



COMP 472 - Project Part #2

Report #2

Team OB_05

Bryan Carlo Miguel 40231530

Yasser Ameer 40212780

William Nazarian 40213100

GitHub: <https://github.com/WilliamNazarian/Comp472Ai.git>

We certify that this submission is the original work of members of the group and meets the Faculty's Expectations of Originality

Table Of Contents

Dataset	4
Provenance Information.....	5
• Dataset Overview.....	5
• Data Description	5
• Data Collection	6
• Licensing and Permissions.....	6
Data Cleaning.....	7
• 1. Techniques and Methods	7
• 2. Challenges and Solutions.....	7
• 3. Example Images.....	7
Data Labeling.....	8
• 1. Method Used.....	8
• 2. Challenges faced.	8
Dataset Visualization	10
• 1. Class Distribution.....	10
• 2. Pixel Intensity Distribution	11
• 3. Sample Image	12
Report: CNN Architecture and Evaluation	16
Main Model Overview:.....	16
• List of layers with their types, sizes, and activation functions.....	16
• Discussion of any design nuances:	16
Variant 1:	18
Variant 2:	19
Training Process.....	20
• 1. Training Methodology:	20
• 2. Optimization Techniques:	20
Performance Metrics.....	21
• Evaluation	21
Main Model.....	22
Variant 1	23
Variant 2	24
• Common Patterns.....	24
• Overall Comparison	25
Impact of Architectural Variations.....	25
Conclusions	26
References	27

Table 1 Total of number of Images.....	4
Table 2 Source provenance:	5
Table 3 CSV file:	5
Table 4 example of images:	7
Table 5 Area of focus:.....	9
Table 6 Macro vs Micro:.....	21
Table 7 Confusion Matrix for Main Model	22
Table 8 Main Model Performance Metrics:	22
Table 9 Confusion Matrix Variant 1	23
Table 10 Variant 1 Performance Metrics:	23
Table 11 Confusion Matric Table Variant 2 and the Performance Metrics:	24
Table 12 Overall Comparison:	25
Figure 1 Engaged/Focused.....	9
Figure 2 Class distribution.	10
Figure 3 Pixel Intensity Distribution.....	11
Figure 4 Sample Images Engaged.....	12
Figure 5 Sample Images Happy	13
Figure 6 Sample Images Neutral.....	14
Figure 7 Sample Images Anger.....	15
Figure 8 Main Model design.....	17
Figure 9 Variant 1 Model.....	18
Figure 10 Variant 2 Model.....	19

Dataset

The project began by searching for an appropriate dataset. After considering many datasets (ex. FER2013, CK+, expW - to name a few), we decided to use the [AffectNet](#) dataset. Some notable characteristics about this dataset is its variety in demographic, backgrounds, face shot orientations, and lighting conditions.

The dataset is already pre-labelled into eight classes as displayed in the table below:

Table 1 Total of number of Images

Class	Number of images
Anger	3218
Contempt	2871
Disgust	2477
Fear	3176
Happy	5044
Neutral	5126
Sad	3091
Surprise	4039

The dataset includes individuals across a variety of age groups, from toddlers to elderly adults. Additionally, the dataset includes individuals from multiple ethnic backgrounds, such as Asian, African, Hispanic, etc. This demographic variety closely reflects the variety of students that can be found in classrooms. Furthermore, the dataset is not just limited to frontal face shots. There are plenty of images with a $\frac{3}{4}$ -profile shot. Images in the dataset also contain diverse backgrounds, with a wide range of settings and environments. This variety in the dataset reflects “real-world” situations closely, which could improve the performance of our model by making it more able to deal with noise and differences.

Despite the benefits that this variety provides, it can also pose some challenges. A potential problem would be overfitting due to the model focusing on very specific patterns, or underfitting due to the model not recognizing enough patterns. Additionally, the model may be more complex in order to handle the variability. One final potential problem may be a potential bias towards certain demographics. Regardless of these challenges, we believe that the dataset will provide a strong foundation to train a robust model with “real-world” applications, capable of dealing with noise and variances.

Provenance Information

- **Dataset Overview**

Table 2 Source provenance:

Name	FER AffectNet
Source	Kaggle
URL(s)	https://www.kaggle.com/datasets/noamsegal/affectnet-training-data/ https://web.archive.org/web/20240205175353/http://mohammadmahoor.com/affectnet/
Author(s)	Ali Mollahosseini, Behzad Hasani, and Mohammad H. Mahoor, Noam Segal
Temporal Coverage Start Date	12/30/1938
Temporal Coverage End Date	12/30/2022
Date of Download	5/24/2024

- **Data Description**

The dataset is organized as a collection of 96×96px .png files, each categorized into the eight classes mentioned before. Additionally, the dataset comes with a .csv containing index ‘i’ of the image, path of the image file in the dataset, label/class, and the PFC%s. In this case, the PFC%s represents the monochromaticity of an image. A high PFC%s (~>99%) means that an image is grayscale, while a lower PFC%s (~<80%) means that an image has plenty more RGB values. Below is a subset of the contents of the .csv file:

Table 3 CSV file:

#	Path	label	relFCs
0	anger/image0000006.jpg	surprise	0.8731421290934949
1	anger/image0000060.jpg	anger	0.852310783018639
2	anger/image0000061.jpg	anger	0.80095684657714
...

We decided to select images with a relFC value near 1 to ensure high-quality images overall. Some problems we faced with this dataset included having the same person in different poses within the same class. This needed to be addressed to maintain consistency. Additionally, we had to filter out irrelevant pictures and ensure that each folder contained images reflecting the correct emotions.

The relFC value was particularly helpful for us in determining if an image was clear and effectively expressed the person's emotions.

- **Data Collection**

The images were collected by querying three major search engines using 1250 keywords in six different languages [2]. The images were collected ‘in the wild’, meaning that the images were not collected in a controlled and posed environment [1]. There is no information on where exactly each individual image is sourced from.

- **Licensing and Permissions**

The dataset is licensed under the “[Attribution-NonCommercial-ShareAlike 3.0 IGO \(CC BY-NC-SA 3.0 IGO\)](#)” licence.

