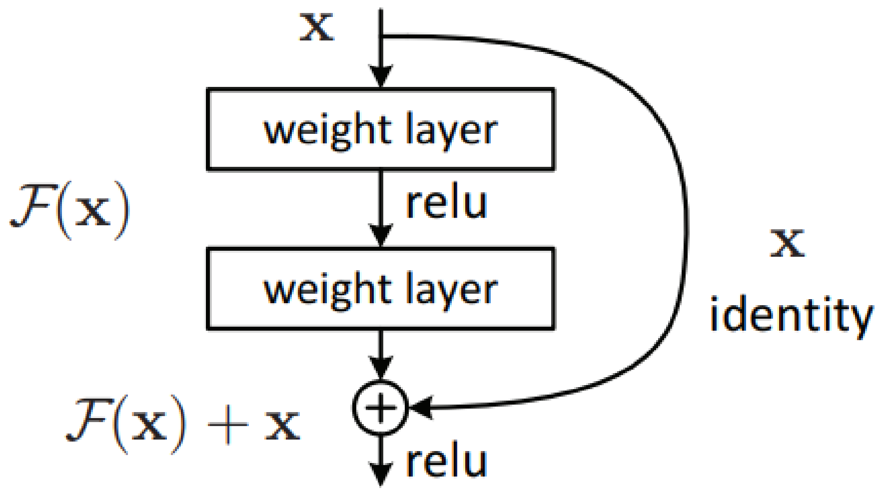
- ResNet introduces skip connections (also known as residual connections) that allow the input to bypass certain layers, reducing the number of layers the gradient must propagate through.
- In order to solve the problem of the vanishing/exploding gradient, this architecture introduced the concept called Residual Blocks.
- **Graph**: Include a block diagram showing residual connections.

(a)



(b)

-



The advantage of adding this type of skip connection is that if any layer hurt the performance of architecture then it will be skipped by regularization

Step-by-step details:

- **Input**: A 224x224 RGB image.
- **First Layer**: Convolution with a 7x7 kernel, followed by max-pooling.
- **Residual Blocks**: Repeated 2-3 times for deeper networks (ResNet-50, ResNet-101).
- **Final Layers**: Global average pooling followed by a fully connected layer and softmax for classification.

Step 1: Understand the Architecture

- **Initial Layers:**
  - Input image size: 224×224×3224 \times 224 \times 3224×224×3.
  - Convolution: 7×77 \times 77×7, stride 2, followed by BatchNorm, ReLU, and MaxPooling.
- **Residual Layers:**
  - Four stages of bottleneck blocks:
    - Stage 1: 333 blocks with 646464 filters.
    - Stage 2: 444 blocks with 128128128 filters.

        - Stage 3: 666 blocks with 256256256 filters.
        - Stage 4: 333 blocks with 512512512 filters.
    - Each block uses skip connections.
- **Final Layers:**
    - Global Average Pooling.
    - Fully Connected Layer with softmax for classification.

Step 2: Define Bottleneck Block

- Implement the bottleneck block using:
    - 1×11 \times 11×1 convolution (for dimension reduction).
    - 3×33 \times 33×3 convolution (feature extraction).
    - 1×11 \times 11×1 convolution (dimension restoration).
    - Add a skip connection if input and output dimensions differ.

Step 3: Build ResNet-50

1. Stack the bottleneck blocks according to the ResNet-50 architecture.
2. Add the initial layers and the final classification layers.
3. Use downsampling in residual blocks when spatial dimensions reduce.

Step 4: Load Dataset

- Load a dataset such as CIFAR-10 or ImageNet.
- Preprocess images (resize to 224×224224 \times 224224×224, normalize, and augment).

Step 5: Train the Model

1. Define the loss function (e.g., CrossEntropyLoss for classification).

2. Choose an optimizer (e.g., SGD or Adam).
3. Train the model for several epochs and monitor the loss and accuracy.

Step 6: Evaluate the Model

- Test the trained model on a validation or test set.
- Measure performance using metrics like accuracy or F1 score.

Implementation:

