Emotion Recognition from Videos

Angsar Zhexengali, Zhanibek Murat

19.11.2024

Applied Machine Learning

Sunggat Aiymbay

2024

**Project goal**

Our goal is to develop a robust system for recognising human emotions from video data, using advanced machine learning techniques to improve accuracy and applicability in real-world scenarios.

**Dataset**

<https://www.kaggle.com/datasets/shuvoalok/raf-db-dataset/data>

Facial emotion recognition projects frequently use the Real-world Affective Faces Database, or RAF-DB. It includes 15000 Internet-sourced facial photos, and only basic emotion labels. This is a thorough description that pertains to our project:

The RAF-DB Dataset's salient features include:

Emotion Categories

Happiness, sadness, surprise, anger, disgust, fear, and neutral are among the seven basic emotions.

Method for Annotation:

Forty independent annotators categorize images by evaluating facial features to determine emotional expressions.

To guarantee accuracy and dependability, the data is subjected to extensive quality tests.

Applications:

It can be used to train and evaluate machine learning models for tasks like facial expression analysis, emotion identification, and even the creation of artificially emotive faces using GANs (Generative Adversarial Networks).

Diversity in the real world:

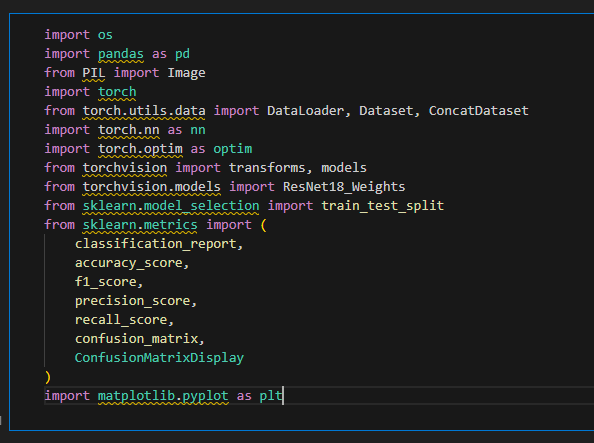
Numerous changes in lighting, angles, facial occlusions, and demographic characteristics like age, gender, and ethnicity are captured in the dataset. This guarantees that RAF-DB-trained models translate well to practical situations.

Format of Data:

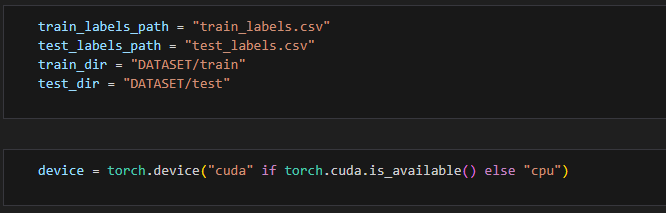
Because images are saved with their labels, preprocessing for activities like feature extraction or classification is simple.

Since it offers both strong and subtle emotional categories, this dataset is ideal for our project, Emotion Recognition from Videos, as it supports the construction of models that can manage the diversity of facial expressions found in the real world. CNNs and other deep learning models can be trained to accurately identify and distinguish emotions by integrating RAF-DB.

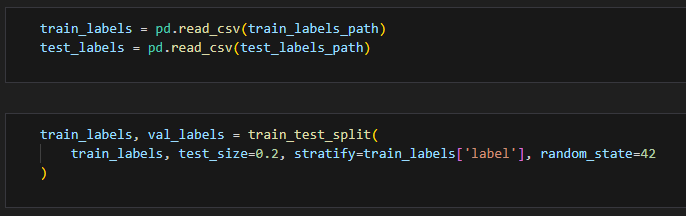
**Model**

****

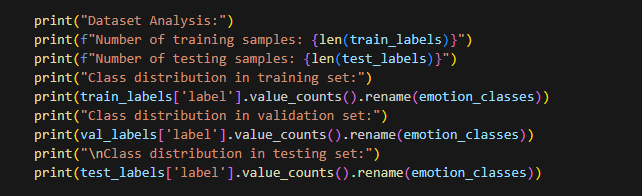
Here, we import the required libraries and set up an image categorization project. We analyze pictures using PIL, manage data using Pandas, and create and train neural networks using PyTorch, including using a pre-trained ResNet-18 model. We also use matplotlib to visualize results such as confusion matrices and scikit-learn to partition the data and assess the model's performance. We can effectively train and evaluate our machine learning model with this configuration.



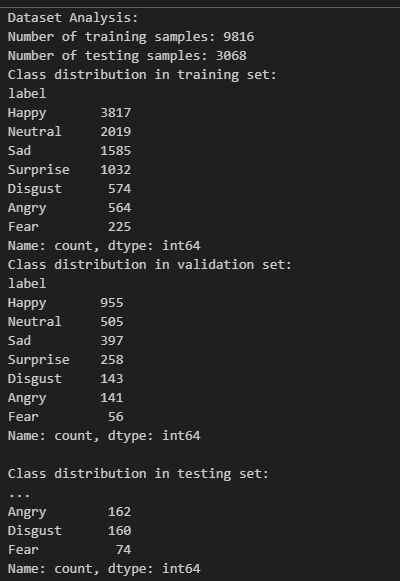
We define the directories holding the matching photos (`DATASET/train` and `DATASET/test`) as well as the paths for the training and testing labels (`train\_labels.csv` and `test\_labels.csv`). In addition, we changed the compute device to a GPU by checking for availability (‘cuda’). This guarantees that our hardware and data are set up correctly for model testing and training.



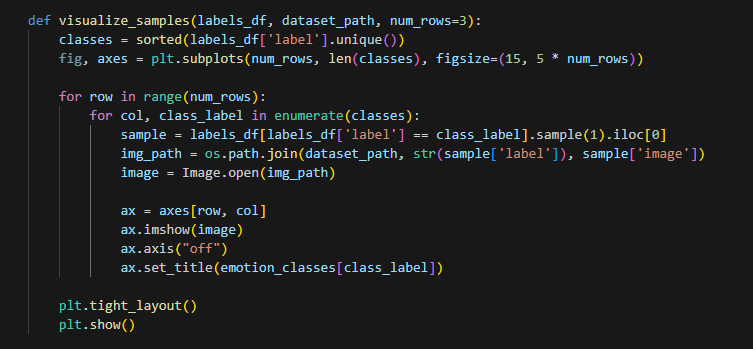
In this step, we load the label data and get it ready for validation and training. The training and testing label files are read into Pandas DataFrames using the pd.read\_csv function. Next, we use train\_test\_split to divide the training data into training and validation sets, making sure that the stratify option maintains a balanced class distribution. 20% of the training data is set aside for validation by using test\_size=0.2, and repeatability is guaranteed by using random\_state=42. The data is ready for model training and performance assessment in this step.



The number of samples in the training and testing sets is printed here, and the class distribution in the training, validation, and testing data is examined. The frequency of each class's appearance is counted using `value\_counts()`, and the labels are renamed for clarity using the `emotion\_classes` dictionary. Before starting training, this phase helps us make sure the dataset is balanced and comprehend its structure.

