Deep Learning Framework for Emotional Valence Prediction Using Popular CNN Architectures and Efficient Data Generation

Reuben O Jacob, Sujay Anantha, Tejas Hiremutt

Department of Electrical and Computer Engineering, Tandon School of Engineering
New York University
6 MetroTech Center, Brooklyn, NY 11201
{roj2009, sa7055, th3038}@nyu.edu

Abstract

This paper proposes a deep learning-based framework for Emotional Valence Prediction in images using popular CNN architectures including VGG16, ResNet50, and InceptionV3. This paper utilized the ArtEmis dataset. The framework utilizes pre-processing procedure to convert the dataset(image data and labels) stored as compressed NumPy arrays on Google Drive. In this implementation, a CNN model is constructed by combining a selected base model architecture with custom classification layers. The base model is initialized with pre-trained weights from the ImageNet dataset and its layers are frozen to preserve the learned features during training. Niche techniques have been implemented like, early stopping to prevent overfitting and compute the validation accuracy and a classification report for performance evaluation. The framework further includes a model selection mechanism that iterates through the specified model names and tracks the best-performing model based on validation accuracy. The framework allows researchers to easily experiment with different CNN architectures, leverage pre-trained models, and assess model performance using appropriate evaluation metrics. Overall, this framework provides a valuable tool for deep learning-based emotional valence prediction research and its applications.

Introduction

The recognition of Emotional Valence and understanding from visual content, particularly images, has gained significant attention in recent years due to its wide range of applications in various fields, including human-computer interaction and social robotics. The ability to accurately predict emotional valence from images can provide valuable insights into human affective states and facilitate personalized experiences and interactions. In this paper, we propose a deep learning-based framework for predicting emotional valence from images. Emotional valence refers to the positive or negative sentiment associated with an emotion, which plays a crucial role in understanding human affective responses. Our framework utilizes popular convolutional neural network (CNN) architectures, including VGG16, ResNet50, and InceptionV3, as the backbone models. By leveraging pre-trained weights obtained from large-scale image datasets, such as ImageNet, we can benefit from the learned representations and transfer the knowledge to our emotional valence prediction task. To train and evaluate the models, we employ a dataset of images labelled with emotional valence scores. The data is preprocessed and stored as NumPy arrays, allowing for efficient loading and processing during model training. We utilize a data generator function to generate batches of images and labels, enabling memory-efficient training and avoiding potential bottlenecks. In addition to model training, our framework incorporates essential components such as early stopping to prevent overfitting and evaluation metrics to assess the model's performance. We calculate the validation accuracy and generate a comprehensive classification report to analyze the model's predictive capabilities. We further investigate the performance of different CNN architectures and identify the best-performing model. The insights gained from this research can contribute to advancing the field of affective computing and pave the way for applications that require real-time emotion recognition from visual content. Overall, our proposed deep learning framework provides a valuable contribution to the field of emotional valence prediction from images. By combining powerful CNN architectures, efficient data handling, and comprehensive evaluation methods, we strive to enhance the understanding of human emotions and open up possibilities for a wide range of applications where emotion recognition plays a significant role.

The repository of the project can be found here: Repository

Related Work

In the study (3), a deep learning-based approach is proposed for continuous affect prediction from facial expression images. The authors focus on predicting emotional valence and arousal in the valence-arousal space, utilizing a deep convolutional neural network (CNN) architecture. The results demonstrate the effectiveness of their approach in accurately predicting affective states from facial expressions. (4) discusses the extraction of moods from pictures and sounds to enable personalized television experiences. The study highlights the potential of using machine learning and signal-processing techniques to extract moods from multimedia content. By understanding the emotional content of media,

personalized TV systems can adapt and deliver tailored content based on individuals' preferences and emotional states. The study by Cowen and Keltner (2017)([5]) demonstrates that self-report data can effectively capture 27 distinct categories of emotion, providing valuable insights for the field of emotional valence prediction using deep learning frameworks.

Methodology

In this paper, we first preprocessed the ArtEmis data using the $preprocess - artemis - data.py$ script that was provided by the authors of the dataset from ([1]). The Artemis dataset is a curated collection of images specifically designed for emotion recognition and analysis tasks. The Artemis dataset is a curated collection of images specifically designed for emotion recognition and analysis tasks, it offers several key characteristics that make it particularly useful for emotion recognition research. Firstly, the dataset covers a wide range of emotions, including both positive and negative valence. This allows for comprehensive analysis and prediction of emotional states. Secondly, the dataset includes a large number of images, ensuring a substantial training and evaluation set for model development. This facilitates robust training and reliable performance assessment.
A snippet of the dataset is given in Figure 1 as seen in ([2]).

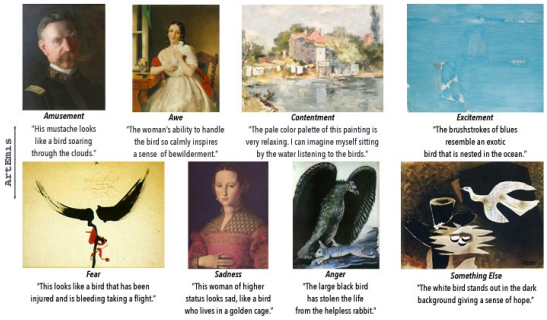


Figure 1: ArtEmis Dataset.

Pre-Processing:
The script for preprocessing the ArtEmis dataset consists of the following steps:

1. The required libraries and modules are imported to ensure the availability of necessary functionalities. The paths to the frequency file and the GloVe file are defined. These files are used for spell-checking and tokenization.
2. A function is defined to parse the command-line arguments. This allows the user to provide input parameters when running the script. Another function is defined to group the ground truth annotations per artwork. This grouping is useful for further analysis and evaluation.
3. The raw ArtEmis dataset is loaded from a CSV file. Artworks with high repetition coverage are handled based on the specified threshold. This step allows for excluding artworks with an excessive number of annotations.
4. The data is split into train, validation, and test sets. The splitting is done based on the unique combinations of artwork and painting. Optionally, some masterpieces can be placed in the test set. These artworks serve as notable examples for showcasing purposes. An optional sub-selection of examples from the dataset can be performed. This is useful for quick debugging or testing purposes. Spell checking is applied to the dataset using SymSpell, a spell-checking library. This helps correct misspelled tokens.
5. Utterances that are too short or too long are dropped based on specified thresholds. This filtering ensures that the dataset contains utterances of reasonable length. A vocabulary is built based on the tokens in the training data. The vocabulary consists of unique words in the dataset.
6. Tokens are encoded as integers using the vocabulary. This step replaces each word with its corresponding integer representation. Emotions are encoded as integers. Each emotion label is mapped to a unique integer value. The preprocessed dataframe, vocabulary, and missed tokens (tokens not found in the vocabulary or GloVe embeddings) are returned.
7. In the main block of the script, the command-line arguments are parsed using the defined function. The NLTK tokenizer is checked and downloaded if necessary. This tokenizer is used for tokenization in the spell-checking process.
8. The preprocess function is called with the provided command-line arguments. This initiates the preprocessing pipeline. The preprocessed dataframe, vocabulary, and configuration file are saved to the specified output directory. These files can be used for further analysis or model training.

The Classification Report of the Model is as Follows:

	precision	recall	f1-score	support
0	0.21	0.17	0.19	672
1	0.00	0.00	0.00	88
2	0.22	0.06	0.09	1217
3	0.28	0.90	0.43	1871
4	0.00	0.00	0.00	352
5	0.47	0.01	0.02	650
6	0.29	0.05	0.09	712
7	0.33	0.00	0.00	774
8	0.21	0.06	0.09	832
accuracy			0.27	7168
macro avg	0.22	0.14	0.10	7168
weighted avg	0.26	0.27	0.17	7168

Validation accuracy: 27.34%

Figure 2: Classification Report : VGG16 Model.