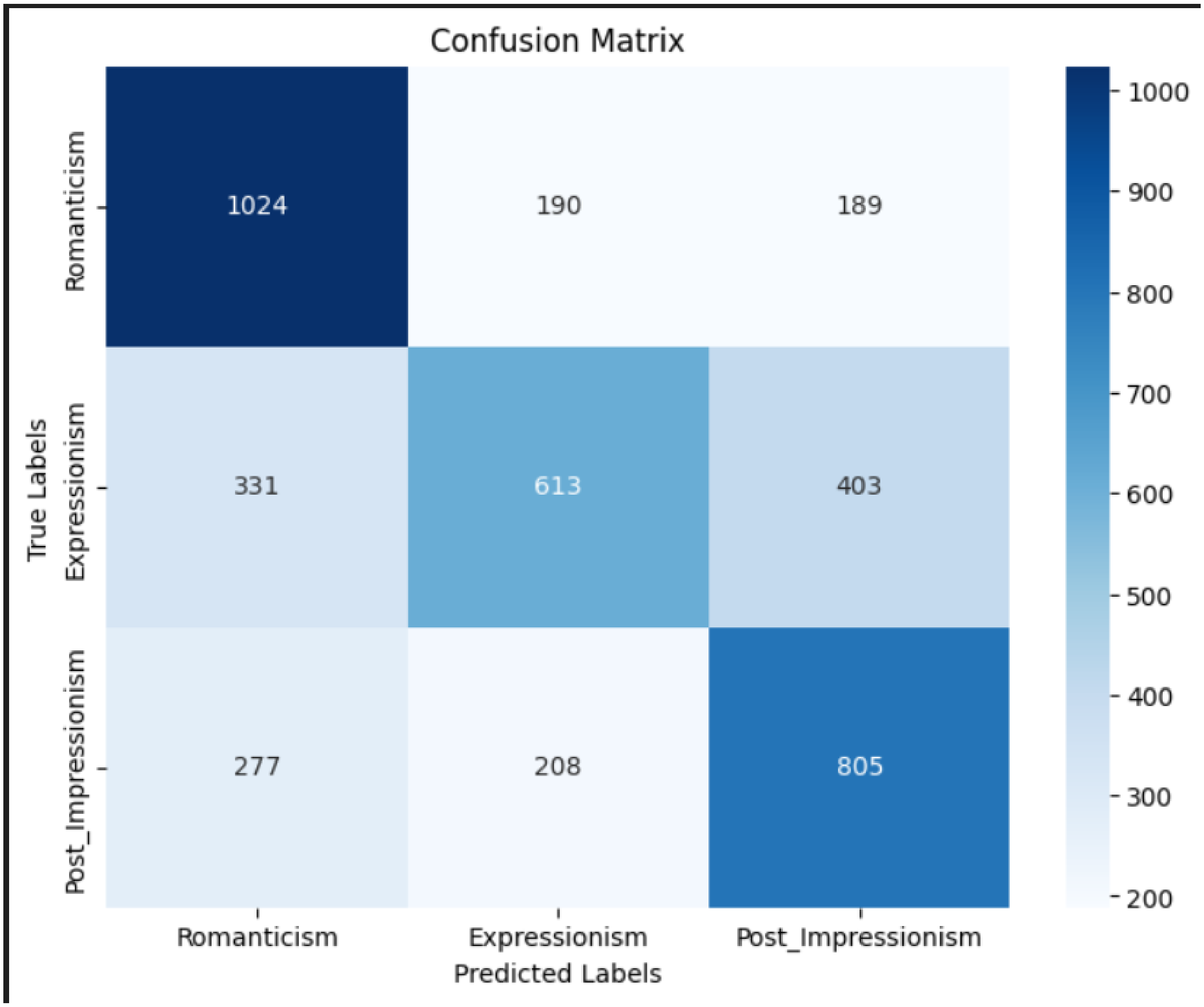
Using the Tensorflow and Keras API, we can design ResNet architecture (including Residual Blocks) from scratch. Below is the implementation of different ResNet architecture. For this implementation, we use the CIFAR-10 dataset. This dataset contains 60, 000 32×32 color images in 10 different classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks), etc. This dataset can be assessed from k*eras.datasets* API function.

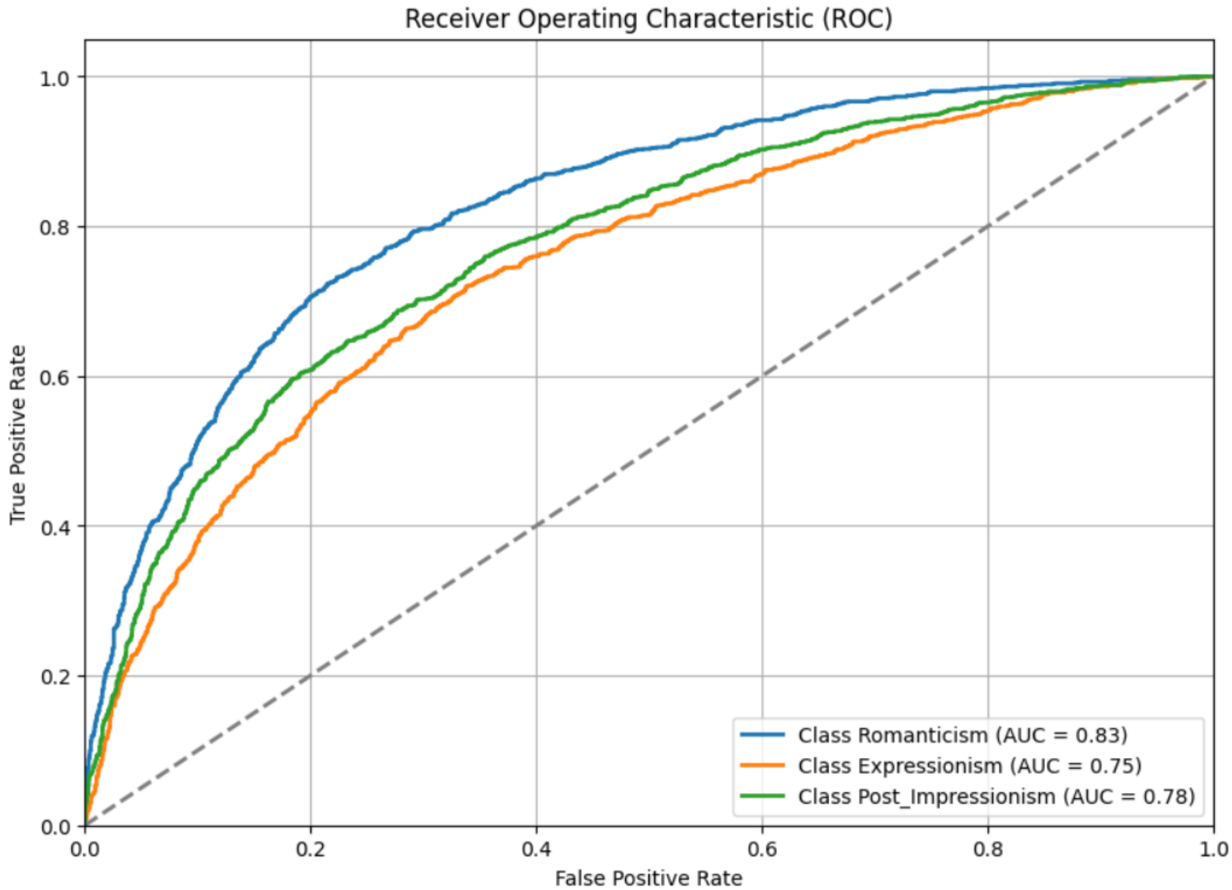
Results for your models (accuracy with visualization, loss curve with visualization, confusion matrix with visualization, recall, precision, f-score, ROC, AUC graph)

```
Classification Report:
                    precision    recall  f1-score   support

       Romanticism       0.63      0.73      0.67      1403
     Expressionism       0.61      0.46      0.52      1347
Post_Impressionism       0.58      0.62      0.60      1290

          accuracy                           0.60      4040
         macro avg       0.60      0.60      0.60      4040
      weighted avg       0.60      0.60      0.60      4040
```