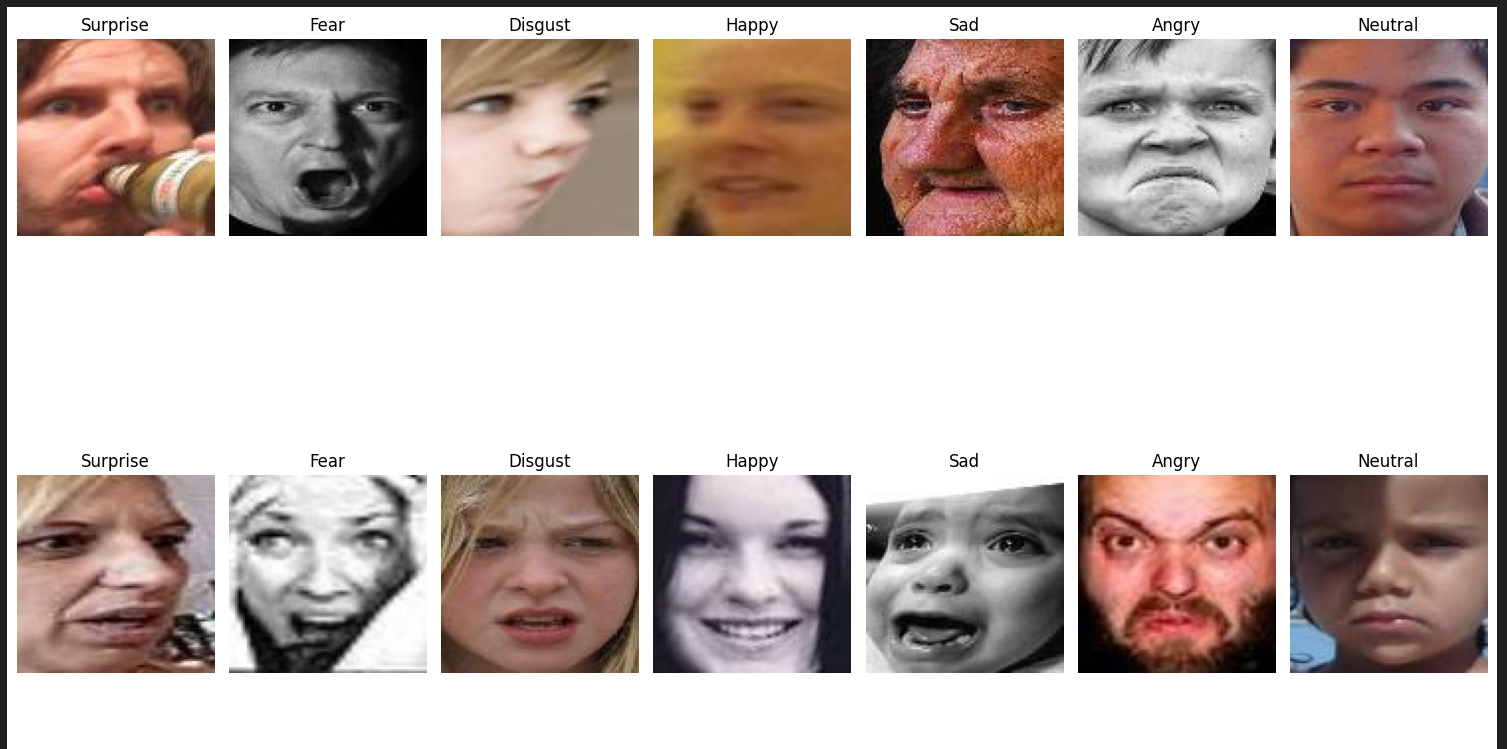


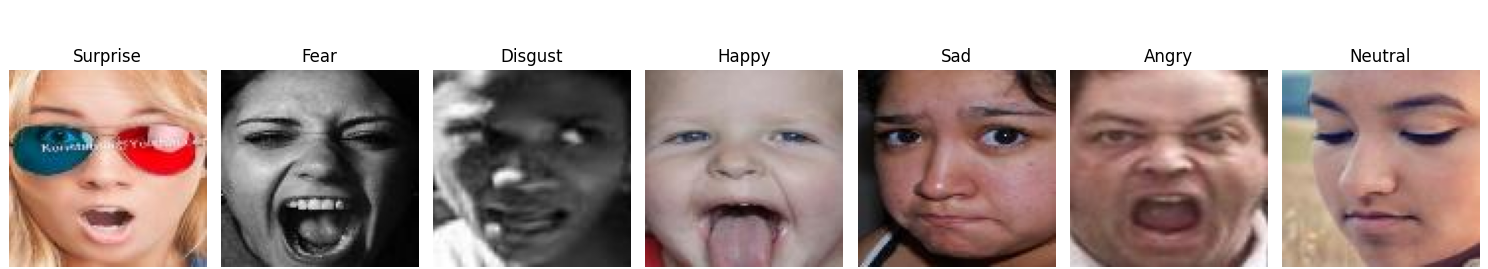
In our dataset, we have 9,816 samples for training, 2,448 for validation, and 3,068 for testing. The class distribution in the training set is led by "Happy" with 3,817 samples, followed by "Neutral" (2,019), "Sad" (1,585), and smaller amounts for "Surprise" (1,032), "Disgust" (574), "Angry" (564), and "Fear" (225). In the validation set, "Happy" also has the highest count with 955 samples, and the distribution is similar, though with fewer samples across each class. The testing set shows a comparable pattern, with "Happy" having 1,185 samples and the other classes following in similar proportions. This distribution helps us assess potential class imbalances before training the model.



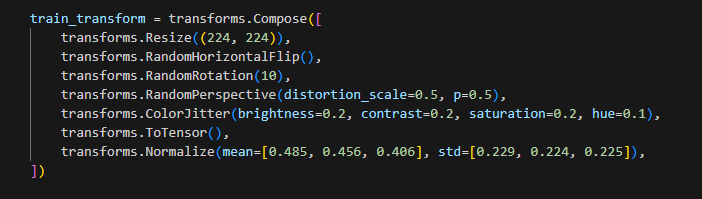
A grid of example photos from the dataset arranged by class labels is shown in the `visualize\_samples` function. We take as input the dataset path, DataFrame labels and the number of rows to be displayed. Next, we select a random image for each class, load it from the given path and display it on the subplot. Each distinct class is represented by a single column in the grid, and each image is labeled with the name of the appropriate emotion class from the `emotion\_classes` dictionary. Before starting model training, this enables us to visually inspect the dataset to make sure the image quality and class distribution are suitable.



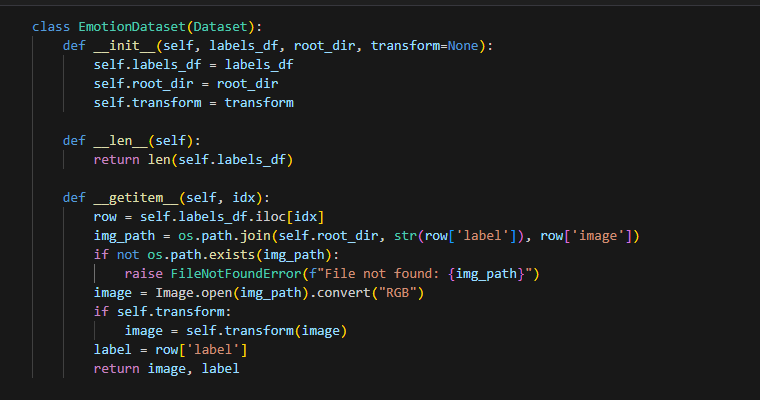




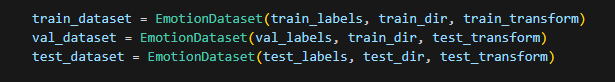
We show a grid of pictures from the training dataset using `visualize\_samples`. Images for each class are chosen at random from `train\_labels` and `train\_dir` and displayed in a grid with class labels. This enables us to rapidly assess the organization of the class and the quality of the images.