Data Cleaning

- **1. Techniques and Methods**

To begin with, after organizing our dataset and personal images into proper directories, we needed to standardize the dataset. To do this, we created a script to resize every image to 96 x 96 pixels, ensuring uniformity in image size.

According to a research paper, converting images from RGB to grayscale can achieve better classification accuracy using genetic algorithms.[3] To optimize our accuracy, we wrote a script to transform the images into black and white. This script also performs histogram equalization to adjust the lighting. At the end of the process, the script stores the processed images in a new folder.

- **2. Challenges and Solutions**

This method involved using OpenCV to adjust the brightness and color of the images, ensuring they were consistently prepared for our project. Using OpenCV in the Anaconda environment was a bit difficult for us, as we had no prior experience with it. We had to learn how to use OpenCV and integrate it into our workflow.

- **3. Example Images**

Here is some of the examples of image for each category we have done:

Table 4 example of images:

Before	After
	
	
	

Data Labeling

- **1. Method Used**

One of the emotions that the team wasn't able to find was engagement and focus. To address this, we had to select a bunch of images from a chosen dataset, specifically from the contempt and surprise folders, and use the suggested website LabelBox. We went through around 1500 images to pick about 500 that seemed to depict engaged and focused facial emotions.

LabelBox provided several useful features that allowed us to verify and label the images we wanted to use. We could then export the labeled data in a JSON file, enabling us to extract the image file names and update our CSV file, changing "contempt" to "engagement." This helped us build a well-structured dataset.

With this data, we were able to create a Python script that filtered the images into the correct categories, ultimately giving us a new folder containing only images for the engaged and focused emotion.

Finding a dataset with decent-sized images wasn't easy. Most of the images were 96 x 96, which doesn't provide the best resolution possible. For this reason, we decided to manually review all the chosen data to ensure they had good resolution, reducing ambiguity in later phases of the project.

The dataset we chose mostly contained images with a single person, which is crucial for our project's accuracy and functionality. We needed to ensure that all images in the dataset featured only one person and did not include multiple faces. Manual labeling in LabelBox helped us achieve this requirement.

- **2. Challenges faced.**

Regarding our chosen dataset, we were able to find everything we needed within a single dataset that had several folders representing different classes of emotions. This meant we didn't face any issues with handling multiple datasets or mapping those classes.

One challenge we faced during the labeling phase was that the criteria for identifying engaged facial emotions differed from those for other emotions. For example, we labeled an image from the contempt folder as an engaged image.


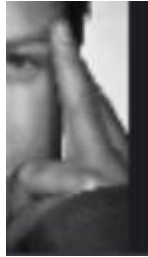



Figure 1 Engaged/Focused

We had to ensure that the engaged facial expressions didn't resemble other classes, such as neutral faces. Many of the engaged images gave the impression of being neutral, which could cause inaccuracies later in the project. It was crucial for us to accurately label these emotions to maintain the integrity of our data.

We also needed to determine how the AI would recognize an emotion and establish the criteria for this recognition. For example, engaged facial expressions might be identified by specific features such as slightly raised eyebrows, focused eyes, and subtle mouth movements, distinguishing them from neutral expressions. Establishing clear criteria like this is essential for accurate emotion recognition.

Table 5 Area of focus:

Face Part	Image
Mouth	
Hand gesture	
eyes/ forehead/ eyebrows	

Dataset Visualization

- **1. Class Distribution**

This part of the report was conducted using some Python scripts that can be found in a folder in the GitHub repository or in the zip file called "scripts".

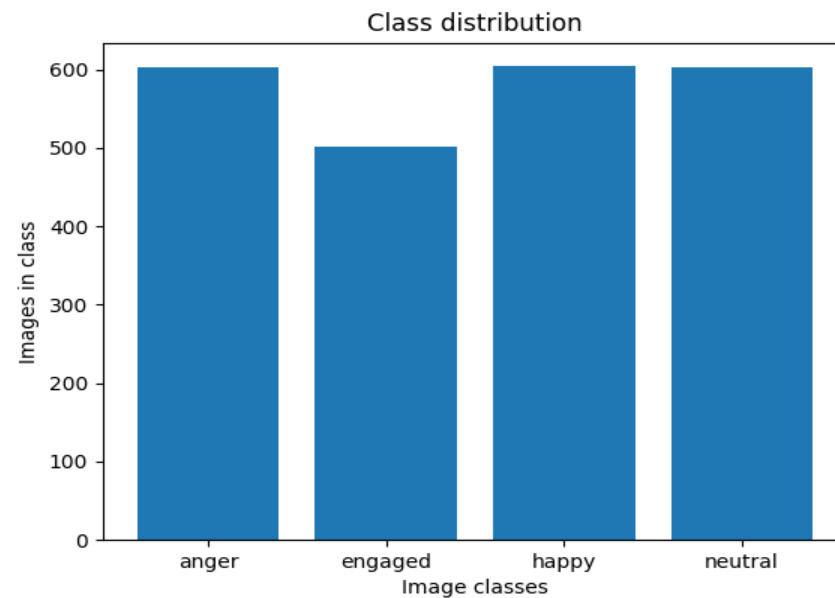
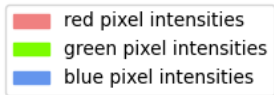


Figure 2 Class distribution.

