Facial Emotion Recognition

Project Details

Table of Contents

---

1. Introduction

This project is about building a system that can recognize emotions on people's faces using pictures and realtime camera video input. It's called facial emotion recognition, and it's useful in many areas like making computers understand humans better, helping in mental health studies, or even in security systems. We used a dataset called AffectNet, which has lots of face images, and a model architecture called ResEmoteNet to identify seven emotions: neutral, happiness, sadness, surprise, fear, disgust, and anger. This document explains everything about the project, including the dataset, the model, how we explored the data, and how the code works.

---

2. Dataset: AffectNet

2.1. Overview of AffectNet

AffectNet is a big collection of face images used for studying emotions. It has over 1 million pictures that were collected from the internet by searching with 1,250 emotion-related words in six different languages on three major search engines. Out of these, about 440,000 images were carefully labeled by people to show seven main emotions: neutral, happiness, sadness, surprise, fear, disgust, and anger, plus an extra emotion called contempt. The dataset also measures two other things called valence (how positive or negative the emotion is) and arousal (how intense the emotion is). This makes AffectNet special because it's one of the largest datasets for this kind of work and supports two ways of looking at emotions: as separate categories (like happiness or anger) and as a range of feelings (using valence and arousal). For this project, we focused on the seven emotions and did not use contempt because it's less common in studies like ours.

2.2. Class Distribution

The images in AffectNet are not evenly spread across all emotions. Some emotions have more pictures than others, which can make it harder to train a model. Below is a table showing how many images are in the training and validation sets for each category, including some extra categories like “None,” “Uncertain,” and “Non-Face” that we didn’t use in our project.

Emotion	**Number of Images**
Neutral	75,374
Happy	134,915
Sad	25,959
Surprise	14,580
Fear	6,878
Disgust	4,303
Anger	25,382
Contempt	4,250
None	33,588
Uncertain	12,145

Emotion	Number of Images
Non-Face	82,915
Total	**420,299**

From the table, we can see that “Happy” has the most images (134,915), while “Disgust” has the fewest (4,303) among the emotions we used. This imbalance means the model might get better at recognizing happiness but struggle with emotions like disgust unless we handle it carefully during training.

---

3. Model Architecture: ResEmoteNet

3.1. Structure of ResEmoteNet

ResEmoteNet is the model we used to recognize emotions from face images. It’s a deep learning model built using a library called PyTorch, and it has several parts that work together to look at an image and figure out the emotion. Let’s break it down step by step, using the code to explain each part in detail with some math to make it clear.

Initial Convolutional Layers

The model starts with a few layers that look at the image and find basic patterns, like edges or shapes. These are called convolutional layers. In the code, we see three of them:

- **First Convolutional Layer**: The model takes an image with 3 color channels (red, green, blue) and applies a convolutional layer

(`self.conv1`) with 64 filters, each of size 3x3, and adds padding of 1 to keep the image size the same. This is followed by batch normalization (`self.bn1`) to make training stable, and a ReLU activation (`F.relu`) to add non-linearity, meaning it helps the model learn more complex patterns by setting negative values to zero. The output size here is 64 channels (because of 64 filters), and the image size stays the same, say 64x64 if that's the input size. Mathematically, this step is: [X_1 = (((X)))] where (X) is the input image of size (3, 64, 64), and (X_1) has size (64, 64, 64). Then, a max-pooling layer (`F.max_pool2d`) reduces the image size by half to (64, 32, 32) by taking the maximum value in each 2x2 area.

- **Second Convolutional Layer**: Next, the model applies another convolutional layer (`self.conv2`) with 128 filters, again with a 3x3 size and padding of 1. This increases the number of channels to 128, so the output size is (128, 32, 32). It's followed by batch normalization (`self.bn2`), ReLU, and another max-pooling, which reduces the size to (128, 16, 16). This step is: [X_2 = (((X_1)))] followed by max-pooling.

- **Third Convolutional Layer**: The third layer (`self.conv3`) uses 256 filters, so the output has 256 channels, with size (256, 16, 16). After batch normalization (`self.bn3`), ReLU, and max-pooling, the size becomes (256, 8, 8). This is: [X_3 = (((X_2)))]

The code also adds a dropout layer (`self.dropout1`) with a 20% chance of ignoring some parts of the data during training. This helps the model not to rely too much on any one part, making it better at handling new images.