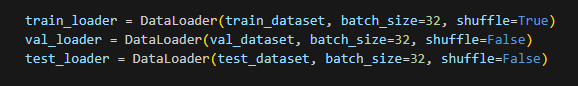
This code prepares and enhances images for training by defining `train\_transform`. To provide a consistent input size, we first resize every image to 224x224 pixels. Then, in order to add unpredictability, we randomly rotate and flip horizontally up to 10 degrees. We then imitate camera or picture warping by adding random perspective distortions ({distortion\_scale=0.5`) with a 50% chance. To make the model more resilient to variations in illumination, a `ColorJitter` is utilized to slightly alter brightness, contrast, saturation, and hue. `ToTensor()` is used to transform the photos to tensors, scaling the pixel values to a range of 0-1. In order to ensure uniform input distribution throughout training, we lastly normalize the images using mean and standard deviation values unique to pre-trained models like ResNet.



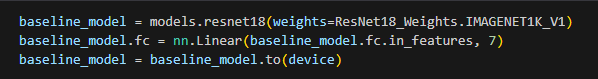
To manage our photos and labels, we build a unique PyTorch dataset under the `EmotionDataset` class. The labels DataFrame (`labels\_df`), the root directory (`root\_dir`), and optional image transformations (`transform`) are passed to the `\_\_init\_\_` method to build up the dataset. We can determine the total number of samples in the collection using the `\_\_len\_\_` method. By building the image path, determining whether the file exists, and loading the image in RGB format, the `\_\_getitem\_\_` method retrieves an image-label pair for a given index. We perform any alterations that are supplied to the image before sending it back with the label.



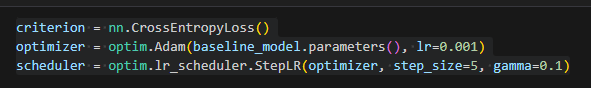
To arrange our training, validation, and testing data, we use the `EmotionDataset` class to construct three datasets. The training directory (`train\_dir`), the training labels (`train\_labels`), and the `train\_transform` pipeline are used by the `train\_dataset` to incorporate data augmentation for reliable training. The validation labels (`val\_labels`) and training data (`train\_dir`) are used to generate the `val\_dataset`. For consistency in evaluation, the simpler `test\_transform` is applied. Likewise, the `test\_dataset` prepares the data for model assessment by using the testing labels (`test\_labels`), the testing directory (`test\_dir`), and the `test\_transform`. This configuration guarantees that every dataset is appropriately preprocessed and arranged for its intended use.



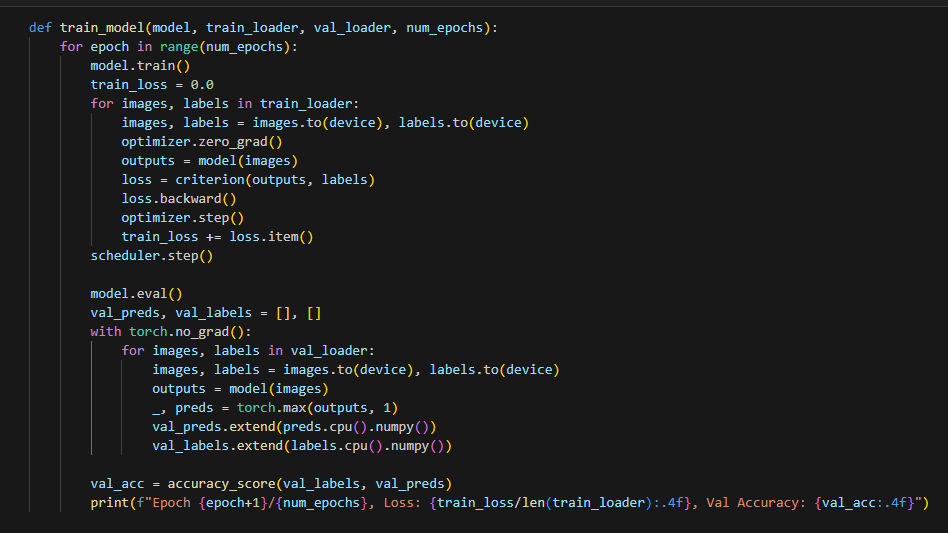
To effectively handle and batch the data during model training and evaluation, we develop data loaders for training, validation, and testing datasets. In order to avoid overfitting, the `train\_loader` loads data from the `train\_dataset` in batches of 32. It uses `shuffle=True` to randomly arrange the samples in each epoch. In batches of 32, the `val\_loader` and `test\_loader` load data from the `val\_dataset` and `test\_dataset`, respectively, using `shuffle=False` to guarantee that the data is processed in a predetermined sequence for reliable testing and validation. Large datasets can be handled smoothly and effectively thanks to this configuration, which arranges the data flow.



The code uses the pre-trained ResNet-18 architecture from the torchvision library to initialize a baseline model. Weights pre-trained on the ImageNet dataset are loaded using `weights=ResNet18\_Weights.IMAGENET1K\_V1`. We use `nn.Linear(baseline\_model.fc.in\_features, 7)` to change the model's last fully connected layer (`fc`) to produce 7 values rather than the default 1000 because our assignment entails classifying photos into 7 classes. This modification guarantees that the model is appropriate for our particular categorization task. To enhance performance, we use ResNet-18 as the baseline and take advantage of its pre-learned features from ImageNet. Lastly, to guarantee effective training and evaluation, we use `baseline\_model.to(device)` to transfer the complete model to the designated device.

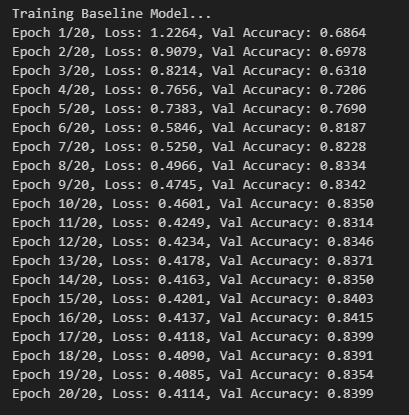


The learning rate scheduler, optimizer, and loss function are configured in this code in order to train our baseline model. For multi-class classification jobs, cross-entropy loss is frequently employed, as indicated by the `criterion = nn.CrossEntropyLoss()`. The Adam optimization approach, which helps modify the model's parameters during training, is then used with a learning rate of 0.001 by defining the `optimizer = optim.Adam(baseline\_model.parameters(), lr=0.001)`. To assist the model converge more successfully as training goes on, the `scheduler = optim.lr\_scheduler.StepLR(optimizer, step\_size=5, gamma=0.1)` lastly configures a learning rate scheduler that lowers the learning rate by a factor of 0.1 every 5 epochs.