Data Generators:
To feed the data into the model, a separate Data Generator file was used. It is designed to preprocess the ArtEmis dataset and generate NumPy arrays for training deep learning models. The ArtEmis dataset contains annotations and emotions associated with various artworks. This data generator code serves as a crucial step in preparing the ArtEmis dataset for training deep learning models to analyze and un-

derstand the emotional content of artworks.
The working of this function is as follows:

1. The paths to the dataset directory (wikiart-dir) and the preprocessed ArtEmis data file are defined. The ArtEmis data is loaded from the preprocessed CSV file into a Pandas dataframe (artemis-data).
2. The emotion labels are encoded using the LabelEncoder from Scikit-learn, which assigns a unique integer to each emotion class. The image size is defined as 224x224 pixels. The load-image function is defined to read and preprocess an image file. It uses TensorFlow functions to read the file, resize the image, convert the color space, and normalize the pixel values.
3. The generate-numpy-arrays function takes a dataframe as input and generates the NumPy arrays for images and labels. It initializes the data array with zeros and converts the emotion labels to one-hot-encoded labels. It then uses parallel processing to load and preprocess the images, storing them in the data array. Finally, it converts the labels to float32 and returns the data and labels arrays.
4. The ArtEmis data is split into train, validation, and test sets using the train-test-split function from Scikit-learn. The train set is further split into a reduced train set (train-df-reduced) and a validation set (val-df-reduced) due to computational constraints.
5. Since we are using Colab Pro+ to run the model, a high amount of the dataset could not be used. Hence to effectively run three models using limited RAM, we have used only **5 percent** of the dataset.
6. The number of samples in each dataset is printed to verify the sizes. Reduction ratios are defined for each dataset (train-frac, val-frac, test-frac) to reduce the number of samples.
7. Reduced datasets are created by randomly sampling a fraction of the original datasets using the sample function from Pandas. The number of samples in the reduced datasets is printed to verify the sizes.
8. The generate-numpy-arrays function is called for each reduced dataset to generate the NumPy arrays for images and labels. The NumPy arrays are saved to the specified directory (save-dir) in .npy and compressed .npz formats using the np.save and np.savez-compressed functions.
9. The saving process is repeated for each array (train data, train labels, val data, val labels, test data, test labels) by modifying the file names and array variables.

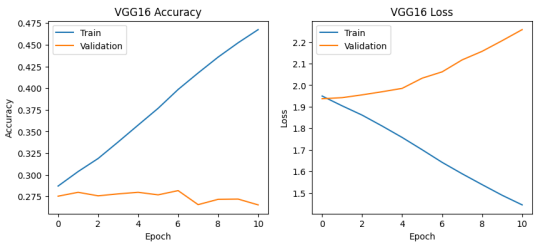


Figure 3: Accuracy and Loss per Epoch: VGG16.

Array Data Generator:
After the Numpy arrays have been generated, the model loads the saved NumPy arrays of train, validation, and test data and labels from Google Drive. Then, it defines the array-data-generator function, which takes the data, labels, and batch size as input parameters.
The working of this function is as follows:

1. The array-data-generator function takes three inputs: data (the input data array), labels (the corresponding labels array), and batch size. It utilizes a while True loop to ensure an infinite generation of batches.
2. Within each iteration of the loop, the function shuffles the data and labels arrays using the same permutation. This step randomizes the order of the data and labels, introducing randomness and avoiding potential biases during training. The np.random.permutation function is used to generate a random permutation of indices based on the total number of samples in the data array.
3. Next, the function calculates the number of batches based on the total number of samples and the specified batch size. This calculation is performed using integer division to determine the maximum number of complete batches that can be generated. Afterwards, a for loop iterates through the data and labels arrays in chunks of the given batch size. The loop calculates the starting index and ending index for the current batch based on the batch index and the specified batch size.
4. Within each iteration of the loop, the function uses the yield keyword to generate a batch of data and labels. The batch is obtained by indexing the data and label arrays using the calculated start and end indices. The yield keyword allows the function to generate batches on-the-fly, and it suspends the execution until the next batch is requested.

After the data arrays have been generated, the CNN model is created for training the data, to do so a function called create-model is defined. It facilitates the construction of a model architecture by combining a pre-trained base model with custom classification layers.

Model Creation Function(create-model):

1. The function takes three parameters: model name, num-classes, and input-shape. The model-name parameter allows the selection of a pre-trained base model from popular architectures such as VGG16, ResNet50, and InceptionV3. This flexibility enables researchers and practitioners to experiment with different model architectures and leverage their respective strengths in image feature extraction.
2. The function checks the model name and initializes the corresponding base model with pre-trained weights obtained from the ImageNet dataset. If an invalid model name is provided, a ValueError is raised to ensure the selection of a valid architecture.
3. Once the base model is selected, the function builds a custom classification head to be added on top of the base

model. It retrieves the output of the base model using base model.output. To reduce the spatial dimensions of the output, global average pooling is applied. This pooling operation aggregates the spatial information, allowing the model to focus on the most relevant features. A dense layer with 512 units and ReLU activation is then added to introduce non-linearity and facilitate the learning of complex patterns. Dropout with a rate of 0.5 is applied to prevent overfitting by randomly dropping connections between neurons during training. Finally, a dense layer with num-classes units and softmax activation is added to produce class predictions.

4. The function combines the base model and the custom layers into a final model using the Model class from the Keras library. The input to the model is set as base-model.input, and the output is set as predictions, which represents the predicted probabilities for each class.
5. To prevent the pre-trained layers of the base model from being updated during training, a loop iterates over each layer of the base model and sets the trainable attribute to False. This freezing of pre-trained layers ensures that the valuable knowledge captured by the pre-training is retained and not overwritten during the subsequent training process.
6. Finally, the constructed model is returned as the output of the function. This model can then be compiled and trained using appropriate optimization algorithms, loss functions, and evaluation metrics to perform image classification tasks. The create-model function provides a flexible and efficient way to build deep learning models for image classification by leveraging pre-trained architectures and custom classification heads, ultimately enabling accurate predictions on image data.

The Classification Report of the Model is as Follows:

	precision	recall	f1-score	support
0	0.28	0.18	0.22	699
1	0.00	0.00	0.00	81
2	0.32	0.10	0.15	1198
3	0.30	0.89	0.45	1882
4	0.00	0.00	0.00	349
5	0.00	0.00	0.00	663
6	0.27	0.15	0.19	706
7	0.16	0.04	0.07	756
8	0.27	0.05	0.08	834
accuracy			0.29	7168
macro avg	0.18	0.16	0.13	7168
weighted avg	0.23	0.29	0.20	7168

Validation accuracy: 29.24%

Figure 4: Classification Report : Inception V3 Model.