- 2. Pixel Intensity Distribution



Aggregate RGB Pixel Intensities for each Class:

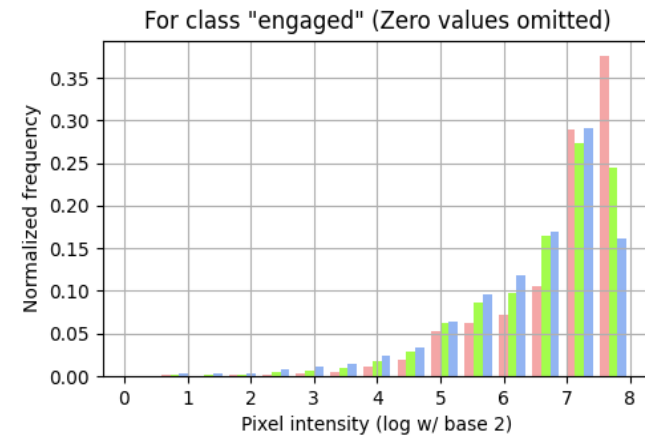
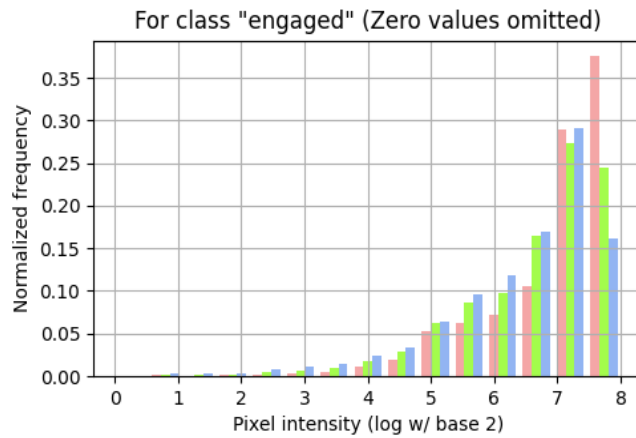
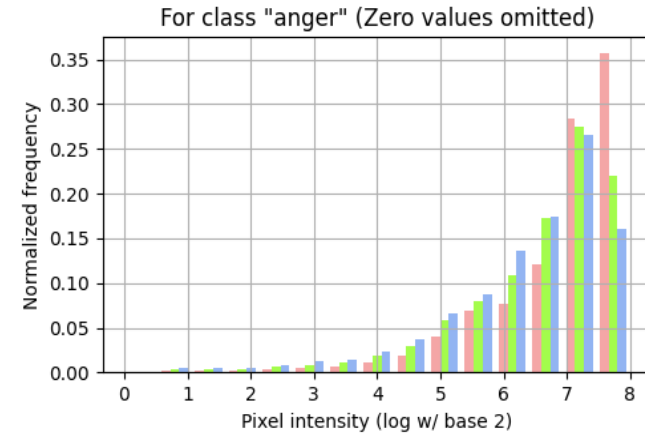
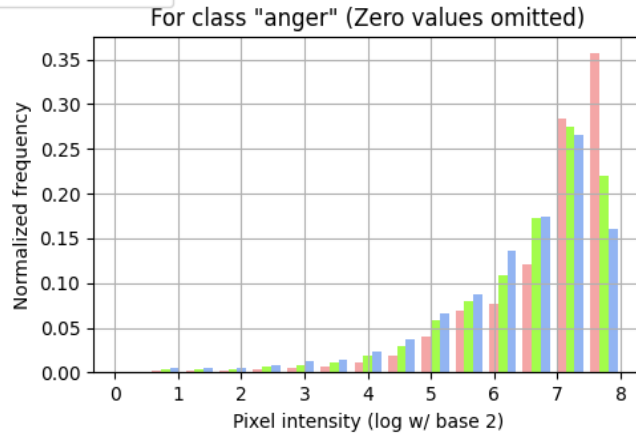
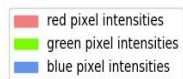


Figure 3 Pixel Intensity Distribution

- 3. Sample Image



15 sampled images from the class "engaged"

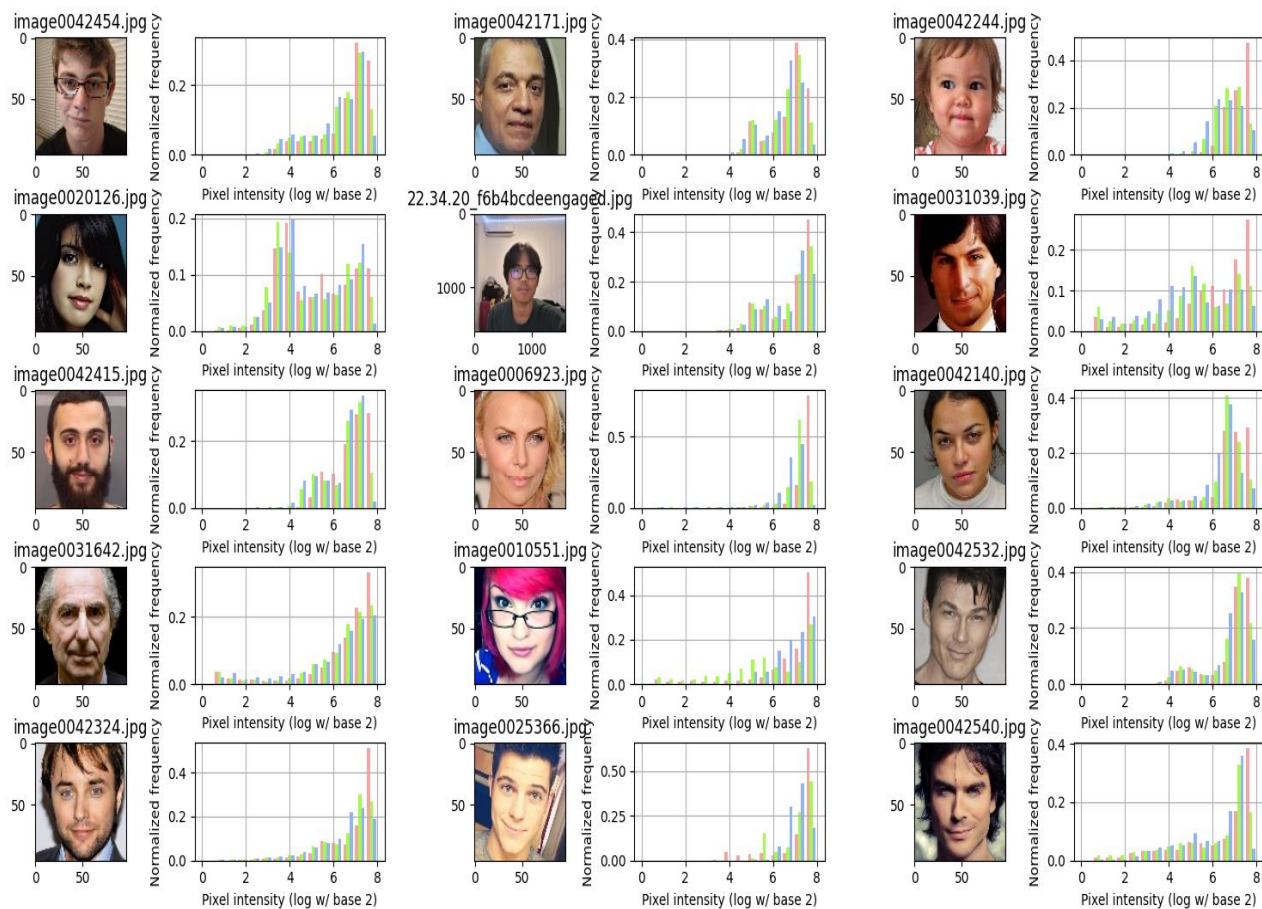
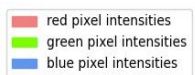