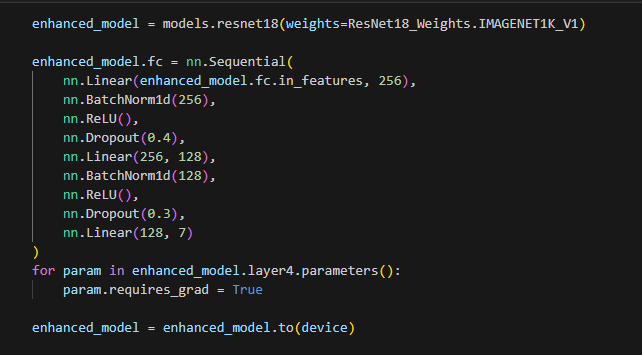


We specify our model's training procedure in this function. The model, training data loader (`train\_loader`), validation data loader (`val\_loader`), and the number of epochs (`num\_epochs`) are all inputs to the `train\_model` function. In order to track the cumulative loss during training, we first initialize `train\_loss` and set the model to training mode using `model.train()` for each epoch. After that, we cycle through the batches in `train\_loader`, transfer the labels and pictures to the device (GPU or CPU), compute the model's predictions, compute the loss using the criterion, use `loss.backward()` to execute backpropagation, and use `optimizer.step()` to adjust the model's parameters. We use `scheduler.step()` to adjust the learning rate after processing every batch in the training data.

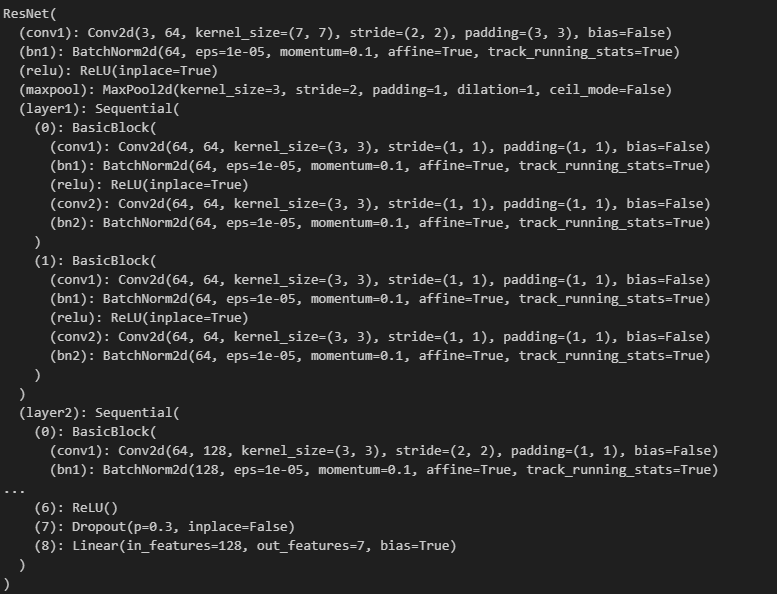
Next, we use model.eval() to switch the model to evaluation mode and compute validation performance. To reduce memory and computation time, we do not use torch.no\_grad() to compute gradients for each batch in the validation set. After gathering true labels and predictions, we use accuracy\_score to determine the accuracy. To track the model's development, we lastly report the validation accuracy and average training loss at the end of each session.



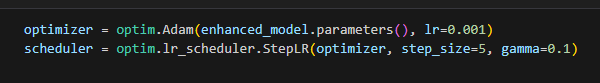
So now, the result displays the baseline model's training progress over a period of 20 epochs. In the beginning, the loss drops from 1.2264 to 0.4114, showing that the model is getting better at making predictions. Starting at 68.64%, the validation accuracy rises gradually over training, peaking at 84.15% in epoch 16. The accuracy stabilizes between 83 and 84%, despite a minor oscillation in subsequent epochs. With steady gains in validation accuracy and loss reduction, the model shows an overall upward trend.



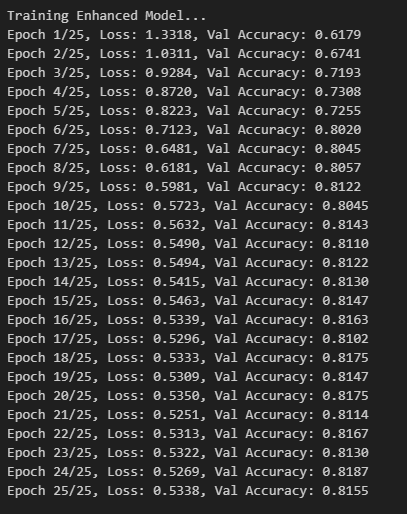
We are developing an improved ResNet-18 model with this code. First, we use ImageNet's pre-trained weights to initialize the model. The fully connected (fc) layer of the model is then altered by substituting a unique layer sequence. The new sequence is composed of two linear layers separated by dropout regularization, batch normalization, and ReLU activation. The purpose of this structure is to decrease overfitting and enhance model generalization. Following batch normalization, ReLU, and dropout (with a dropout rate of 0.4), the first linear layer shrinks the output size to 256. With a dropout rate of 0.3 and comparable batch normalization and ReLU, a second linear layer shrinks the size to 128. The final layer, which corresponds to the number of output classes, shrinks the size to 7. Furthermore, we make sure that the last layer's (layer 4) parameters are trainable, meaning that while the previous layers remain frozen, they will be altered during the backpropagation process. The device (CPU or GPU) is then shown this model.



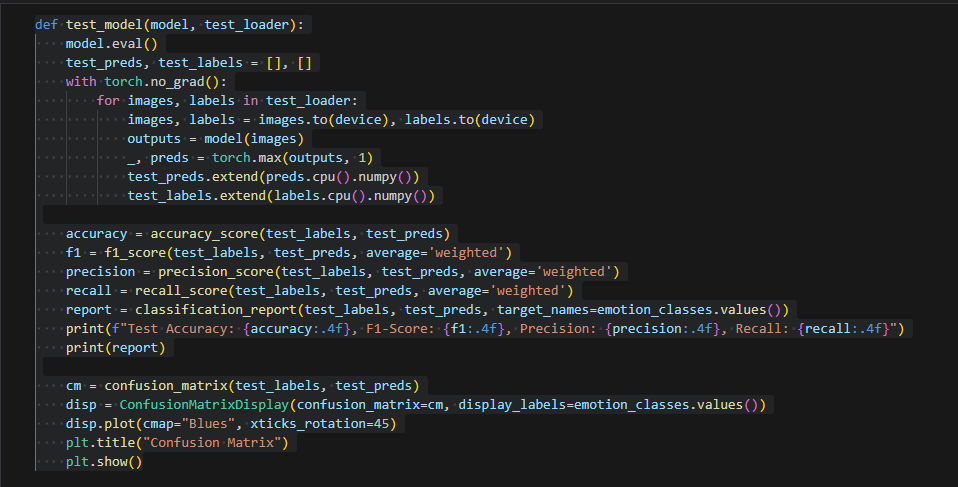
The output displays a modified ResNet-18 model's architecture. The convolutional layer (`conv1`) is the first layer in the model. Batch normalization (`bn1`), ReLU activation, and max pooling come next. The learning of complicated features is then made efficient by the inclusion of many residual blocks (`layer1`, `layer2`, etc.), each of which consists of two convolutional layers with batch normalization and ReLU activations. The model's own fully connected layers with batch normalization, ReLU activation, and dropout for regularization come after the convolutional layers. The seven units in the final output layer match the seven classification classes. During training, the dropout layers lessen overfitting.