Training and Evaluation:

The function train-evaluate-model function orchestrates the training and evaluation process. Its working is as follows:

1. It initializes the lengths of the training, validation, and test dataset. These values represent the total number of samples in each dataset and can be predefined or calculated based on the dataset.
2. Next, the function creates the model by calling the create-model function, passing the model's name and number of classes as arguments. The create-model function constructs the model architecture based on the specified base model and custom layers and returns the model object.
3. After creating the model, the function compiles it using the Adam optimizer with a learning rate of 0.0001, categorical cross-entropy as the loss function, and accuracy as the evaluation metric. This configuration prepares the model for the training process.
4. The function also configures an early-stopping callback named early-stopping. This callback monitors the validation loss and restores the weights of the best model observed during training, allowing for the prevention of overfitting and improved generalization.
5. The number of steps per epoch for both the training and validation sets is calculated based on the lengths of the training and validation datasets divided by the batch size. This calculation determines the number of iterations required to complete an epoch.
6. The model is then trained using the fit method, specifying the training and validation generators, the number of epochs, the steps per epoch for training and validation, and the early stopping callback. The training process is performed with verbosity enabled (verbose=1), allowing for the display of progress and metrics during training. If early stopping is activated, meaning the training was stopped early due to a lack of improvement in the validation loss, a message indicating the activation of early stopping is printed.
7. After the training, the function proceeds to evaluate the model. It initializes empty lists, y-true and y-pred, to store the true labels and predicted labels, respectively. The number of batches in the validation set is determined based on the length of the validation dataset divided by the batch size.
8. The function iterates through the test set using a for loop. In each iteration, it retrieves the next batch of data and labels from the test generator. The true labels are stored by transforming them using np.argmax to obtain the indices of the highest values along the last axis. Predictions for the current batch are made using the trained model's prediction method, and the predicted labels are stored in y-pred after applying np.argmax.
9. To facilitate result interpretation, the true and predicted labels are transformed into class names using a label encoder. The inverse-transform method of the label encoder converts the integer labels back into their original class names. Additionally, to avoid any potential type errors, the class names are converted to strings.
10. The function then prints the classification report, including precision, recall, and F1-score, using the true and predicted labels. This report provides insights into the model's performance on each class. The validation accuracy is calculated by comparing the true and predicted labels and taking the mean of the resulting Boolean array.

Hyperparameters:

1. Model Creation Function(create-model): Models: (VGG16, ResNet50, InceptionV3); number of classes: 10; input shape: Shape of the input image (default: (224, 224, 3)); Dense layer units: 512; Dropout rate: 0.5.
   These parameters were selected to achieve the best performance in decent computing time making them the most efficient with our given hardware constraints.
2. Training and Evaluation: Learning rate: 0.0001, Loss function: categorical cross entropy, Optimizer: Adam, Patience for early stopping: 10, Metrics: accuracy, Batch Size: 128, Epochs: 100.
   Note: These hyperparameters were chosen to get the most efficient working model with hardware computation that was available to us and not the ideal metrics.

The Classification Report of the Model is as Follows:

	precision	recall	f1-score	support
0	0.16	0.09	0.12	701
1	0.00	0.00	0.00	85
2	0.00	0.00	0.00	1164
3	0.27	0.96	0.42	1875
4	0.00	0.00	0.00	346
5	0.14	0.00	0.01	672
6	0.00	0.00	0.00	724
7	0.00	0.00	0.00	767
8	0.06	0.00	0.00	834
accuracy			0.26	7168
macro avg	0.07	0.12	0.06	7168
weighted avg	0.11	0.26	0.12	7168

Validation accuracy: 25.98%

Figure 5: Classification Report : ResNet50 Model.

Finally, the validation accuracy is printed, and both the validation accuracy and the training history are returned as outputs of the function.

This function is used to train three popular CNN models for valence prediction, its classification reports are printed and the accuracy and loss graphs are plotted as well.

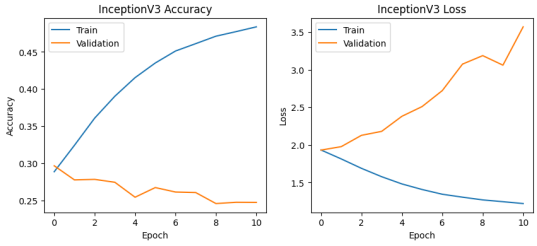


Figure 6: Accuracy and Loss per Epoch: Inception V3.

Results

Since only 5 percent of the dataset was used due to computational constraints, the highest accuracy achieved was only **29.24%**.

An explanation of the results of the various models is as follows:

VGG16:

The classification report of this model is given in Figure 2 represented in a table format. The results are as follows:

1. Each row in the table corresponds to a specific class, labeled from 0 to 8.
2. The precision, recall, and F1-score metrics are reported for each class, indicating the model's performance in correctly identifying instances of each class. The "support" column denotes the number of instances present in the dataset for each class.
3. Overall accuracy is reported as **27.34%**, indicating the proportion of correctly predicted instances across all classes. The "macro avg" row represents the average precision, recall, and F1-score across all classes, giving an overall performance measure.
4. The "weighted avg" row represents the weighted average of precision, recall, and F1-score, taking into account the number of instances in each class. It is important to note that some classes have low performance metrics, such as class 1, 4, 5, and 7, which have very low precision, recall, and F1-scores.
5. Class 3 stands out with relatively high recall and F1-score (0.90 and 0.43, respectively), indicating good performance in correctly identifying instances of this class. The overall performance of the model, based on the weighted average metrics, shows limited effectiveness in accurately predicting the emotional valence for the given dataset

Hence the achieved accuracy for the VGG16 model was **27.34%**. The graphs that plot the Accuracy per Epoch and the Loss per Epoch are given in Figure 3.

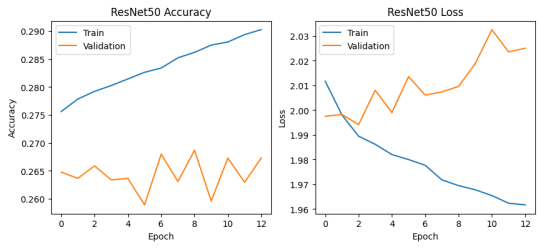


Figure 7: Accuracy and Loss per Epoch: ResNet 50.

ResNet 50:

The classification report of this model is given in Figure 4 represented in a table format. The results are as follows:

1. Each row in the table corresponds to a specific class, labelled from 0 to 8.
2. The precision, recall, and F1-score metrics are reported for each class, indicating the model's performance in correctly identifying instances of each class. The "support"

column denotes the number of instances present in the dataset for each class.

3. Overall accuracy is reported as **25.98%**, indicating the proportion of correctly predicted instances across all classes. The "macro avg" row represents the average precision, recall, and F1-score across all classes, giving an overall performance measure.
4. The "weighted avg" row represents the weighted average of precision, recall, and F1-score, taking into account the number of instances in each class. The majority of the classes have very low precision, recall, and F1 scores, with many of them reported as 0. This suggests that the model struggles to accurately predict the emotional valence for most classes. This is because of the extremely low size of the dataset used due to computational constraints.
5. Class 3 stands out with relatively high recall and F1-score (0.96 and 0.42, respectively), indicating good performance in correctly identifying instances of this class. The overall performance of the model, based on the weighted average metrics, shows limited effectiveness in accurately predicting the emotional valence for the given dataset.

Hence the achieved accuracy for the ResNet 50 model was **25.98%**. The graphs that plot the Accuracy per Epoch and the Loss per Epoch are given in Figure 5.

Inception V3:

The classification report of this model is given in Figure 6 represented in a table format. The results are as follows:

1. Each row in the table corresponds to a specific class, labelled from 0 to 8.
2. The precision, recall, and F1-score metrics are reported for each class, indicating the model's performance in correctly identifying instances of each class. The "support" column denotes the number of instances present in the dataset for each class.
3. Overall accuracy is reported as **29.24%**, indicating the proportion of correctly predicted instances across all classes. The "macro avg" row represents the average precision, recall, and F1-score across all classes, giving an overall performance measure.
4. The "weighted avg" row represents the weighted average of precision, recall, and F1-score, taking into account the number of instances in each class. It is important to note that some classes have higher precision, recall, and F1-scores compared to others, indicating better performance in identifying instances of those classes.
5. Class 3 stands out with relatively high recall and F1-score (0.89 and 0.45, respectively), indicating good performance in correctly identifying instances of this class. On the other hand, classes 1, 4, 5, and 7 have very low precision, recall, and F1-scores, suggesting that the model struggles to accurately predict the emotional valence for these classes.

Hence the achieved accuracy for the Inception V3 model was **29.24%**. The graphs that plot the Accuracy per Epoch and the Loss per Epoch are given in Figure 7. This model performed with the highest accuracy among all three models.

Conclusion

In this project, we aimed to predict emotional valence from images using deep learning models. However, the results obtained from the models indicate that satisfactory performance was not achieved across the different models evaluated. This can be attributed to the limitations imposed by the small size of the dataset used for training and validation.