Confusion Matrix
True Labels
Romanticism
Expressionism
Post_Impressionism
1024
190
189
331
613
403
277
208
805
Romanticism
Expressionism
Post_Impressionism
Predicted Labels
1000
900
800
700
600
500
400
300
200

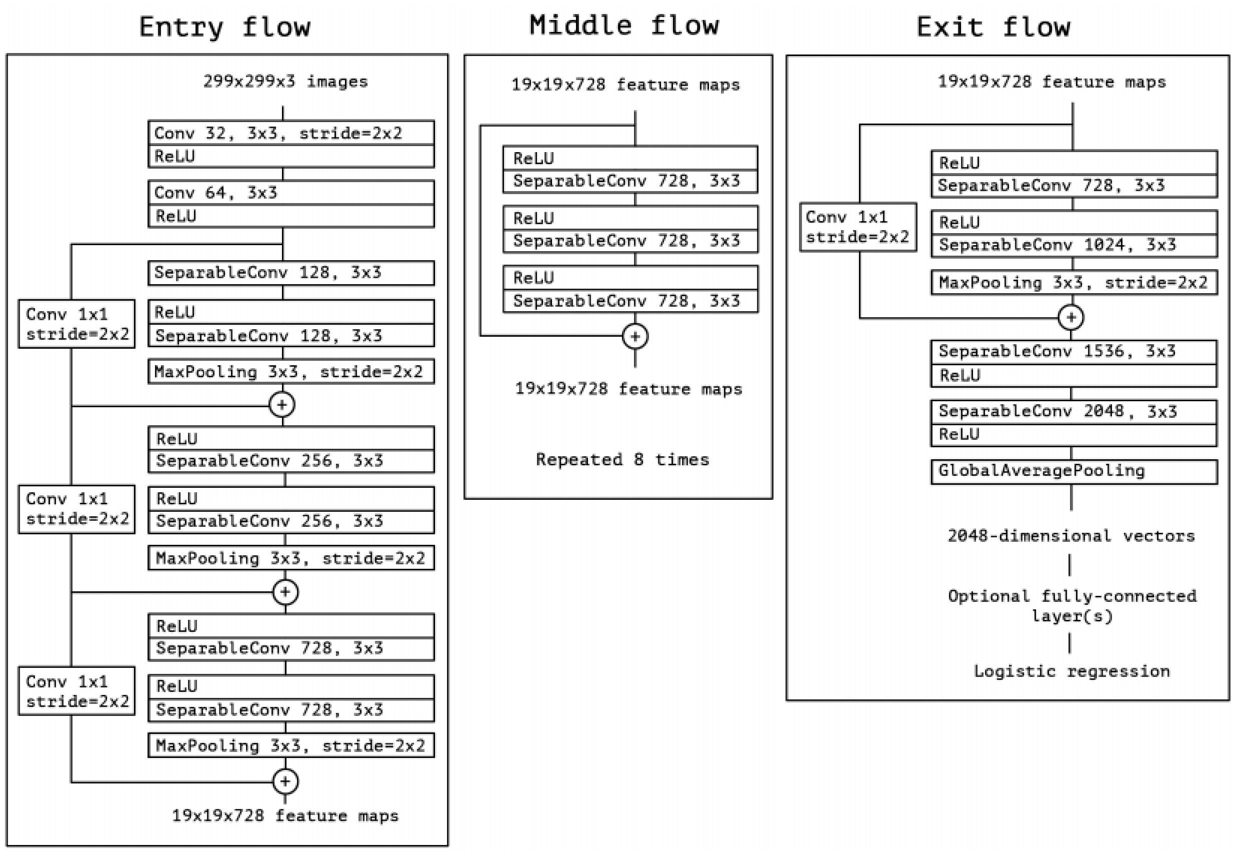


/***/

2-Xception Finetuning

Architecture:

- Xception consists of a series of depthwise separable convolutions (separating spatial and channel convolutions) that help improve the efficiency of convolutions.
- **Graph**: Show how depthwise separable convolutions are implemented in Xception.



Step-by-step details:

- **Input**: Image resizing to 299x299.
- **Initial Layers**: Standard convolution followed by depthwise separable convolutions.
- **Final Layer**: Fully connected layer followed by softmax for classification.

  1-Load Pre-Trained Model:

  - Use the Xception model with weights pre-trained on the ImageNet dataset.
  - Exclude the top layers to modify the architecture according to our needs.

  2-Freeze Layers:

  - Freeze the base model layers to retain the learned weights from ImageNet training and only train the new top layers.

  3-Custom Classification Head:

  - Add a GlobalAveragePooling2D layer to reduce the output dimensions.

- Use the Dense layer with a softmax activation function for multi-class classification across the selected art styles.

4-Compile the Model:

- Use the Adam optimizer and categorical cross-entropy loss function to train the model.

Implementation:

- Load a pre-trained Xception model and fine-tune it using a smaller learning rate, typically freezing the initial layers and training the last few for your dataset.

Papers:

[Xception: Implementing from scratch using Tensorflow | by Arjun Sarkar | Towards Data Science]

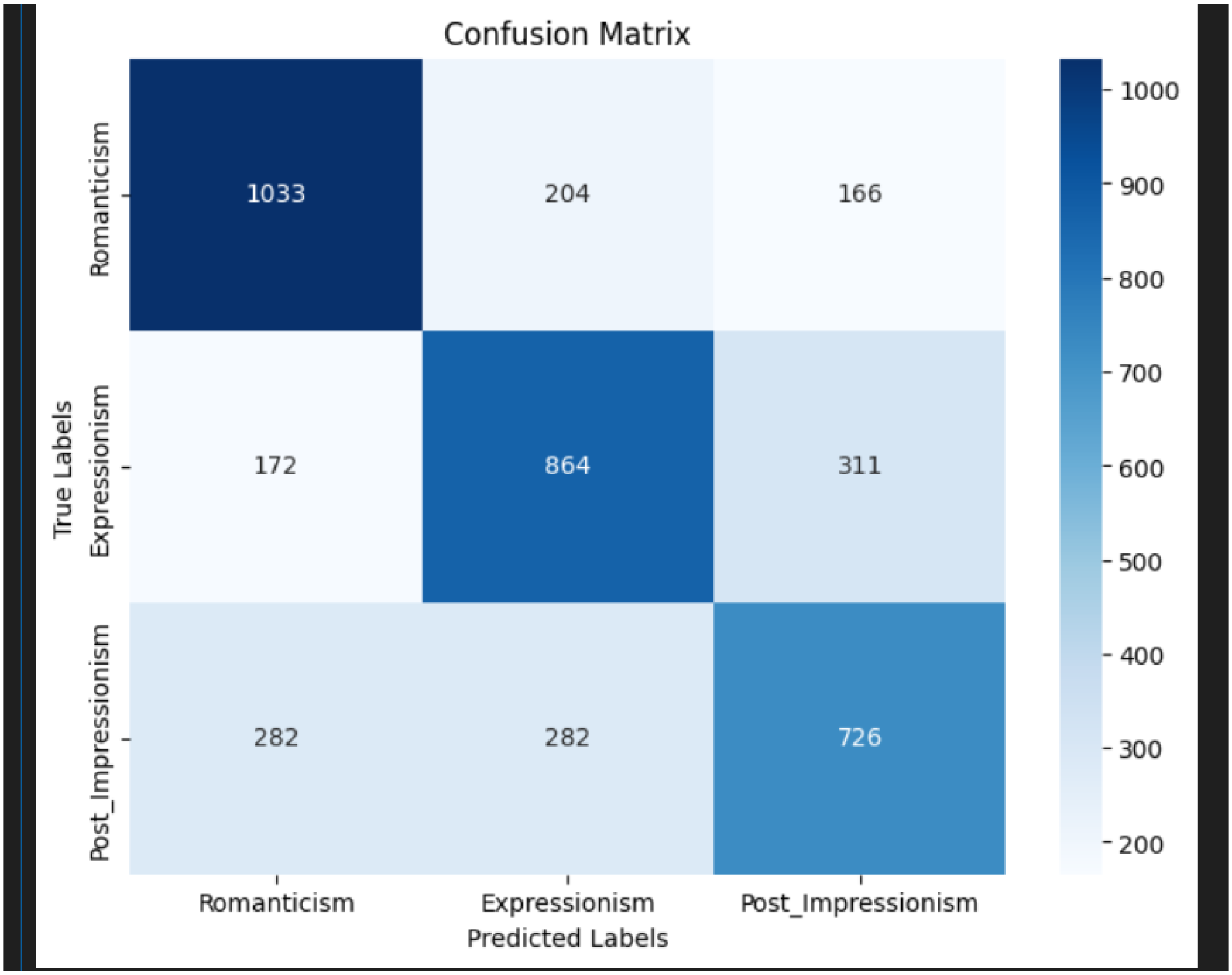
[[1610.02357] Xception: Deep Learning with Depthwise Separable Convolutions]