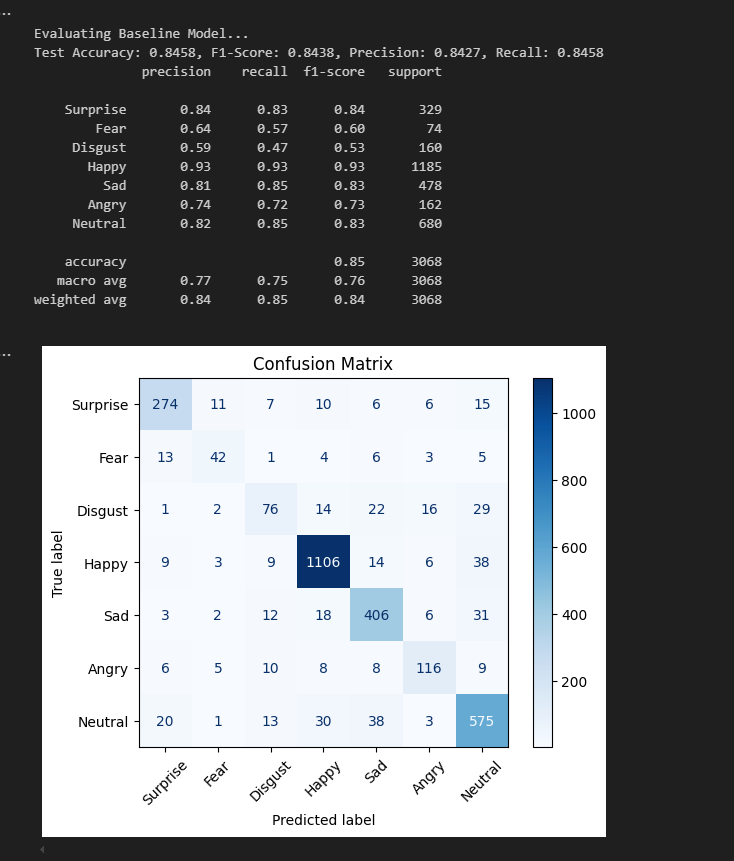
This code enables the model to modify its weights during training by configuring the Adam optimizer for the `enhanced\_model` with a learning rate of 0.001. Additionally, we used `StepLR` to construct a learning rate scheduler that lowers the learning rate by 0.1 every five epochs. By permitting faster learning at first then reducing it as training goes on, this technique aids the model in fine-tuning its learning process, enhancing its capacity to converge and maybe preventing overfitting later on.



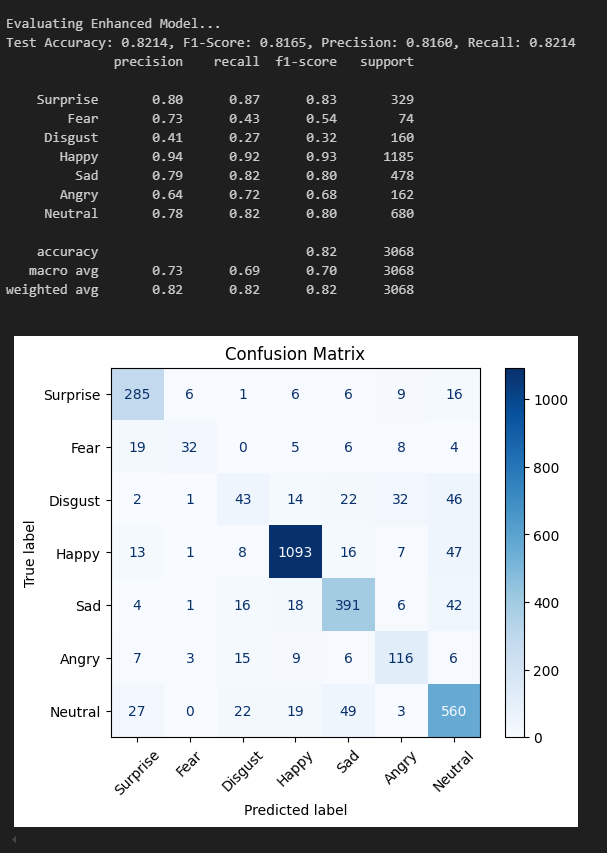
Validation accuracy steadily improves during 25 epochs of training the improved model. The model's validation accuracy was 61.79% at the beginning and grew to 81.87% by the end of the epoch. As time went on, the loss gradually dropped from 1.33 to 0.53, suggesting improved learning. Between epochs 6 and 10, there were notable gains in accuracy, going from 80.20% to 81.43%. The model achieved a final validation accuracy of 81.55%, indicating that it learned efficiently throughout training, despite some accuracy swings in subsequent epochs.



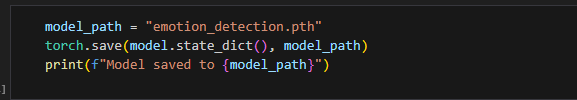
To completely assess our trained model's performance on the test dataset, we created the `test\_model` function. We collected genuine labels and model outputs to calculate important metrics including accuracy, F1-score, precision, and recall while ensuring dependable predictions by utilizing evaluation mode. Additionally, we displayed the confusion matrix to find trends in misclassifications and produced a thorough classification report to evaluate performance by class. These procedures ensured a thorough and perceptive evaluation process by assisting us in determining the model's generalization ability to new data and identifying opportunities for development.



Using the test dataset, we assessed the baseline model and obtained 84.58% accuracy, 84.38% weighted F1-score, 84.27% precision, and 84.58% recall. Happy had the highest classification accuracy (93% precision, recall, and F1-score) among the emotions, but Fear and Disgust had the lowest classification accuracy (F1-scores of 60% and 53%, respectively), most likely as a result of the dataset's lower support. Although underrepresented emotions require development, the model's overall performance was good, with a weighted average F1-score showing balanced performance across classes.



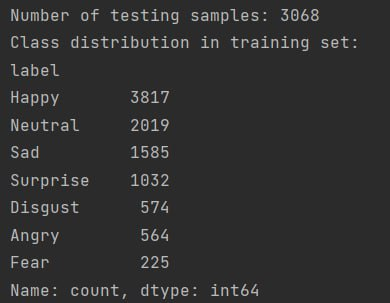
The improved model obtained 82.14% test accuracy, 81.65% F1-score, 81.60% precision, and 82.14% recall. Its accuracy (84.58%, F1: 84.38%) is marginally below the baseline, although it performs better when handling some difficult emotions such as Fear (precision: 73% vs. 64%). Its accuracy in classifying Disgust, however, drastically decreased (precision: 41% vs. 59%). This demonstrates how the model's architectural changes result in performance trade-offs across various emotions.

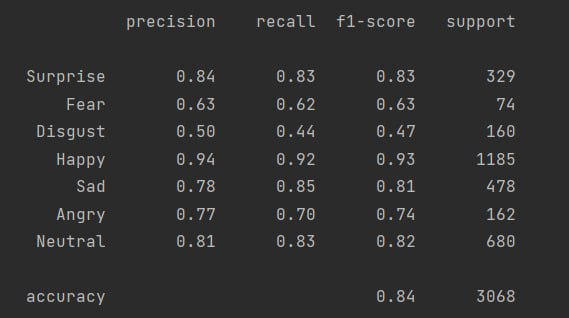


We saved this model for future use

**Additional Data**

Despite the diversity and categorised emotion classes, the RAF-DB dataset has a significant drawback: it lacks sufficient samples for several emotions, most notably fear and disgust. This imbalance negatively affects the performance of deep learning models, especially when it comes to the learning that determines the model's ability to accurately recognise each instance of a particular emotion. The model learns less about these emotions when there are fewer examples, resulting in low recognition accuracy.





Manual data augmentation is our solution.

To solve this problem, we improved the dataset and our model's performance by doing the following:

Manual Collection of Samples:

We carefully looked through numerous open-source databases, internet archives, and carefully selected image collections to find excellent representations of disgust and fear.