The classification reports demonstrate low precision, recall, and F1-scores for most classes, indicating that the models struggled to accurately predict the emotional valence for the majority of instances. Although some classes showed relatively higher performance, the overall effectiveness of the models in capturing the nuanced emotional content of the images was limited. One of the key factors contributing to these suboptimal results is the small size of the dataset. With a limited number of samples available for training and validation, the models lacked the necessary complexity and diversity to learn robust representations of emotional valence. As a result, they failed to generalize well to unseen instances, leading to poor predictive performance. The computational limitations imposed by the dataset size restricted the models' ability to capture the complex relationships and patterns present in emotional valence prediction from images.

A larger dataset would have provided more instances for the models to learn from, potentially enabling them to discover more discriminative features and improve their predictive capabilities. In conclusion, while the attempted models and methodology hold promise for predicting emotional valence from images, the limited size of the dataset used in this project hindered the achievement of good results. Future work should focus on acquiring or generating larger datasets to overcome these limitations and enhance the models' performance in capturing the subtle emotional nuances conveyed by visual stimuli.

References

[1] https://www.artemisdataset.org/#dataset.

[2] P. Achlioptas, M. Ovsjanikov, K. Haydarov, M. Elhoseiny, and L. Guibas, "ArtEmis: Affective Language for Visual Art," in CoRR, vol. abs/2101.07396, 2021.

[3] S. K. W. Hwooi, A. Othmani and A. Q. M. Sabri, "Deep Learning-Based Approach for Continuous Affect Prediction From Facial Expression Images in Valence-Arousal Space," in IEEE Access, vol. 10, pp. 96053-96065, 2022, doi: 10.1109/ACCESS.2022.3205018.

[4] A.Hanjalic,"Extracting moods from pictures and sounds : Towards truly personalized tv,"IEEE Signal Processing Magazine,vol.23,no.2,pp. 90–100,2006.

[5] Cowen, A. S., Keltner, D. (2017). Self-report captures 27 distinct categories of emotion bridged by continuous gradients. "Proceedings of the National Academy of Sciences", 114(38), E7900-E7909.

Data_Generator

May 16, 2023

[ ]:
```
#Import Required Libraries
import os
import numpy as np
import cv2
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from google.colab import drive
from tqdm import tqdm
from joblib import Parallel, delayed
```

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

```
Mounted at /content/drive
```

[ ]:
```
# Define data directories
wikiart_dir = '/content/drive/MyDrive/DeepLearningProject/wikiart'
artemis_data_file = '/content/drive/MyDrive/DeepLearningProject/ProcessedData/
 ↪artemis_preprocessed.csv'

# Load the ArtEmis data
artemis_data = pd.read_csv(artemis_data_file)

# Define emotion label encoder
le = LabelEncoder()
le.fit(artemis_data['emotion'])

# Define image size
img_size = (224, 224)
```

```
import tensorflow as tf

def load_image(file_path):
    # Read the image from the file_path using TensorFlow's io.read_file and␣
 ↪image.decode_image functions
    image = tf.io.read_file(file_path)
    image = tf.image.decode_image(image)

    # Resize the image to the desired size (img_size) using TensorFlow's image.
 ↪resize function
    image = tf.image.resize(image, img_size)

    # Convert the color space of the image from BGR (default in OpenCV) to RGB␣
 ↪using TensorFlow's image.rgb_to_grayscale function
    image = tf.image.rgb_to_grayscale(image)

    # Normalize the pixel values of the image by converting the data type to␣
 ↪float32 and dividing by 255.0
    # This brings the pixel values to the range [0, 1] which is commonly used␣
 ↪in deep learning models
    image = tf.cast(image, tf.float32) / 255.0

    # Return the preprocessed image
    return image
```

The code starts by defining the paths to the data directories and files, specifically the directory containing the WikiArt images (wikiart_dir) and the file path to the preprocessed ArtEmis data (artemis_data_file). The ArtEmis data is loaded into a pandas DataFrame using the read_csv function. The data is read from the CSV file specified by artemis_data_file.

The code defines a label encoder (le) to encode the emotion labels present in the ArtEmis data. The encoder is fit on the 'emotion' column of the ArtEmis DataFrame using the fit method. The image size is defined as (224, 224), indicating the desired dimensions for the images after resizing. TensorFlow library is imported as tf.

The code defines a function called load_image that takes a file path as input and performs image preprocessing using TensorFlow functions. Inside the load_image function, the image is read from the file path using TensorFlow's io.read_file function, which reads the image data as a byte string. The decode_image function from TensorFlow's image module is used to decode the image from the byte string format.

The image is resized to the desired dimensions (img_size) using TensorFlow's image.resize function. This ensures that all images have the same size for consistency during model training. The color space of the image is converted from the default BGR (commonly used in OpenCV) to grayscale using TensorFlow's image.rgb_to_grayscale function. This converts the image to a single-channel grayscale representation.

The pixel values of the image are normalized by converting the data type to float32 and dividing

by 255.0. This normalization step scales the pixel values to the range of [0, 1], which is a common practice in deep learning models. The preprocessed image is returned as the output of the load_image function.

[ ]:
```
def generate_numpy_arrays(dataframe):
    # Get the total number of samples in the input dataframe
    num_samples = len(dataframe)

    # Initialize the data array with zeros
    data = np.zeros((num_samples, img_size[0], img_size[1], 3), dtype=np.
 ↪float32)

    # Convert emotion labels to one-hot-encoded labels
    labels = to_categorical(le.transform(dataframe['emotion']),␣
 ↪num_classes=len(le.classes_))

    # Define a helper function to load and preprocess an image
    def process_image(index):
        row = dataframe.iloc[index]
        image_path = os.path.join(wikiart_dir, row['art_style'],␣
 ↪row['painting'] + '.jpg')
        return load_image(image_path)

    # Load and preprocess images in parallel
    images = Parallel(n_jobs=-1)(delayed(process_image)(i) for i in␣
 ↪tqdm(range(num_samples), desc='Loading Images'))

    # Store the preprocessed images in the data array
    data[:] = images

    # Convert labels to float32
    labels = labels.astype(np.float32)

    # Return the data and labels arrays
    return data, labels
```

The provided code defines a function called generate_numpy_arrays that processes a DataFrame containing image data and generates numpy arrays for the images and their corresponding labels. Let's understand how the code works:

The function takes a DataFrame (dataframe) as input, which contains image data and emotion labels. The total number of samples in the input DataFrame is obtained using the len function, and it is stored in the num_samples variable.

An array called data is initialized with zeros. It has dimensions (num_samples, img_size[0], img_size[1], 3) and a data type of float32. This array will hold the preprocessed image data.

The emotion labels in the DataFrame are transformed into one-hot-encoded labels using the to_categorical function. The labels are first transformed using the label encoder (le) to convert

them into numeric representations. The number of classes is determined by the length of the label encoder's classes (len(le.classes_)).

The code defines a helper function, process_image, which takes an index as input and retrieves the corresponding row from the DataFrame. It constructs the file path for the image by joining the directory path (wikiart_dir), art style, and painting name with the appropriate file extension. This function then calls the load_image function (defined earlier) to preprocess the image.

The Parallel class from the joblib library is used to load and preprocess the images in parallel. The Parallel class runs multiple instances of the process_image function concurrently, speeding up the image loading process. The n_jobs=-1 parameter specifies that all available CPU cores should be used for parallel processing.

The images list is populated with the preprocessed images obtained from parallel processing. The tqdm function is used to display a progress bar indicating the loading progress. The preprocessed images in the images list are stored in the data array using the slicing operation data[:] = images. This copies the elements of the images list to the corresponding positions in the data array.