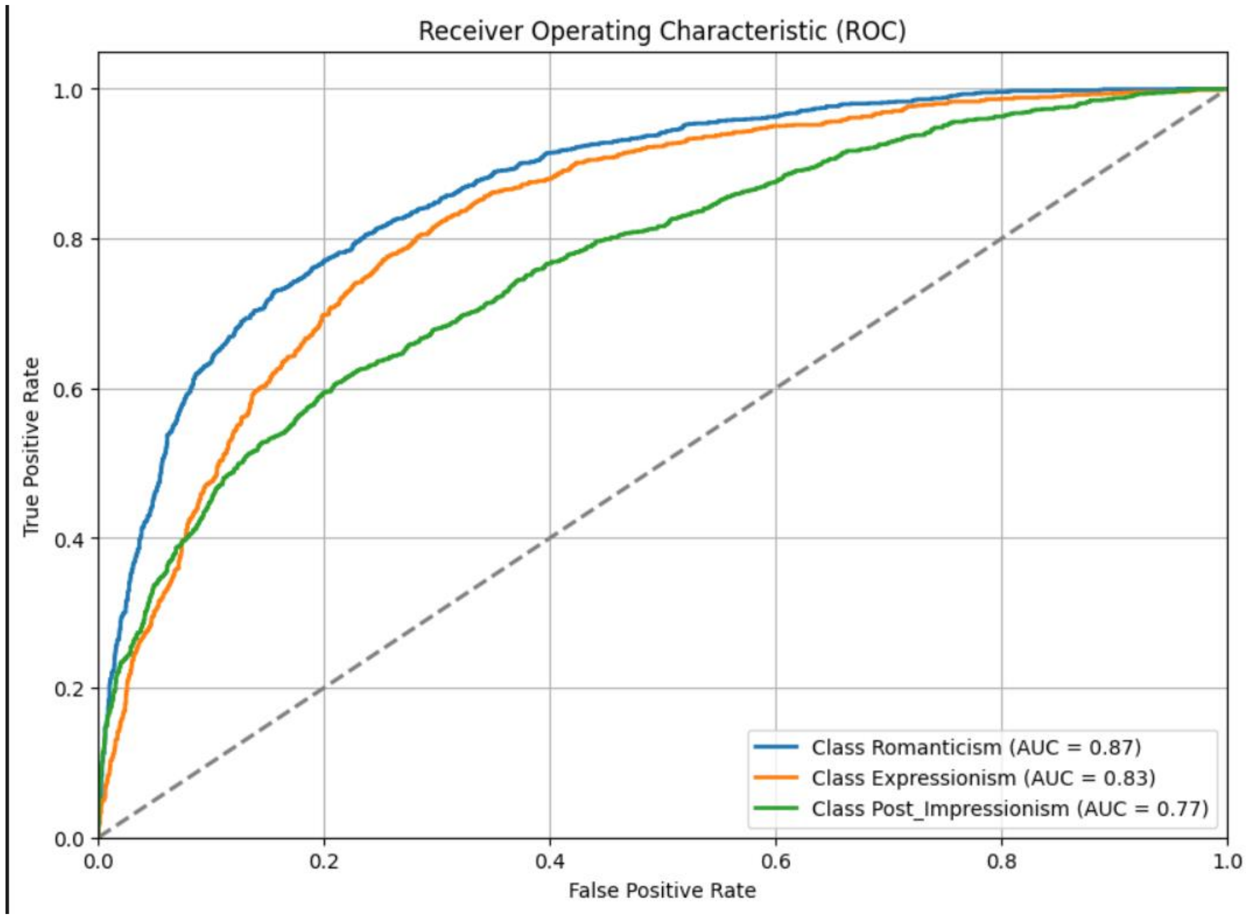
Results for your models (accuracy with visualization, loss curve with visualization, confusion matrix with visualization, recall, precision, f-score, ROC, AUC graph)

```
Classification Report:
                    precision    recall  f1-score   support

       Romanticism       0.69      0.74      0.71      1403
     Expressionism       0.64      0.64      0.64      1347
Post_Impressionism       0.60      0.56      0.58      1290

          accuracy                           0.65      4040
         macro avg       0.65      0.65      0.65      4040
      weighted avg       0.65      0.65      0.65      4040
```


Confusion Matrix
Romanticism
Expressionism
Post_Impressionism
True Labels
1033
204
166
172
864
311
282
282
726
Romanticism
Expressionism
Post_Impressionism
Predicted Labels
1000
900
800
700
600
500
400
300
200