These photos were chosen to reflect a variety of changes, such as lighting, demographic diversity, and facial angles, making sure they matched the original dataset's features.

Preprocessing images:

To conform to the measurements of the RAF-DB dataset, all gathered photos were reduced to 48x48 pixels. This scaling procedure preserved key facial features required for emotion recognition while guaranteeing consistency in the data format.

In order to prepare the photos for machine learning models, the preprocessing also involved cropping and normalization where necessary.

Including in the Dataset:

The amount of samples in each emotion category was successfully balanced by the smooth addition of the freshly processed photos to the existing dataset.

In order to preserve quality and accuracy, labels for these emotions were thoroughly examined.

Results of the Augmentation

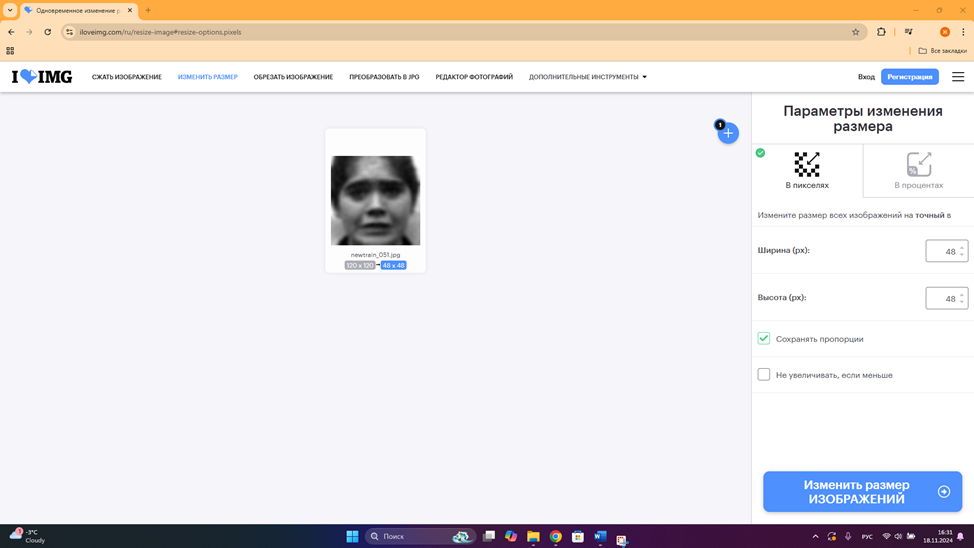
This kind of dataset augmentation allowed the model to be exposed to a wider range of disgust and fear examples, which improved its ability to learn these emotions.

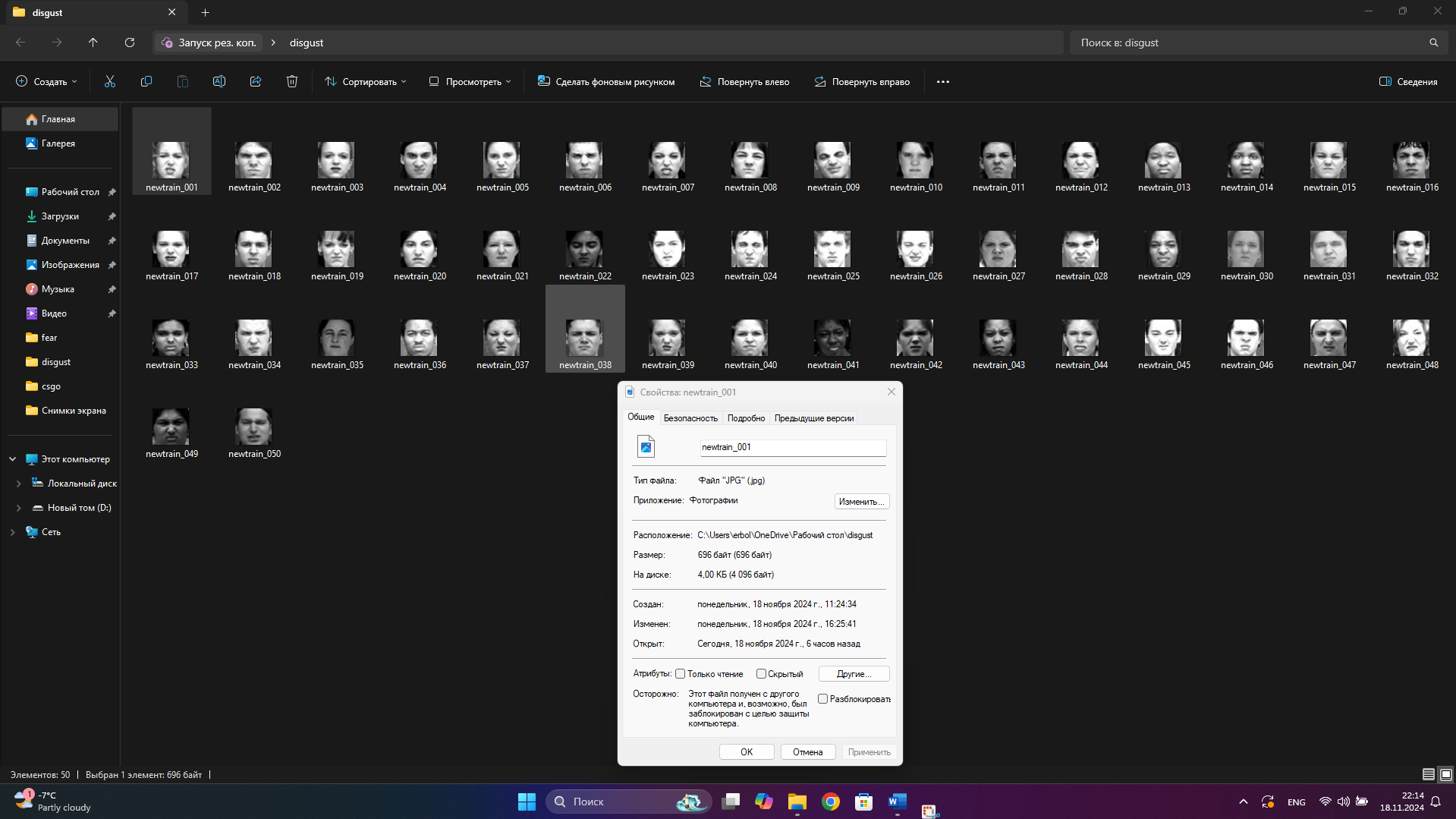
As a result:

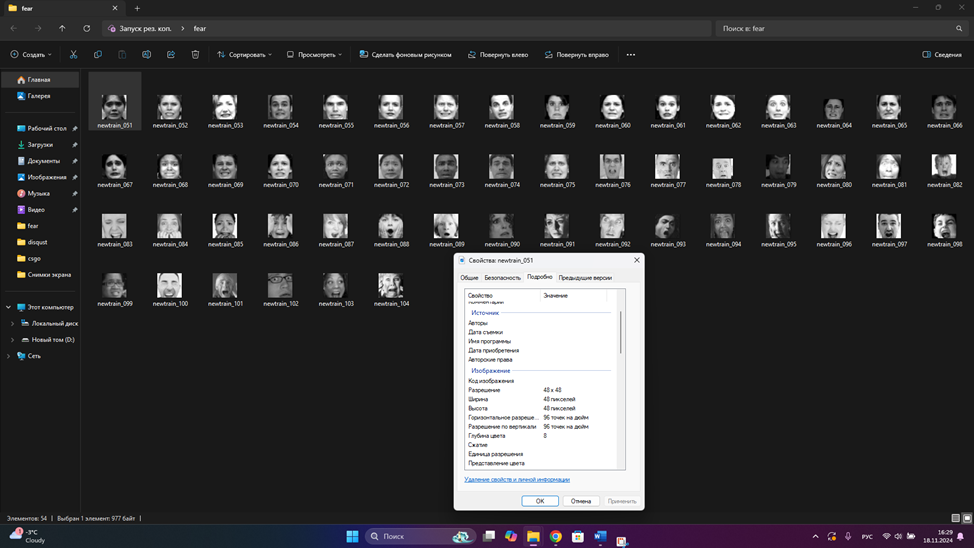
The model got more adept at recognizing disgust and fear, as seen by the notable improvement in recall ratings.