The data type of the labels is converted to float32 using the astype method.Finally, the function returns the data array (containing the preprocessed images) and the labels array (containing the corresponding labels) as the output.

train_df: 72% of the data for training (0.8 * 0.9 = 0.72)

val_df: 8% of the data for validation (0.8 * 0.1 = 0.08)

test_df: 20% of the data for testing

[ ]:
```
train_df, test_df = train_test_split(artemis_data, test_size=0.2,␣
 ↪random_state=2021)
train_df, val_df = train_test_split(train_df, test_size=0.1, random_state=2021)

# Find Number of samples used:
num_train_samples = len(train_df)
num_val_samples = len(val_df)
num_test_samples = len(test_df)

print(f"Number of training samples: {num_train_samples}")
print(f"Number of validation samples: {num_val_samples}")
print(f"Number of test samples: {num_test_samples} \n")

# Reduce the datasamples to train on colab:

#create reduction ratios
train_frac = 0.05
val_frac = 0.1
test_frac = 0.1

#Create the reduced datasets
train_df_reduced = train_df.sample(frac=train_frac, random_state=2021)
val_df_reduced = val_df.sample(frac=val_frac, random_state=2021)
```

```
test_df_reduced = test_df.sample(frac=test_frac, random_state=2021)

num_train_samplesr = len(train_df_reduced)
num_val_samplesr = len(val_df_reduced)
num_test_samplesr =len(test_df_reduced)

print(f"Number of reduced training samples: {num_train_samplesr}")
print(f"Number of reduced validation samples: {num_val_samplesr}")
print(f"Number of reduced test samples: {num_test_samplesr}")
```

```
Number of training samples: 327372
Number of validation samples: 36375
Number of test samples: 90937

Number of reduced training samples: 16369
Number of reduced validation samples: 3638
Number of reduced test samples: 9094
```

[ ]:
```
# Generate the NumPy arrays for each dataset
train_data, train_labels = generate_numpy_arrays(train_df_reduced)
val_data, val_labels = generate_numpy_arrays(val_df_reduced)
test_data, test_labels = generate_numpy_arrays(test_df_reduced)
```

```
Loading Images:   5%|          | 888/16369 [02:15<20:45,
12.43it/s]/usr/local/lib/python3.10/dist-
packages/joblib/externals/loky/process_executor.py:700: UserWarning: A worker
stopped while some jobs were given to the executor. This can be caused by a too
short worker timeout or by a memory leak.
  warnings.warn(
Loading Images: 100%|      | 16369/16369 [23:15<00:00, 11.73it/s]
Loading Images:  50%|        | 1812/3638 [00:31<00:33,
54.68it/s]/usr/local/lib/python3.10/dist-
packages/joblib/externals/loky/process_executor.py:700: UserWarning: A worker
stopped while some jobs were given to the executor. This can be caused by a too
short worker timeout or by a memory leak.
  warnings.warn(
Loading Images: 100%|      | 3638/3638 [01:07<00:00, 53.71it/s]
Loading Images:  12%|         | 1080/9094 [00:19<02:24,
55.37it/s]/usr/local/lib/python3.10/dist-
packages/joblib/externals/loky/process_executor.py:700: UserWarning: A worker
stopped while some jobs were given to the executor. This can be caused by a too
short worker timeout or by a memory leak.
  warnings.warn(
Loading Images: 100%|      | 9094/9094 [04:52<00:00, 31.12it/s]
```

[ ]:
```
# Save the NumPy arrays to Google Drive
save_dir = '/content/drive/MyDrive/DeepLearningProject/GeneratedData5p'
```

```
np.save(save_dir + '/train_data.npy', train_data)
np.save(save_dir + '/train_labels.npy', train_labels)

# Save the arrays as compressed NumPy files
np.savez_compressed(save_dir + '/train_data.npz', data=train_data)
np.savez_compressed(save_dir + '/train_labels.npz', labels=train_labels)
# Repeat for other arrays


np.save(save_dir + '/val_data.npy', val_data)
np.save(save_dir + '/val_labels.npy', val_labels)

# Save the arrays as compressed NumPy files
np.savez_compressed(save_dir + '/val_data.npz', data=train_data)
np.savez_compressed(save_dir + '/val_labels.npz', labels=train_labels)

np.save(save_dir + '/test_data.npy', test_data)
np.save(save_dir + '/test_labels.npy', test_labels)

# Save the arrays as compressed NumPy files
np.savez_compressed(save_dir + '/test_data.npz', data=train_data)
np.savez_compressed(save_dir + '/test_labels.npz', labels=train_labels)
```

Model_training_evaluation

May 16, 2023

[ ]:
```
#Import Required Libraries
import os
import numpy as np
import cv2
import tensorflow as tf
import pandas as pd

from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications import ResNet50, InceptionV3
from sklearn.metrics import classification_report
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications import ResNet50, InceptionV3
from sklearn.metrics import classification_report
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from google.colab import drive
from tensorflow.keras.callbacks import EarlyStopping
```

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

The dataset is split into train_df, val_df, and test_df to separate the data into training, validation, and test sets. These DataFrames contain the metadata (e.g., file paths and labels) needed for loading

the images and their corresponding labels.

The custom data generators (train_generator_main, validation_generator_main, and test_generator_main) are functions that take the metadata (in the form of DataFrames) and batch size as input. They yield batches of preprocessed images and their corresponding labels, which can be directly used as input by the model during training and evaluation.

This code first generates NumPy arrays for training, validation, and testing data by calling the generate_numpy_arrays function on the respective reduced dataframes. Then, it saves these NumPy arrays to Google Drive, creating separate files for the data and labels of each dataset. After saving the NumPy arrays, it loads them back from Google Drive and creates custom data generators using the array_data_generator function.

This function takes the data and labels arrays along with the batch size as input and yields batches of shuffled data and labels on each iteration. The code finally creates train_generator, validation_generator, and test_generator using this function, which can be used to feed data in batches to a neural network during training, validation, and testing.

[ ]:
```
# Load the NumPy arrays from Google Drive
train_data = np.load('/content/drive/MyDrive/DeepLearningProject/
 ↪GeneratedData5p/train_data.npy')
train_labels = np.load('/content/drive/MyDrive/DeepLearningProject/
 ↪GeneratedData5p/train_labels.npy')

val_data = np.load('/content/drive/MyDrive/DeepLearningProject/GeneratedData5p/
 ↪val_data.npy')
val_labels = np.load('/content/drive/MyDrive/DeepLearningProject/
 ↪GeneratedData5p/val_labels.npy')

test_data = np.load('/content/drive/MyDrive/DeepLearningProject/GeneratedData5p/
 ↪test_data.npy')
test_labels = np.load('/content/drive/MyDrive/DeepLearningProject/
 ↪GeneratedData5p/test_labels.npy')

# Define the array_data_generator function
def array_data_generator(data, labels, batch_size):
    num_samples = len(data)  # Get the total number of samples in the data array
    while True:
        # Shuffle the data and labels using the same permutation
        idx = np.random.permutation(num_samples)
        data = data[idx]
        labels = labels[idx]

        # Calculate the number of batches based on the batch size
        num_batches = num_samples // batch_size

        # Iterate through the data and labels arrays in chunks of the given␣
 ↪batch_size
```

```
        for batch_idx in range(num_batches):
            start_idx = batch_idx * batch_size  # Calculate the starting index␣
 ↪for the current batch
            end_idx = (batch_idx + 1) * batch_size  # Calculate the ending␣
 ↪index for the current batch

            # Yield a batch of data and labels using the start and end indices
            yield data[start_idx:end_idx], labels[start_idx:end_idx]

# Set the batch size for the data generators
batch_size = 128
# Create data generators for train, validation, and test data using the␣
 ↪array_data_generator function
train_generator = array_data_generator(train_data, train_labels, batch_size)
validation_generator = array_data_generator(val_data, val_labels, batch_size)
test_generator = array_data_generator(test_data, test_labels, batch_size)
```