Squeeze-and-Excitation (SE) Block

After the third convolutional layer, the model uses a special part called the Squeeze-and-Excitation (SE) block (`self.se = SEBlock(256)`). This block

helps the model focus on the most important features, like the eyes or mouth, which are key for emotions. Here’s how it works:

- **Squeeze**: The SE block first takes the output from the third layer, which is size (256, 8, 8), and shrinks it using global average pooling (`self.avg_pool`). This means it takes the average of each channel across all 8x8 pixels, giving a single number for each of the 256 channels. So, the output is a vector of size (256, 1, 1). Mathematically: [z_c = {i=1}^{H} {j=1}^{W} X_3(c, i, j)] where (H = W = 8), and (c) is the channel number (from 1 to 256).

- **Excitation**: Next, the model uses two small layers (`self.fc`) to decide which channels are important. The first layer reduces the 256 numbers to 16 (256 divided by a reduction factor of 16), applies ReLU, and then the second layer brings it back to 256 numbers and uses a sigmoid function to get values between 0 and 1. These values act like weights, showing how important each channel is. This is: [s_c = ((((z))))] where (s_c) is a weight for channel (c).

- **Scaling**: Finally, the SE block multiplies the original output (X_3) by these weights, so important channels get more focus. The output size stays (256, 8, 8). This is: [X_{SE} = X_3 s]

Residual Blocks

The next part of the model uses three residual blocks (`self.res_block1`, `self.res_block2`, `self.res_block3`). These blocks help the model learn better by allowing it to skip some steps if needed, which is useful for deep models. Each block has two convolutional layers and a shortcut connection:

- **Residual Block Structure**: In the code, a `ResidualBlock` takes an input with some number of channels (like 256), applies a 3x3

convolution (`self.conv1`) with a stride (step size) that can shrink the image, then batch normalization and ReLU. It applies another 3x3 convolution (`self.conv2`) with stride 1, and batch normalization again. The shortcut connection (`self.shortcut`) matches the input and output sizes if needed by using a 1x1 convolution. The output is the sum of the main path and the shortcut, followed by ReLU: [= ((((((X)))))) + (X))]

- **First Residual Block**: This block (`self.res_block1`) takes 256 channels and outputs 512 channels, with a stride of 2, so the image size shrinks from (256, 8, 8) to (512, 4, 4).
- **Second Residual Block**: This one (`self.res_block2`) takes 512 channels, outputs 1024 channels, and shrinks the size to (1024, 2, 2) with stride 2.
- **Third Residual Block**: The last block (`self.res_block3`) takes 1024 channels, outputs 2048 channels, and shrinks the size to (2048, 1, 1) with stride 2.

Pooling and Final Layers

At the end, the model uses adaptive average pooling (`self.pool`) to shrink the output to a fixed size of (2048, 1, 1), no matter the input size. Then, it flattens this into a list of 2048 numbers. This is followed by a few fully connected layers:

- **First Fully Connected Layer**: This layer (`self.fc1`) takes the 2048 numbers and reduces them to 1024, with ReLU and a dropout of 50% (`self.dropout2`): [X_{FC1} = ((X_{}))]
- **Second Fully Connected Layer**: This layer (`self.fc2`) reduces from 1024 to 512, with ReLU and dropout: [X_{FC2} = ((X_{FC1}))]

- **Third Fully Connected Layer**: This layer (`self.fc3`) reduces from 512 to 256, with ReLU and dropout: [X_{FC3} = ((X_{FC2}))]
- **Final Layer**: The last layer (`self.fc4`) takes the 256 numbers and outputs 7 numbers, one for each emotion: [= (X_{FC3})] These 7 numbers are the scores for each emotion, and the highest score tells us the predicted emotion.

3.2. Performance Metrics

ResEmoteNet works well for this task. According to studies, it can correctly identify emotions 72.39% of the time on the AffectNet dataset with seven emotions. It was also tested on other datasets, and here are the results:

- FER2013 dataset: 79.79% accuracy
- RAF-DB dataset: 94.76% accuracy
- AffectNet-7 dataset: 72.39% accuracy
- ExpW dataset: 75.67% accuracy

These numbers show that ResEmoteNet is good at recognizing emotions compared to other models. The SE blocks and residual blocks make a big difference, as removing them lowers the accuracy a lot.

---

4. Exploratory Data Analysis (EDA)

4.1. Class Distribution Analysis

We first checked how many images belong to each emotion. The class distribution was unbalanced. “Happy” has the most samples (134,915), while “Disgust” has the fewest (4,303). This imbalance may cause the model to learn to favor predicting “happy” more often.

To address this, we used techniques like: - Monitoring per-class accuracy during training. - Using class weights in the loss function.

4.2. Sample Image Visualization

We viewed a few random images from each emotion category to:

- Check **visual clarity** (some images were blurry or poorly lit).
- Verify **label correctness** (e.g., some “surprise” images looked more like “fear”).
- Understand **facial expressions** better, e.g., smiling for happiness, wide eyes for surprise, furrowed brows for anger.