Figure 4 Sample Images Engaged.



15 sampled images from the class "happy"

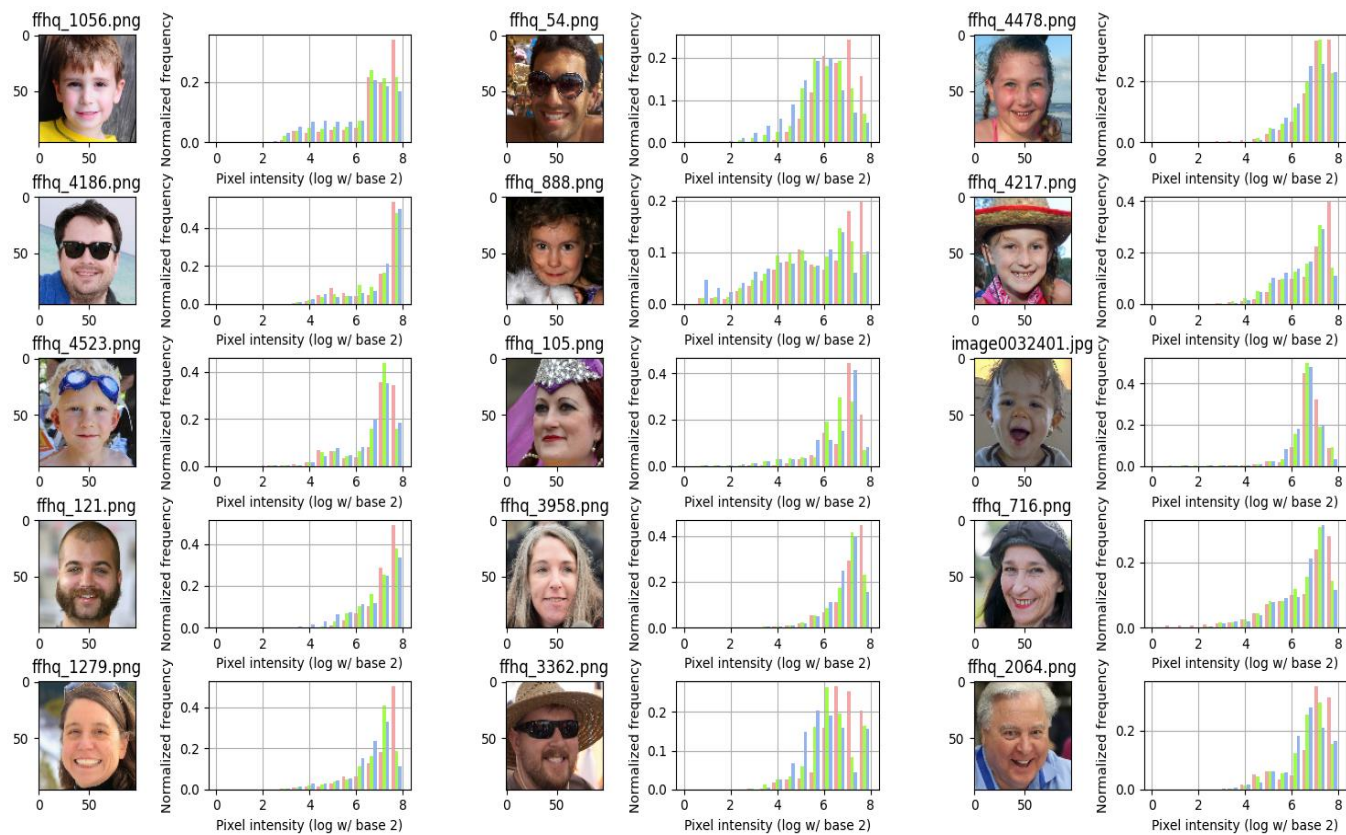
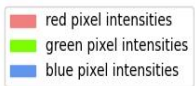


Figure 5 Sample Images Happy



15 sampled images from the class "neutral"