This code loads the saved NumPy arrays of train, validation, and test data and labels from Google Drive. Then, it defines the array_data_generator function, which takes the data, labels, and batch size as input parameters. Inside the function, it calculates the total number of samples and the number of batches based on the given batch size. It uses a while True loop to continuously shuffle the data and labels and iterate through them in chunks of the given batch size. On each iteration, the function yields a batch of data and labels for the neural network to process. Finally, the code sets the batch size and creates train_generator, validation_generator, and test_generator using the array_data_generator function for train, validation, and test datasets, respectively

[ ]:
```
def create_model(model_name, num_classes, input_shape=(224, 224, 3)):
    # Choose the base model based on the provided model name
    if model_name == 'VGG16':
        base_model = VGG16(weights='imagenet', include_top=False,␣
 ↪input_shape=input_shape)
    elif model_name == 'ResNet50':
        base_model = ResNet50(weights='imagenet', include_top=False,␣
 ↪input_shape=input_shape)
    elif model_name == 'InceptionV3':
        base_model = InceptionV3(weights='imagenet', include_top=False,␣
 ↪input_shape=input_shape)
    else:
        raise ValueError("Invalid model name")

    # Add custom classification head to the base model
    x = base_model.output  # Get the output of the base model
    x = GlobalAveragePooling2D()(x)  # Apply global average pooling to reduce␣
 ↪spatial dimensions
    x = Dense(512, activation='relu')(x)  # Add a dense layer with 512 units␣
 ↪and ReLU activation
```

```
    x = Dropout(0.5)(x)  # Apply dropout to prevent overfitting (with a rate of␣
↪50%)
    predictions = Dense(num_classes, activation='softmax')(x)  # Add a dense␣
↪layer for class predictions using softmax activation

    # Combine the base model and custom layers into a final model
    model = Model(inputs=base_model.input, outputs=predictions)

    # Freeze pre-trained layers to prevent them from updating during training
    for layer in base_model.layers:
        layer.trainable = False

    return model
```

This function create_model creates a deep learning model by selecting a base model (VGG16, ResNet50, or InceptionV3) with pre-trained weights from ImageNet, and then adds custom classification layers on top of it. The function takes three arguments: model_name, num_classes, and input_shape. Based on the model_name, it initializes the chosen base model without the top layers (the classification head). Then, it adds custom layers to create a new classification head, consisting of global average pooling, a dense layer with 512 units and ReLU activation, a dropout layer with a 50% rate, and finally a dense layer with num_classes units and softmax activation for class predictions. The custom layers are combined with the base model to create the final model. Lastly, the function freezes the pre-trained layers of the base model to prevent them from updating during training, which allows the custom layers to learn from the features extracted by the base model. The final model is returned by the function.

From the Data Generator Code saved at the GeneratedData70p folder in Drive :

Number of reduced training samples: 16369

Number of reduced validation samples: 3638

Number of reduced test samples: 9094

[ ]:
```
def train_evaluate_model(model_name, train_generator, validation_generator,␣
↪test_generator,num_classes, epochs):
    print(f"Training {model_name} model...")

    len_train_df = 16369
    len_val_df = 3638
    len_test_df = 9094



    # Create the model with the specified base architecture
    model = create_model(model_name, num_classes)

    # Compile the model with appropriate optimizer, loss function, and metric
```

```
  model.compile(optimizer=Adam(lr=0.0001), loss='categorical_crossentropy',␣
↪metrics=['accuracy'])

  # Configure early stopping callback
  early_stopping = EarlyStopping(monitor='val_loss', patience=10,␣
↪restore_best_weights=True)

  # Calculate the number of steps per epoch for both training and validation
  train_steps_per_epoch = len_train_df // batch_size
  val_steps_per_epoch = len_val_df // batch_size

  # Train the model using the specified generators and steps per epoch for␣
↪training and validation
  history = model.fit(
      train_generator,
      epochs=epochs,
      steps_per_epoch=train_steps_per_epoch,
      validation_data=validation_generator,
      validation_steps=val_steps_per_epoch,
      verbose=1,
      callbacks=[early_stopping]
  )

  # Check if early stopping was activated
  if early_stopping.stopped_epoch > 0:
    print("Early stopping was activated. \n")

  # Evaluating the model
  print("Evaluating the model... \n")

  # Initialize lists to store true labels and predicted labels
  y_true = []
  y_pred = []

  # Define the number of batches in the validation set
  evaluation_steps = len_test_df // batch_size

  # Iterate through the validation set
  for i in range(evaluation_steps):
      # Get the next batch of data and labels
      batch_data, batch_labels = next(test_generator)

      # Store true labels
      y_true.extend(np.argmax(batch_labels, axis=-1))

      # Make predictions for the current batch and store predicted labels
      batch_pred = model.predict(batch_data)
```

```
        y_pred.extend(np.argmax(batch_pred, axis=-1))

    # Transform the true labels and predicted labels into class names using the␣
↪label_encoder
    class_names = label_encoder.inverse_transform(range(num_classes))
    y_true = label_encoder.inverse_transform(y_true)
    y_pred = label_encoder.inverse_transform(y_pred)

    # Convert class names to strings to avoid the "TypeError: object of type␣
↪'numpy.int64' has no len()" issue
    class_names = [str(cn) for cn in class_names]

    print(" The Classification Report of the Model is as Follows: \n")

    # Print the classification report (precision, recall, F1-score)
    print(classification_report(y_true, y_pred, target_names=class_names,␣
↪zero_division=0))


    # Calculate the validation accuracy
    validation_accuracy = np.mean(np.array(y_true) == np.array(y_pred))

    # Print the validation accuracy
    print(f"Validation accuracy: {validation_accuracy * 100:.2f}%")

    return validation_accuracy, history
```

This function train_evaluate_model trains and evaluates a deep learning model. It takes as arguments the model_name, train_generator, validation_generator, test_generator,num_classes, and epochs. The function first creates a model with the specified base architecture using the create_model function.

It then compiles the model with the Adam optimizer, categorical crossentropy loss, and accuracy metric. It sets up an early stopping callback to stop training when the validation loss stops improving, with a patience of 5 epochs. The function calculates the number of steps per epoch for both training and validation sets, then trains the model using the specified generators and steps per epoch.

After training, it evaluates the model by iterating through the test set, storing true labels and predicted labels. It then prints a classification report with precision, recall, and F1-score, calculates the validation accuracy, and returns it along with the training history.

[ ]:
```
import matplotlib.pyplot as plt

def plot_history(model_name, history):
    # Plot training & validation accuracy values
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
```

```
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title(f'{model_name} Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')

    # Plot training & validation loss values
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title(f'{model_name} Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')

    plt.show()
```

[ ]:
```
from sklearn.preprocessing import LabelEncoder

# Train and evaluate models
model_names = ['VGG16', 'ResNet50', 'InceptionV3']

best_model_name = None
best_accuracy = 0

#Define Label Encoder to find the number of classes
label_encoder = LabelEncoder()
train_labels_encoded = np.argmax(train_labels, axis=1)
label_encoder.fit(train_labels_encoded)

# Define the number of classes
num_classes = train_labels.shape[1]

#Define the Number of Epochs
epochs = 100

# Iterate through the list of model names
for model_name in model_names:
    # Train and evaluate the current model
    accuracy, history = train_evaluate_model(model_name, train_generator,␣
 ↪validation_generator, test_generator, num_classes, epochs)

    # Plot training and validation accuracy and loss
    plot_history(model_name, history)
```

```
    # Update the best model and accuracy if the current model has better␣
↪accuracy
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_model_name = model_name
```