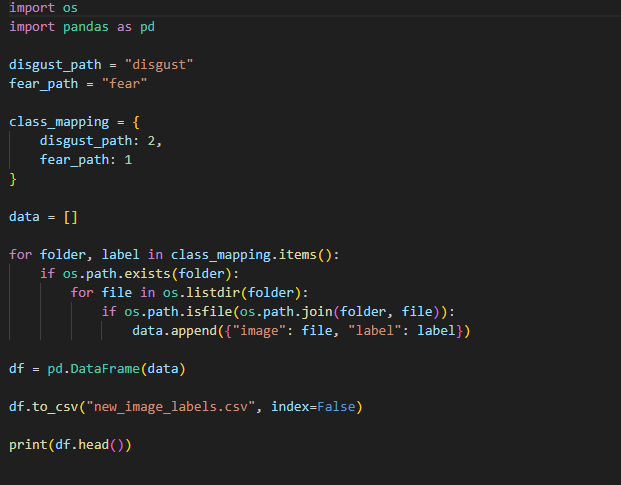
All emotional categories may be accurately and fairly recognized thanks to the overall improvement in model performance.

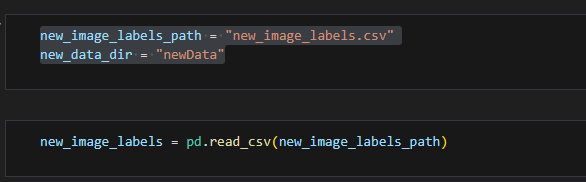
In addition to strengthening our project and improving the dataset, this manual augmentation made it more appropriate for emotion recognition tasks in the real world.



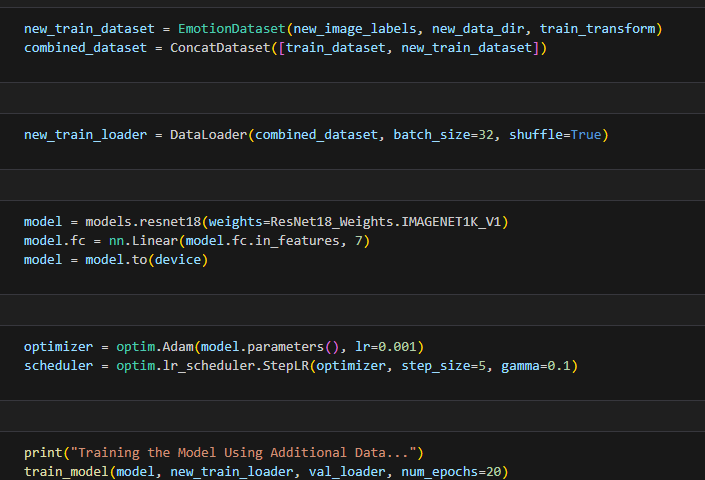


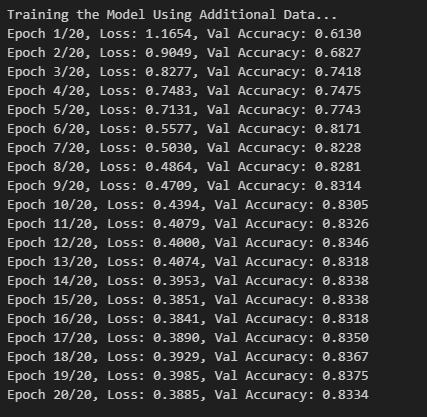


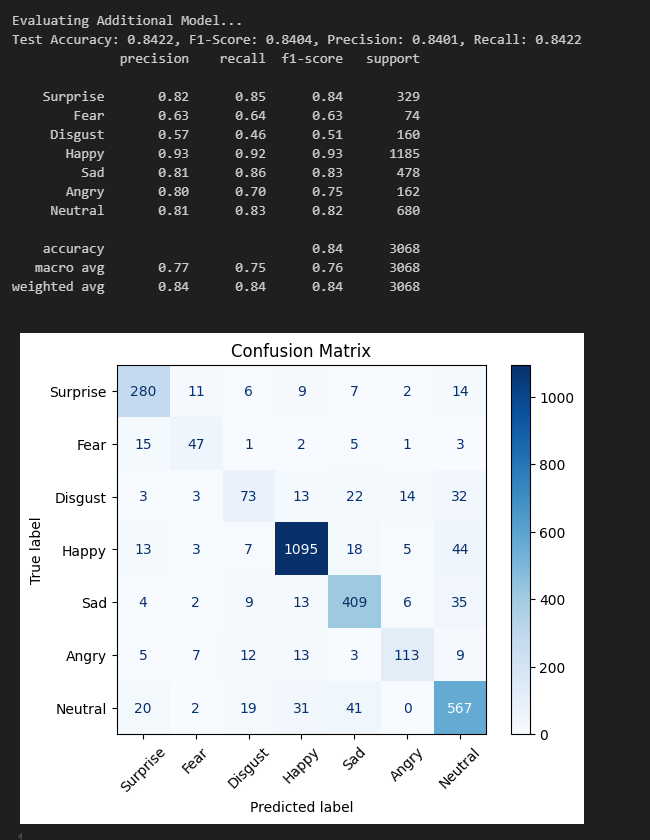




We are loading a fresh dataset in this code for additional analysis or processing. The file path for a CSV file (`new\_image\_labels.csv`) that contains labels for the dataset's images is specified by the variable `new\_image\_labels\_path`. The directory (`newData`) containing the corresponding photos is referenced by the `new\_data\_dir` variable. To access and manage the image labels programmatically, we use `pandas` to read the CSV file into a DataFrame called `new\_image\_labels`. With this configuration, the data is ready for activities like testing, training, and model inference.