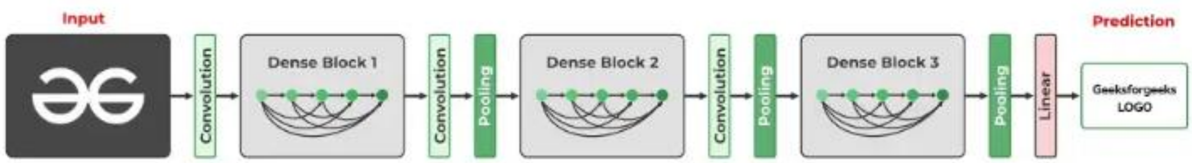


/***/

3-DenseNet Finetuning

Architecture:

- Each layer receives input from all preceding layers, making DenseNet unique in its approach to connectivity.
- DenseNet establishes direct connections between all layers within a block. This dense connectivity enables each layer to receive feature maps from all preceding layers as inputs, fostering extensive information flow throughout the network.
- **Graph**: Diagram showing dense connections between layers.



Step-by-step details:

- **Input**: Image resizing to 224x224.
- **Initial Layers**: Standard convolution followed by dense blocks where each layer is connected to all previous ones.
- **Final Layers**: Global average pooling followed by softmax classification.

  1-Load Pre-trained DenseNet

  2-Unfreeze the last few layers of the base model

  3-Add custom classification head with adjusted regularization parameters

  4-Create the full model

  5-Compile the model

Implementation:

- Use a pre-trained DenseNet (e.g., DenseNet-121) and fine-tune it similarly by freezing the initial layers and adjusting the later layers for your dataset.

Papers:

[DenseNet Explained - GeeksforGeeks]()

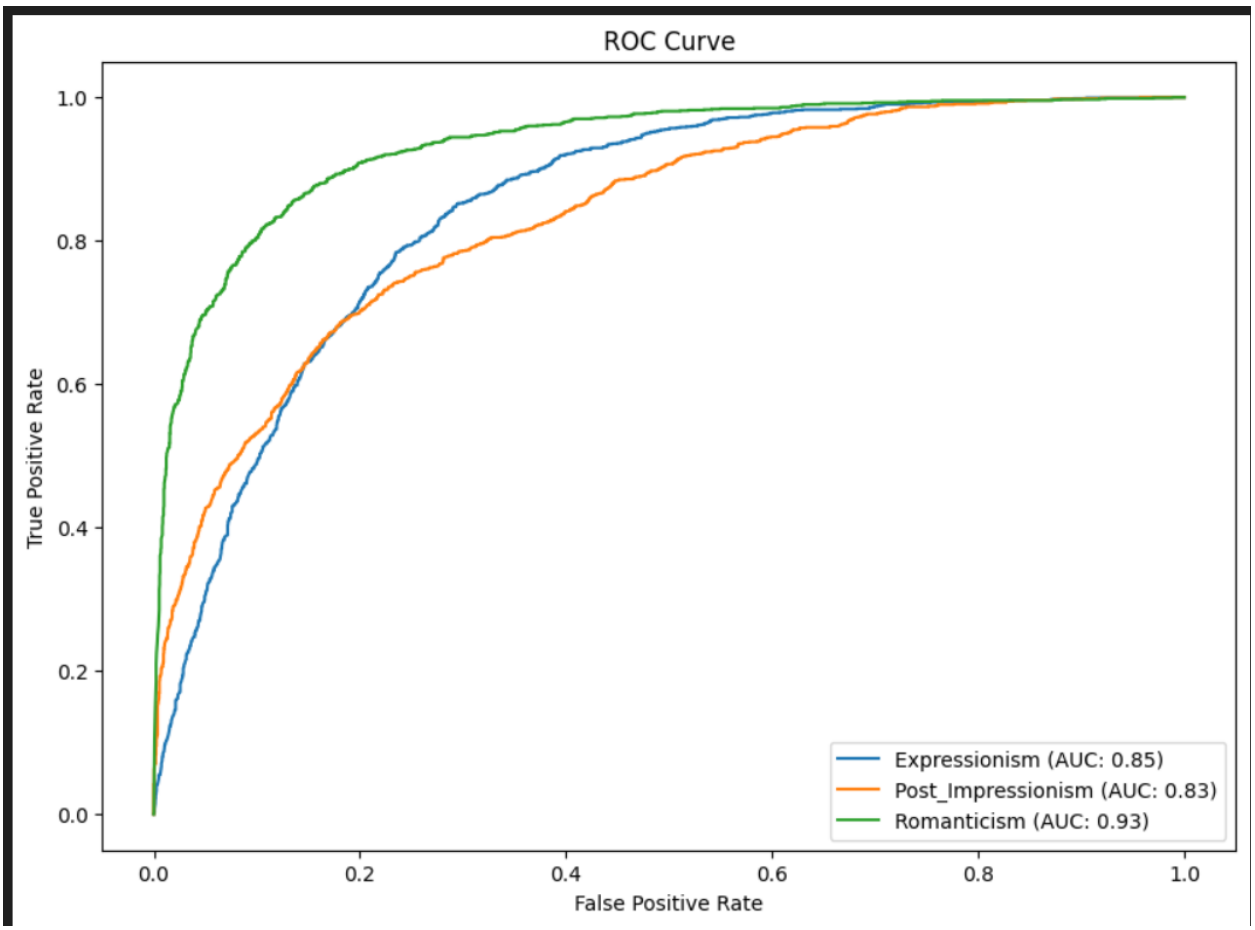
[1608.06993] Densely Connected Convolutional Networks