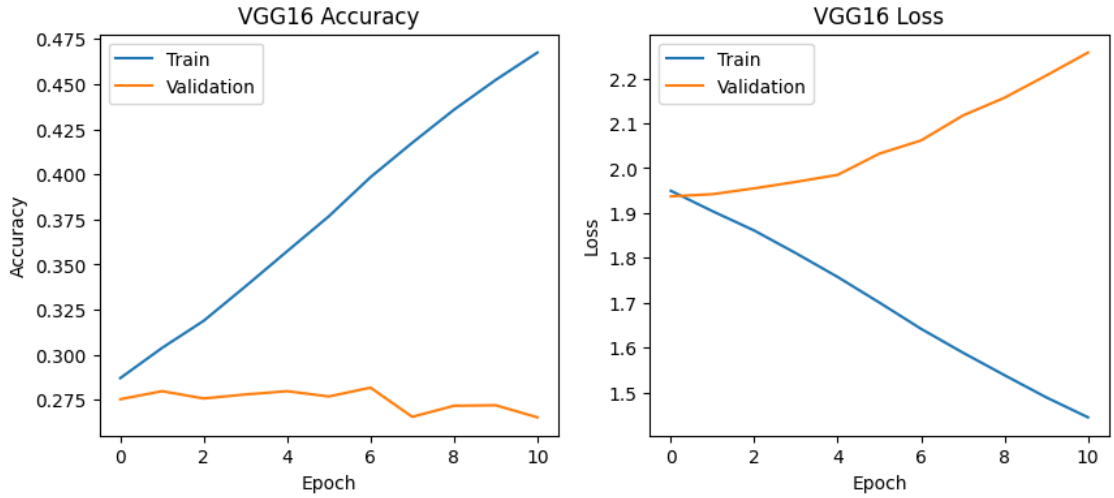
```
Training VGG16 model…

WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate`
or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Adam.

Epoch 1/100
1790/1790 [==============================] - 219s 115ms/step - loss: 1.9494 -
accuracy: 0.2871 - val_loss: 1.9370 - val_accuracy: 0.2754
Epoch 2/100
1790/1790 [==============================] - 197s 110ms/step - loss: 1.9040 -
accuracy: 0.3038 - val_loss: 1.9419 - val_accuracy: 0.2799
Epoch 3/100
1790/1790 [==============================] - 195s 109ms/step - loss: 1.8609 -
accuracy: 0.3189 - val_loss: 1.9549 - val_accuracy: 0.2758
Epoch 4/100
1790/1790 [==============================] - 195s 109ms/step - loss: 1.8103 -
accuracy: 0.3379 - val_loss: 1.9694 - val_accuracy: 0.2780
Epoch 5/100
1790/1790 [==============================] - 195s 109ms/step - loss: 1.7574 -
accuracy: 0.3574 - val_loss: 1.9850 - val_accuracy: 0.2799
Epoch 6/100
1790/1790 [==============================] - 195s 109ms/step - loss: 1.7003 -
accuracy: 0.3768 - val_loss: 2.0322 - val_accuracy: 0.2769
Epoch 7/100
1790/1790 [==============================] - 196s 109ms/step - loss: 1.6418 -
accuracy: 0.3985 - val_loss: 2.0616 - val_accuracy: 0.2818
Epoch 8/100
1790/1790 [==============================] - 195s 109ms/step - loss: 1.5888 -
accuracy: 0.4175 - val_loss: 2.1172 - val_accuracy: 0.2656
Epoch 9/100
1790/1790 [==============================] - 196s 109ms/step - loss: 1.5385 -
accuracy: 0.4358 - val_loss: 2.1569 - val_accuracy: 0.2718
Epoch 10/100
1790/1790 [==============================] - 195s 109ms/step - loss: 1.4892 -
accuracy: 0.4524 - val_loss: 2.2063 - val_accuracy: 0.2720
Epoch 11/100
1790/1790 [==============================] - 198s 111ms/step - loss: 1.4446 -
accuracy: 0.4675 - val_loss: 2.2575 - val_accuracy: 0.2653
Early stopping was activated.

Evaluating the model…
```

```
4/4 [==============================] - 1s 17ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
```

```
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 22ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
4/4 [==============================] - 0s 21ms/step
 The Classification Report of the Model is as Follows:
```

	precision	recall	f1-score	support
0	0.21	0.17	0.19	672
1	0.00	0.00	0.00	88
2	0.22	0.06	0.09	1217
3	0.28	0.90	0.43	1871
4	0.00	0.00	0.00	352
5	0.47	0.01	0.02	650
6	0.29	0.05	0.09	712
7	0.33	0.00	0.00	774
8	0.21	0.06	0.09	832
accuracy			0.27	7168
macro avg	0.22	0.14	0.10	7168
weighted avg	0.26	0.27	0.17	7168

```
Validation accuracy: 27.34%
```




```
Training ResNet50 model…

WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate`
```

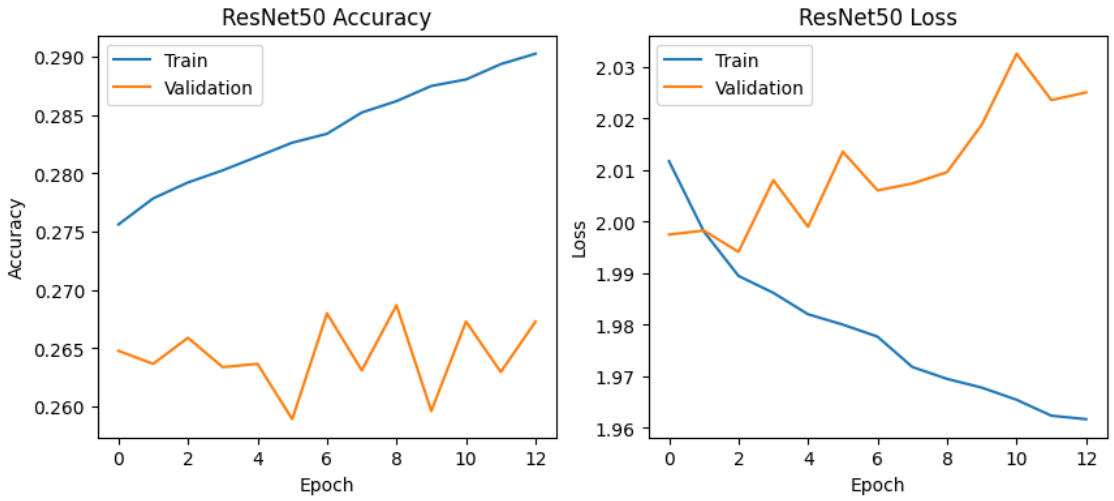
```
or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Adam.

Epoch 1/100
1790/1790 [==============================] - 188s 102ms/step - loss: 2.0117 -
accuracy: 0.2756 - val_loss: 1.9975 - val_accuracy: 0.2648
Epoch 2/100
1790/1790 [==============================] - 177s 99ms/step - loss: 1.9981 -
accuracy: 0.2779 - val_loss: 1.9983 - val_accuracy: 0.2637
Epoch 3/100
1790/1790 [==============================] - 173s 96ms/step - loss: 1.9895 -
accuracy: 0.2792 - val_loss: 1.9942 - val_accuracy: 0.2659
Epoch 4/100
1790/1790 [==============================] - 173s 96ms/step - loss: 1.9862 -
accuracy: 0.2803 - val_loss: 2.0081 - val_accuracy: 0.2634
Epoch 5/100
1790/1790 [==============================] - 173s 97ms/step - loss: 1.9821 -
accuracy: 0.2814 - val_loss: 1.9990 - val_accuracy: 0.2637
Epoch 6/100
1790/1790 [==============================] - 173s 97ms/step - loss: 1.9800 -
accuracy: 0.2826 - val_loss: 2.0136 - val_accuracy: 0.2589
Epoch 7/100
1790/1790 [==============================] - 173s 96ms/step - loss: 1.9777 -
accuracy: 0.2834 - val_loss: 2.0061 - val_accuracy: 0.2680
Epoch 8/100
1790/1790 [==============================] - 173s 97ms/step - loss: 1.9718 -
accuracy: 0.2852 - val_loss: 2.0074 - val_accuracy: 0.2631
Epoch 9/100
1790/1790 [==============================] - 173s 97ms/step - loss: 1.9695 -
accuracy: 0.2862 - val_loss: 2.0096 - val_accuracy: 0.2687
Epoch 10/100
1790/1790 [==============================] - 173s 97ms/step - loss: 1.9678 -
accuracy: 0.2875 - val_loss: 2.0188 - val_accuracy: 0.2596
Epoch 11/100
1790/1790 [==============================] - 176s 98ms/step - loss: 1.9655 -
accuracy: 0.2881 - val_loss: 2.0326 - val_accuracy: 0.2673
Epoch 12/100
1790/1790 [==============================] - 173s 97ms/step - loss: 1.9624 -
accuracy: 0.2894 - val_loss: 2.0236 - val_accuracy: 0.2630
Epoch 13/100
1790/1790 [==============================] - 173s 97ms/step - loss: 1.9617 -
accuracy: 0.2903 - val_loss: 2.0251 - val_accuracy: 0.2673
Early stopping was activated.

Evaluating the model…

4/4 [==============================] - 1s 15ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
```

```
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 18ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 16ms/step
```

```
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 17ms/step
4/4 [==============================] - 0s 18ms/step
 The Classification Report of the Model is as Follows:
```