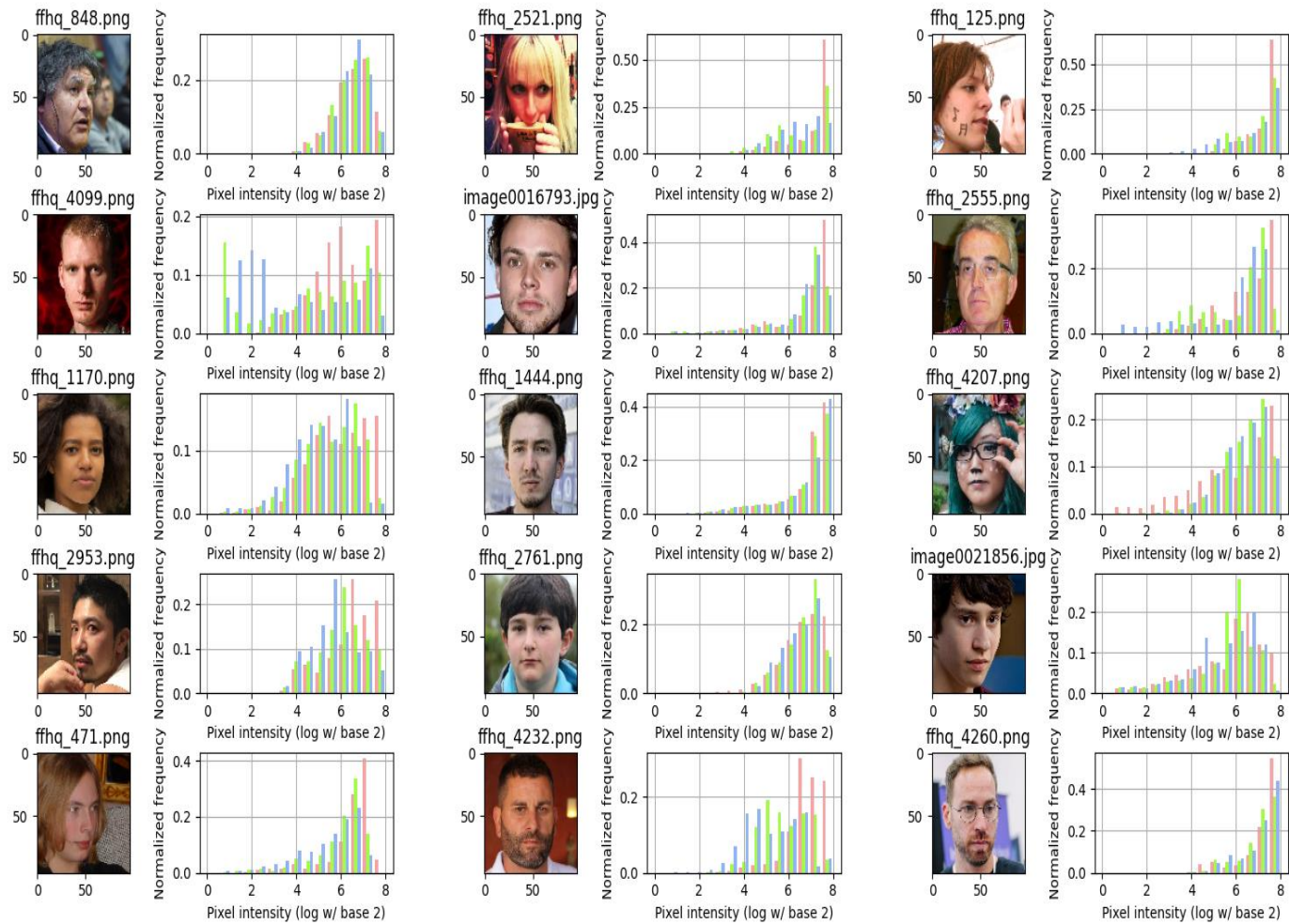
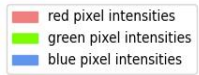


Figure 6 Sample Images Neutral



15 sampled images from the class "anger"

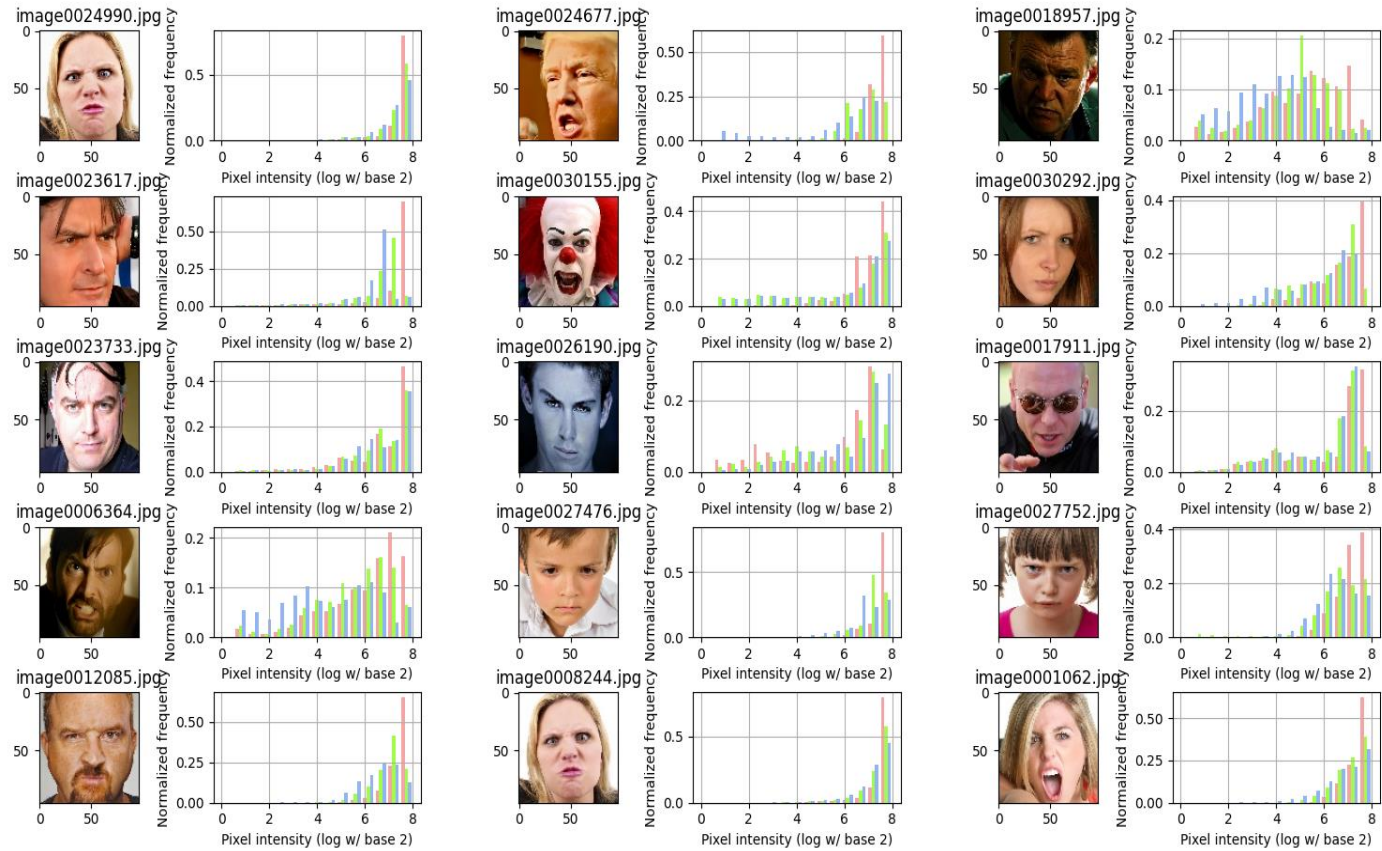


Figure 7 Sample Images Anger

Report: CNN Architecture and Evaluation

Main Model Overview:

To start off, we were inspired by the model used in Lab07 and in a crash course posted on Moodle [4]. These models were used for well-known datasets such as CIFAR-10 (64 x 64) and MNIST (28 x 28). CIFAR-10 is RGB, while MNIST is grayscale. These models achieved high accuracy for training and validation with their respective datasets. This was expected because their datasets were significantly larger than ours, reducing the likelihood of overfitting. Running their model architecture with our dataset would likely result in much higher overfitting, causing the early stopping method to trigger much earlier. Our model is also designed for grayscale images, meaning the input to the model has only 1 channel (not 3, as there is no RGB), and it outputs 4 channels because there are 4 classes for each emotion.

- **List of layers with their types, sizes, and activation functions.**

We decided to test different models, and the best design we obtained included adding 4 convolutional layers with a kernel size of 3x3 and padding size of 1. We chose to use MaxPooling layers because they performed better than AveragePooling layers in our tests. We also utilized Batch Normalization after each convolutional layer to normalize the output.

In the first convolutional layer, the input size is 1 (grayscale image), and the output size is 32. This layer also includes a Batch Normalization layer to match the output size of the convolutional layer. We applied an activation function called LeakyReLU with a negative slope of 0.1.

The second convolutional layer takes an input size of 32 and produces an output size of 64. Similarly, the third convolutional layer has an input size of 64 and an output size of 128. The fourth and final convolutional layer has an input size of 128 and produces an output size of 256. Each of these convolutional layers is followed by a Batch Normalization layer, which takes the output size of its respective convolutional layer.

After the convolutional and pooling layers, we used fully connected layers. The first fully connected layer (fc1) has an input size of $256 * 5 * 5$ and an output size of 256. The second fully connected layer (fc2) has an input size of 256 and an output size of 4, which corresponds to the total number of classes.

- **Discussion of any design nuances:**

One of the design nuances we used in our main model is batch normalization. We applied batch normalization after each convolutional layer, resulting in a total of four batch normalization layers. This helps to stabilize and accelerate the training process by normalizing the inputs to each convolutional layer.

We also added a dropout layer with a dropout rate of 0.5. This randomly sets a fraction of the input units to 0 during each epoch of the training process, which helps to prevent overfitting.

Additionally, we used the Leaky ReLU activation function to prevent the issue of neurons becoming inactive and only outputting 0. The negative slope of the Leaky ReLU ensures that a small gradient is maintained for negative inputs, which helps in training deeper networks effectively.

This following picture represents our Main Model:

```
OB_05Model(  
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
  (relu1): LeakyReLU(negative_slope=0.1)  
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
  (relu2): LeakyReLU(negative_slope=0.1)  
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
  (relu3): LeakyReLU(negative_slope=0.1)  
  (maxpool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (bn4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
  (relu4): LeakyReLU(negative_slope=0.1)  
  (maxpool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (fc1): Linear(in_features=6400, out_features=256, bias=True)  
  (fc2): Linear(in_features=256, out_features=4, bias=True)  
  (dropout): Dropout(p=0.5, inplace=False)  
)
```

Figure 8 Main Model design

Variant 1:

In Variant 1, we decided to test the model with fewer convolutional layers—specifically, only 2—while keeping all other hyperparameters the same as the main model. This includes maintaining the same learning rate, weight decay, optimizer (Adam), learning rate scheduler, and activation functions. The goal is to evaluate the model's performance with reduced complexity while ensuring the same regularization techniques are applied.

```
class Model_Variant1(
    (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu1): LeakyReLU(negative_slope=0.1)
    (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu2): LeakyReLU(negative_slope=0.1)
    (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (fc1): Linear(in_features=3072, out_features=256, bias=True)
    (fc2): Linear(in_features=256, out_features=4, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
```

Figure 9 Variant 1 Model

Initially, we aimed to test how different numbers of layers impact model performance. We wanted to see if fewer layers would lead to quicker overfitting or if more layers would improve accuracy. Our curiosity was to determine if having more layers inherently means a stronger model. Through experimentation, we found that adding more layers made the model too complex for our dataset of 2000 images, leading to overfitting.

Variant 2:

In Variant 2, we decided to keep the same number of convolutional layers but change the kernel size from 3x3 to 7x7. Our goal is to test the time complexity of each epoch, overall performance, and accuracy of the model. By increasing the kernel size, we aim to observe how it affects the model's ability to capture features, computational requirements, and ultimately its effectiveness in handling the dataset.

```
08_05Model_Variant2(  
  (conv1): Conv2d(1, 32, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))  
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
  (relu1): LeakyReLU(negative_slope=0.1)  
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (conv2): Conv2d(32, 64, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))  
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
  (relu2): LeakyReLU(negative_slope=0.1)  
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (conv3): Conv2d(64, 128, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))  
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
  (relu3): LeakyReLU(negative_slope=0.1)  
  (maxpool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (conv4): Conv2d(128, 256, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))  
  (bn4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
  (relu4): LeakyReLU(negative_slope=0.1)  
  (maxpool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  (fc1): Linear(in_features=6400, out_features=256, bias=True)  
  (fc2): Linear(in_features=256, out_features=4, bias=True)  
  (dropout): Dropout(p=0.5, inplace=False)
```

Figure 10 Variant 2 Model

In Variant 2, our primary motivation for this change is to test the time complexity of each epoch. By using a larger kernel size of 7x7 instead of 3x3, we anticipate that each epoch will take longer due to the increased computational load. We are curious to see if this longer processing time results in better model performance and accuracy. The goal is to determine whether a larger filter size, despite increasing computational time, leads to improvements in the model's ability to capture features and overall performance.