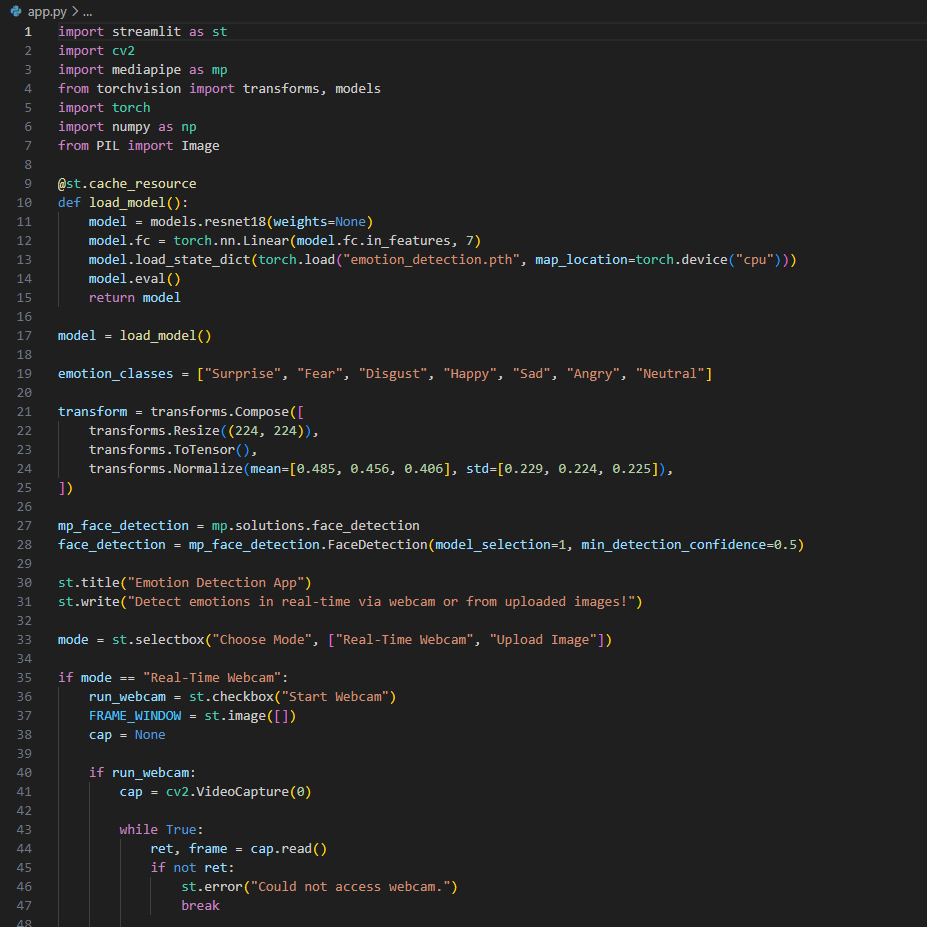


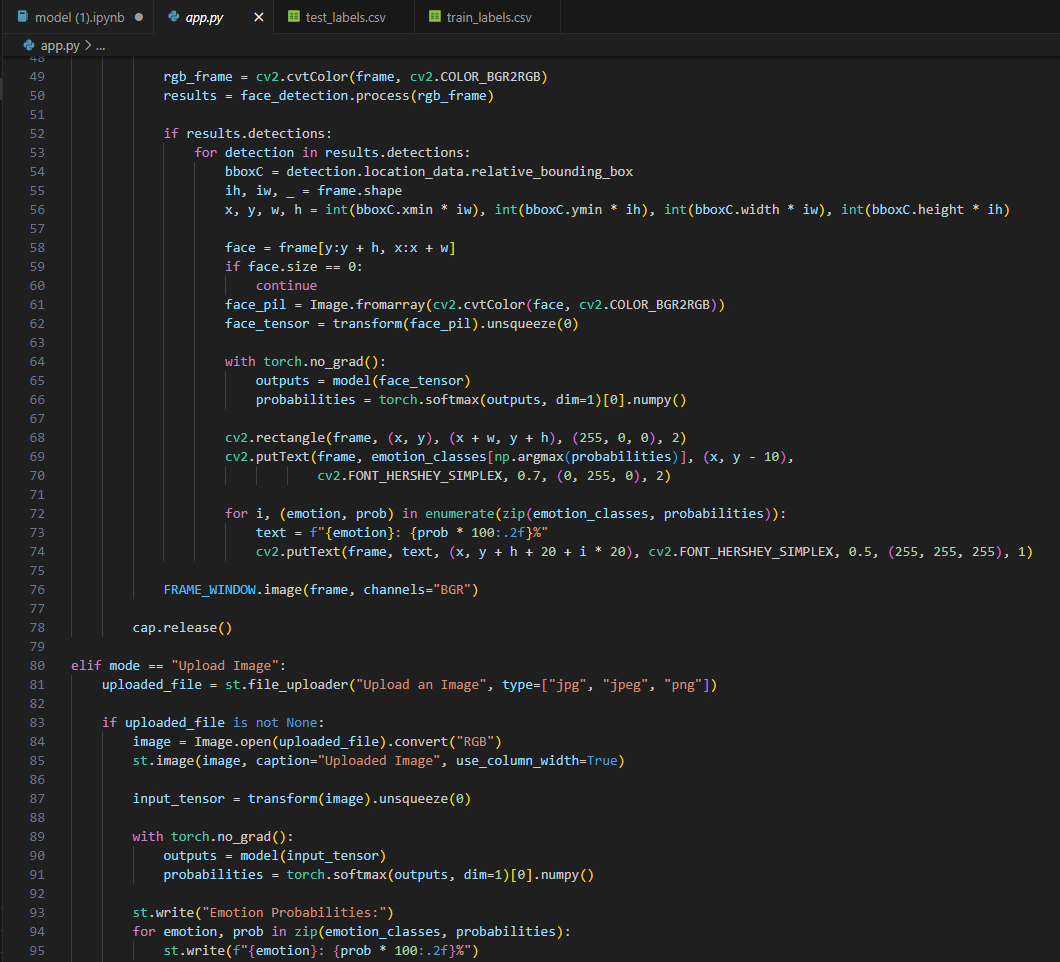


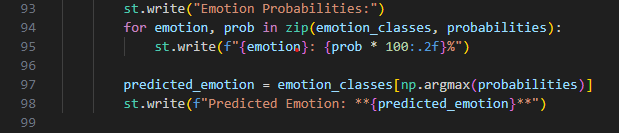


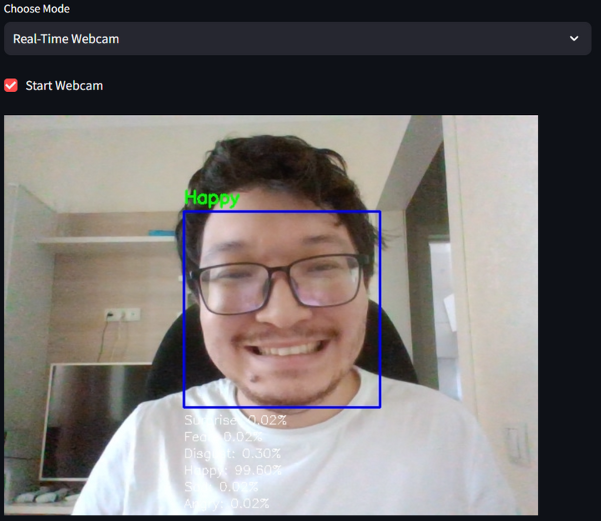
We can see that the new data balances the classes a bit, but does not affect the learning results significantly. Nevertheless, accuracy and all other indicators show high performance.

**Real-Time Emotion Detection: Application Overview**









Using Streamlit, we created an interactive and captivating emotion detection app that skillfully combines deep learning and computer vision. Through the use of either real-time webcam streaming or image uploads, our program enables users to identify emotions from face expressions. We extract the identified face regions, preprocess them with transformations, then feed them into a pretrained ResNet-18 model optimized for emotion classification, utilizing MediaPipe's Face Detection for accurate facial localization. Seven categories of emotions are predicted by the algorithm, which highlights the dominating emotion and presents the results as probabilities. In order to provide an immersive experience, we utilize OpenCV to record camera frames, apply bounding boxes around identified faces, and label emotions straight on the video feed in the real-time mode.