This step helped ensure data quality before model training.

4.3. Face Detection Quality

We used Haar Cascades to detect and crop faces from images before feeding them into the model.

During EDA, we found: - Some images had **no detectable face** and had to be discarded. - Others had **multiple faces**, and only the largest was kept. - A few had **poor cropping**, missing parts of the face like the chin or forehead.

Ensuring accurate face detection improved the model’s understanding of facial features.

4.4. Valence and Arousal Patterns

Even though we didn’t use valence and arousal for training, plotting them revealed patterns: - **Happiness** had high valence (positive) and moderate

arousal. - **Fear** and **Anger** had high arousal but low valence (negative). - **Neutral** was centered near zero on both scales.

This gave us extra confidence that the labels and images were consistent with expected emotional behavior.

---

5. Code Implementation

Folder Structure

A well-organized folder structure is important for a project like this to keep everything neat and easy to manage. Based on the code, here's a typical folder structure for your project:

```
facial-emotion-recognition/
│
├── nn_architecture/
│   └── ResEmote_Net.py          # Defines the ResEmoteNet model architecture
│
├── utils/
│   ├── face_detection.py        # Handles face detection and preprocessing
│   ├── inference.py             # Contains the EmotionPredictor class for making predictions
│   ├── model_downloader.py      # Manages downloading the pre-trained model
│   └── results_manager.py       # Saves prediction results to a CSV file
│
```

```
├── static/
│   ├── style.css                # CSS styles for the Streamlit
app
│   ├── facial-recognition.png   # Logo image for the sidebar
│   └── confidence_scores.html   # HTML template for displaying
confidence scores
│
├── trained_model/
│   └── AffectNet7_Model.pth     # Pre-trained model file
(downloaded)
│
├── app.py                       # Main Streamlit app file
│
└── results.csv                  # Stores prediction results
```

Explanation of Folders and Files

- **nn_architecture**: This folder holds the code for the ResEmoteNet model.
- **utils**: This folder contains helper scripts for face detection, inference, model downloading, and result management.
- **static**: This folder has files for the Streamlit app's appearance, like CSS, images, and HTML templates.
- **trained_model**: This folder stores the pre-trained model file.
- **app.py**: The main file that runs the Streamlit app.
- **results.csv**: A file where prediction results are saved.

Now, let's go through each file in detail, explaining its purpose and how the code works.

Detailed Code Explanation

1. `nn_architecture/ResEmote_Net.py`

This file defines the ResEmoteNet model, which is the core of the project. It has three main classes: `SEBlock`, `ResidualBlock`, and `ResEmoteNet`.

`SEBlock` Class

The `SEBlock` class creates a Squeeze-and-Excitation block, which helps the model focus on important features in the image, like the eyes or mouth, by giving more weight to useful channels.

- **Initialization (`__init__`)**:
  - Takes `in_channels` (number of input channels, e.g., 256) and a `reduction` factor (default 16).
  - Creates an adaptive average pooling layer (`self.avg_pool`) to shrink the image to 1x1 per channel.
  - Sets up two linear layers (`self.fc`):
    - First layer reduces the number of channels to `in_channels // reduction` (e.g., 256 to 16).
    - Second layer brings it back to `in_channels` (e.g., 16 to 256) and applies a sigmoid function to get weights between 0 and 1.
- **Forward Pass (`forward`)**:
  - Takes an input tensor `x` of shape (batch_size, channels, height, width), e.g., (batch_size, 256, 8, 8).
  - Applies global average pooling to get a vector of size (batch_size, channels, 1, 1).
  - Flattens this to (batch_size, channels) and passes it through the two linear layers to get weights.

- Multiplies the original input `x` by these weights to focus on important channels.
- Returns the scaled tensor, same size as the input.

`ResidualBlock` Class

The `ResidualBlock` class creates a residual block, which helps the model learn better by adding a shortcut connection.

- **Initialization (`__init__`)**:
  - Takes `in_ch` (input channels), `out_ch` (output channels), and `stride` (default 1).
  - Creates two convolutional layers:
    - `self.conv1`: 3x3 convolution with stride, changes channels from `in_ch` to `out_ch`.
    - `self.conv2`: 3x3 convolution with stride 1, keeps channels as `out_ch`.
  - Adds batch normalization after each convolution (`self.bn1`, `self.bn2`).
  - Creates a shortcut path (`self.shortcut`):
    - If `stride != 1` or `in_ch != out_ch`, uses a 1x1 convolution to match the input and output sizes.
- **Forward Pass (`forward`)**:
  - Takes an input tensor `x`.
  - Passes it through `conv1`, batch normalization, and ReLU.
  - Passes the result through `conv2` and batch normalization.
  - Adds the shortcut path output to the main path.
  - Applies ReLU and returns the result.