Training Process

- 1. Training Methodology:

In the training process, we decided to use a maximum of 100 epochs because, during the test phase, we were unsure when the early stopping method would be triggered. Many of the models nearly reached the maximum number of epochs before early stopping was activated.

The initial learning rate we decided to go with is 0.0001 because, after several tests with models, we found that a smaller learning rate results in more precise convergence. However, the downside is that it takes more time during the training phase, which is still beneficial for achieving potentially good results and a robust model. We also decided to implement a scheduler (ReduceLROnPlateau). This scheduler in PyTorch updates and reduces the learning rate when the validation loss hasn't improved for several epochs. It helps tune the training process and adjust the learning rate for improved convergence.

We are also using a Cross-Entropy function to help the model minimize the difference between the predicted class probabilities and the actual class labels. This will yield better results for the classification task and minimize the loss. Additionally, we added weight decay in the optimizer to prevent overfitting by adding a penalty to the loss function for large weights. This encourages the model to learn smaller weights, leading to a simpler model that generalizes better to unseen data.

- 2. Optimization Techniques:

We decided to test different optimizer algorithms in the code to see which one could result in better accuracy. For this, we chose to go with the Adam optimizer. We observed that Adam was the most effective in minimizing the loss during the training phase.

We decided to implement an early stopping method to prevent overfitting. This method compares the validation loss value to the best loss value observed. If the current validation loss is higher, it increases a counter. If this counter surpasses the specified patience, early stopping is triggered. Our patience was set to 5 epochs. This means if the validation loss does not improve for 5 consecutive epochs, training will stop early to prevent overfitting and save computational resources.[5]

We also added a timer function during the training phase to measure the execution time of each epoch. This allows us to compare the performance of different models in terms of training speed

Performance Metrics

- **Evaluation**

Table 6 Macro vs Micro:

Performance metrics of the model on the test set

	macro			micro			accuracy
	precision	recall	f1-score	precision	recall	f1-score	
Main Model	0.8835	0.8802	0.8807	0.8851	0.8851	0.8851	0.8851
Variant 1	0.7952	0.7838	0.7802	0.7960	0.7960	0.7960	0.7960
Variant 2	0.8219	0.8199	0.8182	0.8333	0.8333	0.8333	0.8333

Above are the overall performance metrics for each model on the test set. Before we analyze the metrics of each model for each class, we must identify which performance metrics are important given the use-case of the model.

Suppose we use the model to see which students are engaged vs. those who are not. In this case, it would be better for the model to falsely classify non-engaged students as engaged than it would be to falsely classify engaged students as non-engaged. A possible cost of falsely classifying engaged students as non-engaged would be that it discourages students from paying attention because the model would identify that they were not engaged in the first place. In other words, false negatives are more costly. This means that we should focus more on the ‘recall’ metrics of our models.

Before we compare the overall performances of the models against each other, we will first examine the individual performances for each class for each model. We will examine the confusion matrices for each model on the testing set and the performance metrics per class.

Main Model

Table 7 Confusion Matrix for Main Model

Confusion Matrix Table

	anger	engaged	happy	neutral
anger	73	8	1	0
engaged	13	60	0	4
happy	0	2	84	6
neutral	4	1	1	91

Above is the generated confusion matrix for the main model on the testing set. Suppose for a class 'j', the model outputs a prediction 'i'. The index [i, j] in the array then gets incremented. The true positives (TP) for each class are obtained by taking the diagonal of the matrix. The false positives (FP) are obtained by taking the sum of each element in a column, minus the diagonal element. The false negatives (FN) are obtained by taking the sum of each element in a row, minus the diagonal element. The true negatives (TN) are obtained by subtracting the sum of the TPs, FPs, and FNs from the sum of all values in the matrix.

Since the columns represent the actual classes, we can see that the model confuses 'anger' and 'engaged' frequently. The model also occasionally confuses 'neutral' expressions as 'happy'. These misclassifications could be because these emotions invoke some similar (or similar enough) expressions between the confused classes.

From there, we can then calculate the performance metrics of the model per emotion class. We can see from below that the model had trouble identifying 'engaged' images, with the model being only able to identify around 77.9% of the true positive cases – the worst out of all classes.

Table 8 Main Model Performance Metrics:

Performance metrics per class on the test set

	anger	engaged	happy	neutral
precision	0.81111	0.89024	0.84884	0.92529
recall	0.84507	0.77922	0.81081	0.91954
f1_score	0.97674	0.91304	0.94382	0.97126
accuracy	0.90099	0.93814	0.91919	0.95402

Variant 1

Table 9 Confusion Matrix Variant 1

Confusion Matrix Table

	anger	engaged	happy	neutral
anger	70	13	1	1
engaged	35	38	2	2
happy	0	0	84	11
neutral	2	0	4	85

Above is the confusion matrix for variant 1 on the testing set. We can see that it misclassifies 'anger' as 'engaged' 35 times, almost triple that of the first model. Out of 77 outputs that were predicted as 'engaged', approximately 49% of them were true positives ('engaged'). It looks like this model struggles the most with identifying 'engaged' images, mostly confusing it with 'anger' images. We can also see that it somewhat confuses both 'happy' and 'neutral' images. There were 11 instances where the model predicted 'happy', but the image was 'neutral'.

Calculating the performance metrics per class, we can see that this variant struggles with identifying 'engaged' images. The 'recall' values are noticeably lower than the main model, with the 'recall' for the 'engaged' class being at an approximate 49.4% - compared to 77.9% for the main model. It also struggles with identifying 'happy' images, with a 'recall' value of approximately 59.4% compared to 81.1% for the main model.