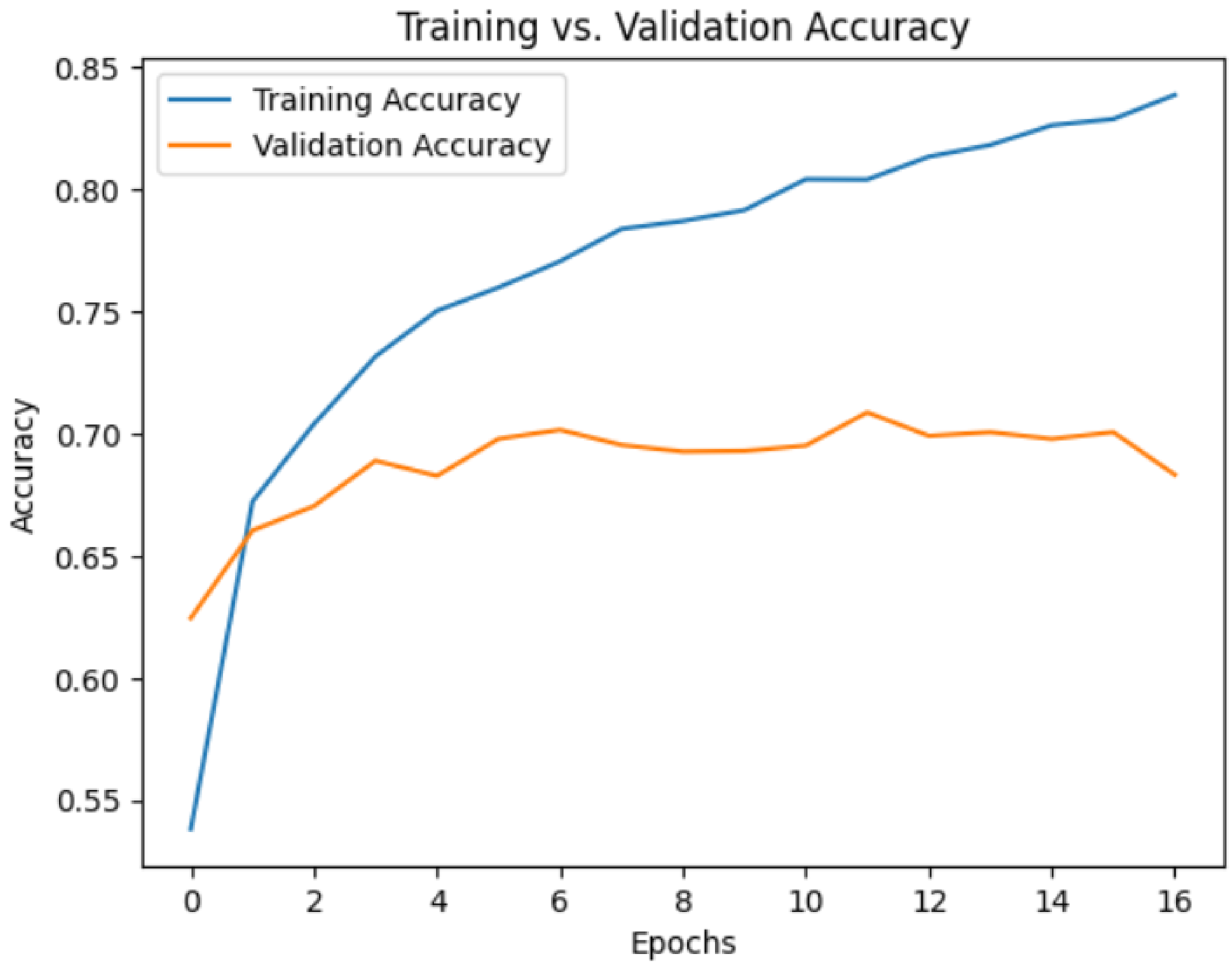
Results for your models (accuracy with visualization, loss curve with visualization, confusion matrix with visualization, recall, precision, f-score, ROC, AUC graph)

