Student Name 1: Quan Hoang Dinh
Student Name 2: Elijah Gray

Project 3 Documentation

I. Files Explanation:

- **code_1.py**: the python codes that used to train the model
- **best_trained_model.pth**: the copy of the best trained model
- **epoch_accs.json**: the stored accuracies over epoch for the best trained model
- **Recalls and Precisions.txt**: stored the output of the precisions and recalls of the 7 expressions
- **Confusion.png**: the image for Confusion matrix of best model
- **Perfomance.png**: the image for Performance graph of best model
- **Misclass.png**: the image for the 3 misclassified images of best model

II. Contributions of Members

Quan Dinh:

- Implemented the transformations, scheduler, ANN architectures, and hyperparameters for the best model
- Debug and tested the code
- Experimented on the weights of models to obtain the best model and minimize overfitting
- Wrote the report
- Researched the practices for ANN, CNNs and FER

Elijah Gray:

- Implemented the evaluations code: performance metrics, confusion matrix, precisions and recalls, overall accuracies, and misclassified images.
- Debug and tested the code
- Experimented on the weights of models to obtain the best model and minimize overfitting
- Wrote the report
- Researched the practices for ANN, CNNs and FER

III. Description of Experiments:

DATA PREPROCESSING

1. Image dimensions
    - We tried out the default dimension of 128x128, which gave us a benchmark for how the dimension affects the training and validation. We found that larger dimensions can lead to longer training time.
    - We learned that normal FER training typically uses images 224x244. We tried it out and it took too much time to train with no noticing increase in accuracy.
    - We reduced the size to 64x64, and found that it reduces the training time significantly without a reduction in accuracy.
    - We looked through the images that are being trained, and found that they are all (or at least mostly) 48x48. We believed that increasing the dimension beyond that is not worth it for the training, and keeping the dimension as they come in will be balanced for the training.
    - The 48x48 dimension also keeps the spatial dimensions of convolution channels small, so we get less neurons per layer, which means faster training.
2. Image transformations
    - We experimented with polarization first because we thought it would help enhance deep shadows of images, which allow the convolution to get features better.
    - However, we found that the purpose of transformation is to make the model resilient and have more diverse data for it to train on. So we used Rotation, Affine, and ResizedCrop to test since these transformations can imitate the turning of the head or the zooming in of faces.
    - The Rotation, Affine, and ResizedCrop changed the training data so much that it caused overfitting in the model, so we lowered the value in the transformation so that the training data is less diverse. It does increase our accuracy of training, but it is difficult to control because we do not truly know if a certain weight of the transformation really is affecting the training.
    - We then decide to go with adding 4 pixels of padding to the original image, then resizing it back to 48x48 with 50-100% crop, and apply a 50% chance of horizontal flip. This allows for more consistent and controlled transformation where we get more diverse images, but we also know that the image keeps crucial expressions for the model to train. This leads to better transformed images that will not stray far from the test images, while allowing the training model to generalize better.

- We also added Normalizations with mean and standard deviation of 0.5 to make sure all the images are approximately the same intensity, which helps with faster convergence and reduces bias.

ANN ARCHITECTURES

3. Convolution layers
    - We only use 2 layers of Convolution in the start, expanding the 1 channel (Greyscale) to 32, then to 64. This only allow the valid accuracy to ~40%
    - We then learn that more convolution layers can lead to increased accuracy for it will detect more edges and complexities. We tried 4 convolution layers to break the channels into 1-32-64-128-256 (1 does not count as a layer because it is the input). This increases the accuracy of the model to ~55-60% consistently regardless of other experiments. That leads us to believe that the 4-layer convolutions model is optimal.
4. Batch Normalization
    - For every convolutional layer, we added batch normalization after every conv2D in order to stabilize the layers of every batch. This allows for faster convergence, hence more efficient training.
    - We did not find any downsides for Batch Normalization so we use it in every iteration of experimentations.
5. Activation Function
    - We use activation after every batch normalization.
    - We did not experiment much with different activation functions aside from ReLU, since we have read sources that ReLU is the more consistent, efficient, and easy to use to train the model. We also believe ReLU is a good activation function for our model.
6. Max Pooling
    - We use max pooling after every activation function. This allows for reduction in spatial dimensions of input images, while keeping important learned features.
    - We believe that this makes the model more robust to slight changes, and also makes the model simpler by reducing the amount of neurons per level.
    - We always use Max Pooling of stride 2 and kernel_size 2, because it is easy to calculate with the set dimension. The spatial dimension always halves every convolution layer, leading to the final dimension of 3x3, which is small enough to flattened effectively and a standard size for final spatial dimension.