Table 10 Variant 1 Performance Metrics:

Performance metrics per class on the test set

	anger	engaged	happy	neutral
precision	0.65421	0.82353	0.72917	0.85057
recall	0.74510	0.49351	0.59375	0.85057
f1_score	0.92308	0.88421	0.90323	0.94828
accuracy	0.85859	0.93407	0.89474	0.94253

Variant 2

Table 11 Confusion Matrix Table Variant 2 and the Performance Metrics:

Confusion Matrix Table					Performance metrics per class on the test set				
	anger	engaged	happy	neutral		anger	engaged	happy	neutral
anger	62	12	0	0	precision	0.72941	0.83784	0.77987	0.89943
engaged	22	46	1	4	recall	0.75410	0.63014	0.68657	0.87931
happy	1	0	92	10	f1_score	0.93878	0.89320	0.91542	0.95115
neutral	0	3	5	90	accuracy	0.86538	0.91837	0.89109	0.93678

Like the previous models, the same pattern of ‘anger’/‘engaged’ and ‘happy’/‘neutral’ confusion persists. We can see that ‘anger’ specifically was being more misclassified as ‘engaged’ compared to the main model, although not as bad as variant 1. Like variant 1, this variant confuses ‘happy’ images as ‘neutral’ images, with 10 ‘neutral’ images being incorrectly classified as ‘happy’.

This is also reflected in the performance metrics, with each metric for each class being lower than the main model but being marginally better than the first variant. For this model, the ‘recall’ for the ‘engaged’ class is approximately 63%.

- **Common Patterns**

We can see that each model roughly struggles with the same classes. For all models, ‘anger’ and ‘engaged’ as well as ‘happy’ and ‘neutral’ were frequently confused for each other.

From the accuracies of each class, we can see that, for all models, ‘neutral’ had the highest accuracy. This implies that it was the most well-recognized class – followed by ‘engaged’. Looking back at the confusion matrices, ‘anger’ and ‘engaged’ were barely confused for ‘happy’ and ‘neutral’, and vice versa. You could say that these pairs form two ‘groups’, and classifications can fall into one of these two ‘groups’. Perhaps ‘engaged’ and ‘neutral’ contained more recognizable features compared to the other class in their respective pairs, which explains they were the most recognized.

Images in one class in the dataset may have features prevalent in another. For example, some angry expressions may look neutral. This explains the confusion between the two classes and why one class is more recognized than the other.

- **Overall Comparison**

Table 12 Overall Comparison:

Performance metrics of the model on the test set

	macro			micro			accuracy
	precision	recall	f1-score	precision	recall	f1-score	
Main Model	0.8835	0.8802	0.8807	0.8851	0.8851	0.8851	0.8851
Variant 1	0.7952	0.7838	0.7802	0.7960	0.7960	0.7960	0.7960
Variant 2	0.8219	0.8199	0.8182	0.8333	0.8333	0.8333	0.8333

Overall, variant 1 performs the worst across all metrics amongst the models. Since all models were trained and tested the same way, it is most likely that the architecture of this model is inferior to the other two models for emotion classification.

Comparing the main model with variant 2, we can see that the main model has better micro and macro metrics. The main model having higher macro metrics suggests that it is better at correctly predicting each class on average when compared to variant 2. In other words, the main model seems to be more balanced across the classes. The main model having higher micro metrics implies that it is more effective at making correct classifications across all classes combined.

From the above data, we can conclude that the main model performs the best out of all other models.

Impact of Architectural Variations

- **Depth of Model:**

Our conclusion is that for the number of images we have, 2 layers are not enough, and 7 layers are too complex for this dataset, leading to quick overfitting. The model with 2 convolutional layers overfits after only 26 epochs, whereas the model with 4 layers can go up to 66 epochs before overfitting.

- **Kernel Size Variations:**

Our main model is using 3 kernel size. The variant 2 is using kernel size 7 and it take a lot of time to compute each epoch around 62 second compared to 32 seconds for a normal kernel size 3.

Conclusions

To conclude, the best model we found has a test accuracy of 89% using 4 convolutional layers. In the evaluation phase we see that there is a nice difference between our main model and variant in term of confusion matrix and macro micro average.

Finally, for testing purposes, we chose to load the saved model and test it with random images. Out of 6 images, we identified 4 correctly, while 2 images were not identified correctly, which is consistent with the test accuracy of 89%. To improve our model, we will need to add more image to our dataset, varies of image of race, sex and other to adjust the model to perform a better performance overall.

References

- [1] AffectNet: A database for facial expression ... - Mohammad Mahoormohammadmahoor.com/wp-content/uploads/2017/08/affectnet_onecolumn-2.PDF, http://mohammadmahoor.com/wp-content/uploads/2017/08/AffectNet_oneColumn-2.pdf (accessed May 31, 2024).
- [2] Mohammad H. Mahoor, "AffectNet" mohammadmahoor.com/affectnet/. <https://web.archive.org/web/20240205175353/http://mohammadmahoor.com/affectnet/> (accessed May. 29, 2024).
- [3] (PDF) optimizing the color-to-grayscale conversion for Image Classification, https://www.researchgate.net/publication/282730506_Optimizing_the_Color-to-Grayscale_Conversion_for_Image_Classification (accessed May 31, 2024).
- [4] Anandsaha, "Deep-learning-with-pytorch-video/3.cnn.for.digits.classification.ipynb at master · anandsaha/deep-learning-with-pytorch-video," GitHub, <https://github.com/anandsaha/Deep-learning-with-PyTorch-video/blob/master/3.cnn.for.digits.classification.ipynb> (accessed Jun. 13, 2024).
- [5] ChatGPT prompt for how to implement EarlyStop method

how to implement an early stop method with a patience of 5 in CNN with PyTorch

To implement an early stop method with a patience of 5 in a CNN with PyTorch, follow these steps:

Step 3: Integrate Early Stopping in the Training Loop

```
python
num_epochs = 100

for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0
    for data, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)
```

```
train_loss = train_loss / len(train_loader.dataset)

model.eval()
val_loss = 0.0
with torch.no_grad():
    for data, targets in val_loader:
        outputs = model(data)
        loss = criterion(outputs, targets)
        val_loss += loss.item() * data.size(0)

val_loss = val_loss / len(val_loader.dataset)

print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}')

scheduler.step(val_loss)
early_stopping(val_loss, model)

if early_stopping.early_stop:
    print("Early stopping")
    break
```