	precision	recall	f1-score	support
0	0.16	0.09	0.12	701
1	0.00	0.00	0.00	85
2	0.00	0.00	0.00	1164
3	0.27	0.96	0.42	1875
4	0.00	0.00	0.00	346
5	0.14	0.00	0.01	672
6	0.00	0.00	0.00	724
7	0.00	0.00	0.00	767
8	0.06	0.00	0.00	834
accuracy			0.26	7168
macro avg	0.07	0.12	0.06	7168
weighted avg	0.11	0.26	0.12	7168

```
Validation accuracy: 25.98%
```

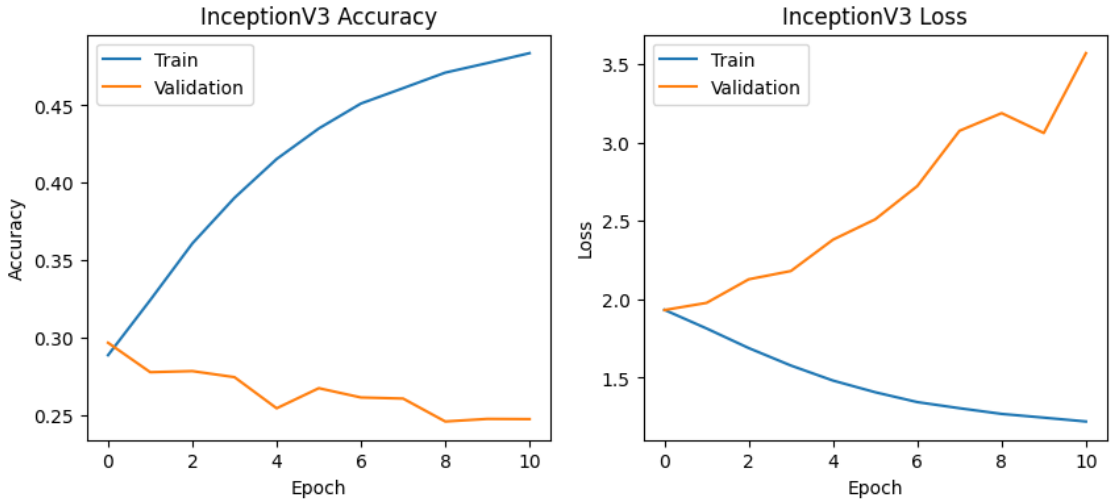



```
Training InceptionV3 model…
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicatio
ns/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5
87910968/87910968 [==============================] - 5s 0us/step

WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate`
```

```
or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Adam.

Epoch 1/100
1790/1790 [==============================] - 167s 89ms/step - loss: 1.9328 -
accuracy: 0.2889 - val_loss: 1.9315 - val_accuracy: 0.2969
Epoch 2/100
1790/1790 [==============================] - 157s 88ms/step - loss: 1.8139 -
accuracy: 0.3244 - val_loss: 1.9774 - val_accuracy: 0.2779
Epoch 3/100
1790/1790 [==============================] - 147s 82ms/step - loss: 1.6888 -
accuracy: 0.3609 - val_loss: 2.1278 - val_accuracy: 0.2786
Epoch 4/100
1790/1790 [==============================] - 142s 80ms/step - loss: 1.5773 -
accuracy: 0.3904 - val_loss: 2.1807 - val_accuracy: 0.2747
Epoch 5/100
1790/1790 [==============================] - 140s 78ms/step - loss: 1.4813 -
accuracy: 0.4154 - val_loss: 2.3810 - val_accuracy: 0.2546
Epoch 6/100
1790/1790 [==============================] - 138s 77ms/step - loss: 1.4070 -
accuracy: 0.4351 - val_loss: 2.5105 - val_accuracy: 0.2676
Epoch 7/100
1790/1790 [==============================] - 138s 77ms/step - loss: 1.3439 -
accuracy: 0.4512 - val_loss: 2.7231 - val_accuracy: 0.2616
Epoch 8/100
1790/1790 [==============================] - 140s 78ms/step - loss: 1.3049 -
accuracy: 0.4611 - val_loss: 3.0756 - val_accuracy: 0.2609
Epoch 9/100
1790/1790 [==============================] - 137s 76ms/step - loss: 1.2686 -
accuracy: 0.4711 - val_loss: 3.1884 - val_accuracy: 0.2461
Epoch 10/100
1790/1790 [==============================] - 137s 76ms/step - loss: 1.2448 -
accuracy: 0.4773 - val_loss: 3.0614 - val_accuracy: 0.2478
Epoch 11/100
1790/1790 [==============================] - 137s 76ms/step - loss: 1.2202 -
accuracy: 0.4837 - val_loss: 3.5713 - val_accuracy: 0.2476
Early stopping was activated.

Evaluating the model…

4/4 [==============================] - 2s 15ms/step
4/4 [==============================] - 0s 14ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 14ms/step
4/4 [==============================] - 0s 14ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 14ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 14ms/step
```

```
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 14ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 15ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
4/4 [==============================] - 0s 16ms/step
 The Classification Report of the Model is as Follows:
```

```
              precision    recall  f1-score   support

           0       0.28      0.18      0.22       699
           1       0.00      0.00      0.00        81
           2       0.32      0.10      0.15      1198
           3       0.30      0.89      0.45      1882
           4       0.00      0.00      0.00       349
           5       0.00      0.00      0.00       663
           6       0.27      0.15      0.19       706
           7       0.16      0.04      0.07       756
           8       0.27      0.05      0.08       834

    accuracy                           0.29      7168
   macro avg       0.18      0.16      0.13      7168
weighted avg       0.23      0.29      0.20      7168

Validation accuracy: 29.24%
```



This code snippet trains and evaluates three deep learning models ('VGG16', 'ResNet50', 'InceptionV3') using the provided train and validation generators. It defines the number of classes, sets the number of epochs to 20, and initializes variables to keep track of the best model and its accuracy.

It then iterates through the list of model names, training and evaluating each model with the train_evaluate_model function. After training and evaluating a model, it plots the training and validation accuracy and loss using the plot_history function. If the current model's accuracy is better than the previous best accuracy, it updates the best model and its accuracy. Finally, it prints the best model and its accuracy.

[ ]:
```
# Print the best model and its accuracy
```

```
print(f"Best model: {best_model_name} with accuracy: {best_accuracy * 100:.
  ↪2f}%")
```

```
Best model: InceptionV3 with accuracy: 29.24%
```