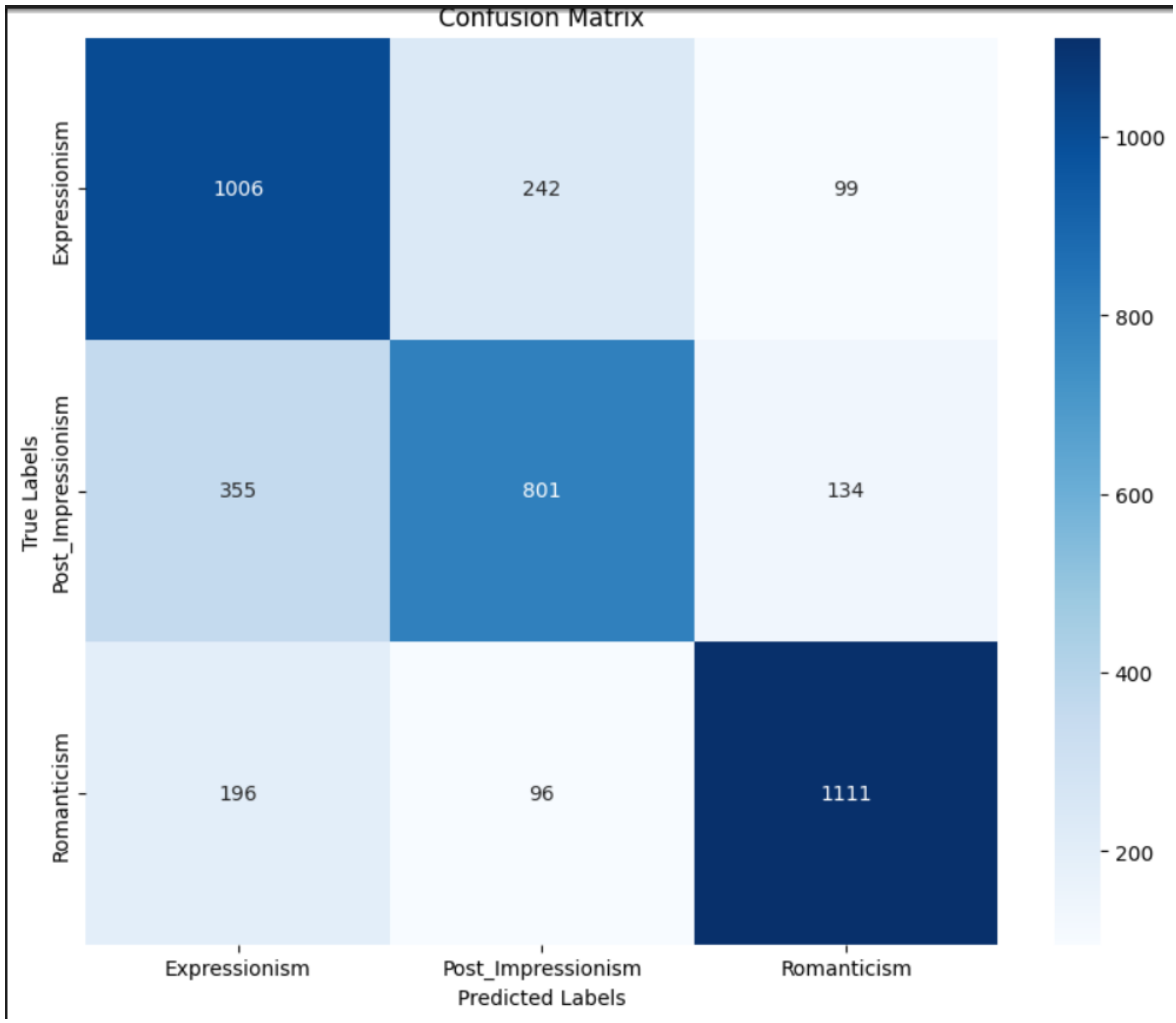
```
Classification Report:
                    precision    recall  f1-score   support

     Expressionism       0.65      0.75      0.69      1347
Post_Impressionism       0.70      0.62      0.66      1290
       Romanticism       0.83      0.79      0.81      1403

          accuracy                           0.72      4040
         macro avg       0.73      0.72      0.72      4040
      weighted avg       0.73      0.72      0.72      4040
```




Training vs. Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0
2
4
6
8
10
12
14
16
Epochs



/***/

Aspect	ResNet	Xception	DenseNet
Architecture Type	Deep residual networks with skip connections	Depthwise separable convolutions	Densely connected layers (each layer connects to all previous layers)

Key Innovation	Residual learning to avoid vanishing gradients	Depthwise separable convolutions for efficiency	Dense connections for feature reuse and gradient flow
Performance	Excellent for very deep networks, high accuracy	Fast inference with fewer parameters, efficient	Excellent feature reuse and gradient flow, good for fine-grained classification
Training Complexity	Can be computationally expensive for very deep models	Lower computational cost due to separable convolutions	Memory-intensive due to dense connections, but fewer parameters
Pros	- Solves vanishing gradient problem - Effective for large datasets - Flexible with depth	- Efficient with fewer parameters - Fast inference - Good for resource-constrained environments	- Maximizes feature reuse - Good for tasks requiring fine-grained details - Better gradient flow
Cons	- Computationally expensive for deeper networks	- May underperform on simpler tasks	- Memory-intensive due to dense connections
Best Use Case	Large-scale image classification (e.g., ImageNet)	Efficient image classification with limited resources	Tasks that benefit from feature reuse (e.g., segmentation, fine-grained classification)
Accuracy (example)	High for large datasets (ResNet-50, ResNet-101)	High, especially for efficient inference	High, particularly for tasks requiring deep feature extraction
result			Validation Accuracy: 0.7081683278083801