`ResEmoteNet` Class

The `ResEmoteNet` class puts everything together to create the full model.

- **Initialization (`__init__`)**:
  - Sets up three convolutional layers:
    - `self.conv1`: 3x3, 3 to 64 channels.
    - `self.conv2`: 3x3, 64 to 128 channels.
    - `self.conv3`: 3x3, 128 to 256 channels.
  - Adds batch normalization after each (`self.bn1`, `self.bn2`, `self.bn3`).
  - Creates an SE block (`self.se`) for 256 channels.
  - Sets up three residual blocks:
    - `self.res_block1`: 256 to 512 channels, stride 2.
    - `self.res_block2`: 512 to 1024 channels, stride 2.
    - `self.res_block3`: 1024 to 2048 channels, stride 2.
  - Adds an adaptive average pooling layer (`self.pool`) to shrink the output to (2048, 1, 1).
  - Creates fully connected layers:
    - `self.fc1`: 2048 to 1024.
    - `self.fc2`: 1024 to 512.
    - `self.fc3`: 512 to 256.
    - `self.fc4`: 256 to 7 (for the 7 emotions).
  - Adds dropout layers: `self.dropout1` (20%) after early layers, `self.dropout2` (50%) after fully connected layers.
- **Forward Pass (`forward`)**:
  - Takes an input tensor `x` of shape (batch_size, 3, 64, 64).
  - Passes it through `conv1`, batch normalization, ReLU, max-pooling (to 32x32), and dropout.

- Passes through `conv2`, batch normalization, ReLU, max-pooling (to 16x16), and dropout.
- Passes through `conv3`, batch normalization, ReLU, max-pooling (to 8x8).
- Applies the SE block.
- Passes through the three residual blocks, shrinking the size to (2048, 1, 1).
- Applies adaptive average pooling and flattens to (batch_size, 2048).
- Passes through the fully connected layers with ReLU and dropout, finally outputting (batch_size, 7).

---

2. `utils/face_detection.py`

This file handles face detection and preprocessing of images before they're fed into the model.

- **Transformations (`transform`)**:
  - Defines a set of transformations using `torchvision.transforms`:
    - Resizes the image to 64x64.
    - Converts to grayscale but duplicates to 3 channels (for compatibility with the model).
    - Converts the image to a tensor.
    - Normalizes the pixel values using mean [0.485, 0.456, 0.406] and standard deviation [0.229, 0.224, 0.225], which are standard values for models trained on ImageNet-like data.
- **Face Detection (`detect_and_crop_face`)**:
  - Takes an image (in NumPy array format, BGR color from OpenCV).
  - Converts it to grayscale.

- Uses OpenCV's Haar cascade classifier (`face_cascade`) to detect faces with parameters:
  - `scaleFactor=1.1`: How much the image size is reduced at each scale.
  - `minNeighbors=5`: How many neighbors each candidate rectangle should have.
  - `minSize=(30, 30)`: Minimum face size.
- If a face is found, crops the face region and converts it to a PIL image (RGB format).
- Returns the cropped face image or `None` if no face is detected.

- **Preprocessing (`preprocess_image`)**:
  - Takes a PIL image (the cropped face).
  - Applies the transformations defined in `transform`.
  - Returns a tensor of shape (3, 64, 64) ready for the model.

---

3. `utils/inference.py`

This file contains the `EmotionPredictor` class, which uses the trained model to make predictions.

- **Initialization (`__init__`)**:
  - Takes a `checkpoint_path` (path to the pre-trained model file) and `device` (CPU or GPU).
  - Creates a `ModelDownloader` object to download the model if needed.
  - Initializes the `ResEmoteNet` model and moves it to the specified device.
  - Loads the pre-trained weights from the checkpoint file using `torch.load`.
  - Sets the model to evaluation mode (`self.model.eval()`).

    - Defines the list of emotion labels: [“neutral”, “happiness”, “sadness”, “surprise”, “fear”, “disgust”, “anger”].
- **Prediction (`predict`)**:
    - Takes an input tensor of shape (1, 3, 64, 64).
    - Runs the tensor through the model without calculating gradients (`with torch.no_grad()`).
    - Applies the softmax function (`torch.softmax`) to get probabilities for each emotion.
    - Finds the emotion with the highest probability using `torch.argmax`.
    - Converts the predicted class to a label (e.g., “happiness”).
    - Converts probabilities to percentages.
    - Returns the predicted label and the probabilities as a NumPy array.

---

4. `utils/model_downloader.py`

This file manages downloading the pre-trained model file.

- **Initialization (`__init__`)**:
    - Sets the Google Drive file ID and model name (`AffectNet7_Model.pth`).
    - Defines the path to save the model in the `trained_model` folder.
- **Download Model (`download_model`)**:
    - Checks if the model file already exists locally.
    - If not, downloads it from Google Drive using the gdown library.
    - Returns the path to the model file.
    - Includes error handling for download failures.
- **Verify Model (`verify_model`)**:
    - Tries to load the model file using `torch.load` to check if it’s valid.

    - If it fails, deletes the file and returns `False`.
- **Other Methods**:
    - `verify_model_exists`: Checks if the model file exists and is valid.
    - `get_model_path`: Returns the path to the model file.

---

5. `utils/results_manager.py`

This file saves prediction results to a CSV file.

- **Initialization (`__init__`)**:
    - Takes a filename (`results.csv` by default).
    - Loads existing results from the file into a list if the file exists.
- **Add Result (`add_result`)**:
    - Takes the image name, predicted label, and probabilities.
    - Creates a dictionary with the image name, predicted emotion, and probabilities for each emotion (as percentages).
    - Adds the dictionary to the results list.
    - Saves the updated results to the CSV file using `pandas`.
- **Get Results (`get_results`)**:
    - Returns the results as a `pandas` DataFrame.

---

6. `static/confidence_scores.html`

This file is an HTML template for displaying confidence scores as animated bars in the Streamlit app.

- **HTML Structure**:
    - Defines a section with a heading “Confidence Scores”.
    - Uses a placeholder `{{html_content}}` for the bars, which will be filled dynamically.

    - Each bar has a label (e.g., “Happiness”), a bar container, and a percentage.
- **CSS Styles**:
    - Styles the bars with a modern look: orange bars, rounded edges, and a smooth animation for the bar width.
    - Uses flexbox for layout and sets colors for text and bars.
- **JavaScript**:
    - Takes probabilities from a JSON string (`{{probs_json}}`).
    - Updates the width of each bar based on the probability, with a 1-second animation.
    - Runs when the page loads (`DOMContentLoaded` event).

---

7. `app.py`

This is the main file that runs the Streamlit app, providing a user interface for emotion recognition.

Setup

- Sets up the Streamlit page with a title and layout.
- Loads CSS styles from `static/style.css`.
- Chooses the device (GPU if available, else CPU) using `torch.device`.
- Initializes the `EmotionPredictor` with the model path and device.
- Creates a `ResultsManager` to save results.

Helper Functions

- **Get Emotion Class (`get_emotion_class`)**:
    - Converts the predicted emotion to a CSS class (e.g., “happiness” to “emotion-happiness”) for styling.

- **Display Confidence Scores (`display_confidence_scores`)**:
    - Takes the probabilities and creates animated bars using the HTML template.
    - Sorts the probabilities from highest to lowest.
    - Fills the HTML template with the bar data and probabilities as JSON.
    - Renders the HTML in Streamlit using `components.html`.

Sidebar Menu

- Adds a sidebar with a logo (`static/facial-recognition.png`) and a menu using `streamlit_option_menu`.
- Menu options: “About & Instructions”, “Single Image(s)”, “Multiple Images (ZIP)”, “Camera Input”, “Results”.
- Styles the sidebar with CSS for a centered logo and menu.

Sections

- **About & Instructions**:
    - Displays a title, description of the app, supported emotions, instructions, and a warning about needing visible faces.
- **Single Image(s)**:
    - Allows uploading multiple images.
    - For each image:
        - Detects and crops the face.
        - If no face is found, shows an error.
        - Displays the cropped face.
        - Preprocesses the image and predicts the emotion.
        - Shows the predicted emotion and confidence scores.
        - Saves the result using `ResultsManager`.

- **Multiple Images (ZIP)**:
  - Allows uploading a ZIP file with images.
  - Extracts the images to a temporary folder.
  - Processes each image like in the single image section.
  - Cleans up the temporary folder.
- **Camera Input**:
  - Warns that camera input only works locally, not on Streamlit Cloud.
  - Uses OpenCV to capture video from the webcam.
  - For each frame:
    - Detects and crops the face.
    - Predicts the emotion and displays it on the frame.
    - Shows confidence scores.
  - Stops when the user unchecks “Start Camera”.
- **Results**:
  - Displays the saved results as a table using `pandas`.
  - Allows downloading the results as a CSV file.