7. Linear layers
    - We used the default 1 layer of linear after flattened at first. With the formula of in-channels of the final convolution channels x spatial dimensions to 7 out channels.
    - We then used 3 layers of linear after flattened, from 256*3*3 (input) to 256, then 256 to 128, then 128 to 7. This allows the model to project the channels into more layers and learn the relationships in the data and generalize slower as it converges to the 7 expressions. We noticed a slightly stabler accuracy curve as we do this (a log curve).
    - Since 3 linear layers and 4 convolution layers were too deep for a complex system, we dropped the linear layers to 2 (256*3*3 - 256 - 7). This allows a simpler model with faster training and no noticeable accuracy drop.
    - We also normalize after the first linear layer, and add a ReLU activation layer before it converges to the final linear.
8. Dropout
    - We did not use drop out initially because we did not know what it did. But gradually we realize that the Dropout can help us reduce overfitting and drop neurons randomly, which can help with making the model “unlearn” erroneous neurons and generalize better.
    - In the 1st experiment, we dropped the neurons at 50% after flattening. It was not effective at closing the gap of overfitting.
    - In the 2nd experiment, we dropped the neurons after every convolution layer by 50%. It was effective at closing the gap of overfitting, but we wanted to increase the accuracy a bit more so we gradually tested with dropping neurons with less percentage.
    - In the 3rd experiment, we dropped the neurons after every other convolution layer by 50%, and the first linear layer by 50%. It was much more effective than the first initially, and then the gap between training and valid accuracy got larger post epoch 20 at a rate of ~0.08.
    - In the 4th experiment, we dropped the neurons after every convolution layer by 30-50% interlaced, and the first linear layer by 50%. This makes the performance graph of the two curves pretty much on top of each other, so we believe this was optimal for our model.

HYPER PARAMETER-VALUES

9. Batch sizes
    - We only control batch sizes to see how it changes the accuracy in respect to the performance and training time. We tested from 32-64-128-256-512.

  - We found that 128-256 works the best (we also used GPU to test). However, 256 often finds itself drastically slower during experimenting when we change the learning rate or model layers.
  - So we decided to go with 128 batch size, which has the most success with good performance and accuracy.

10. <u>Epochs</u>
    - We used epoch 8 at first to see how the values within the optimizer, layers, and transformation do. The short iteration allows us to see the early phases of short training so we can change the other values efficiently.
    - We increased the epoch to 15, and noticed that it is still learning. The program was running very fast post epoch 1, so we assumed we could use more iterations.
    - We increased the epoch to 30, and noticed that the learning hit a plateau. We believe that 30 is a good point to stop training to stop the model from over-learning to memorizing. The plateau is proof that the accuracy will not increase much further.
11. <u>Optimizer learning rate and weight decay</u>
    - We experimented with various learning rates of 0.001, 0.0003, 0.00025, 0.00015, 0.0001. We changed it depending on how fast we want the early phases of learning to accelerate.
    - We used a weight decay of 1e-4 and 1e-5. Since we did not notice much performance or accuracy changes with the performance decay, we keep the decay at 1e-4. This does not mean the weight decay did not help in contrast to if it was to 0.
    - We only tried out Adam and AdamW. Since Adam is just much easier to use and is recommended by various sources, we used Adam throughout these experiments.
    - Since we iterated to 30 epochs, we believe that a scheduler can help us make the learning rate more dynamic.
12. <u>Scheduler</u>
    - We implemented a step scheduler as first: with every 10 epochs, the learning rate multiplied by 0.1. This allows the mid and late training phase to be more closely aligned with the true accuracy value while accelerating the early phases like we wanted. We also changed the scheduler to have: 5 epochs, multiplied by 0.5, but that did not have much impact.
    - We found that a reduceLRonPlateau can be more dynamic at adjusting weight when the loss value has not changed much past a specific amount of epochs. We ended up using this scheduler instead because it was easier to control with all the previous experiments and make a slightly better performance curve.

    - The reduceLRonPlateau reduces the learning rate at every local minima by 0.5, for every 3 epochs that the valid loss does not change.

EXPERIMENTED ACCURACIES RESULTS

- <u>Too low learning rates, low convolution layers, single linear layer</u>
    + They have low initial accuracies of **~24-28%** in the first 5 epochs, which we deemed too nonoptimal for the model and we exit the program early to drop these experiments.
    + We predicted that it will not learn fast enough, meaning an inefficient model.
- <u>Too large batch numbers, too many layers, large preprocessed image dimensions, high neurons per layers (a.k.a too little max pooling or high channels/filters)</u>
    + They have extremely slow training time with little results and accuracies could have otherwise gotten from a more efficiently trained model (**~30-35%** in the first 5 epochs).
    + We think that it is not worth investing time into models like this.
- <u>4-Layers Convolutions, 2 Linear Layers, 256 maxed channels,</u>
    + These sets always perform well no matter what, with accuracies **~50-55%** consistently from the 4th epoch.
    + We think that this is the base and optimal structure for the model.
- <u>Too little/high learning rates, no scheduler</u>
    + These sets grow way to slow from epoch 10th and on, stay stagnant at **~50-55%**
    + We think that a scheduler can help it accelerate faster with a higher initial learning rate and control it as the model progress through epochs
- <u>No/Low Dropouts, No/Too much image transformations, uncontrolled learning rates, no weight decay</u>
    + These sets always perform well initially with guaranteed accuracies **~50-55%** consistently from the 8th epoch.
    + But the validation accuracies are always volatile and far off from the training as it progresses from mid-late phases. For example, the training accuracy would be **80%**, and the validation would be **50%** and plateau.
    + We suspect that the program is remembering the images rather than generalizing.
    + We think that these attributes are making the model overfit or the test images too drastically different from the training.
- <u>With all the experiments, we finetune our best model to have an accuracy of **~60%** consistently.</u>

IV. Description of Best Model / Best Training Procedure:

The best performance model uses:

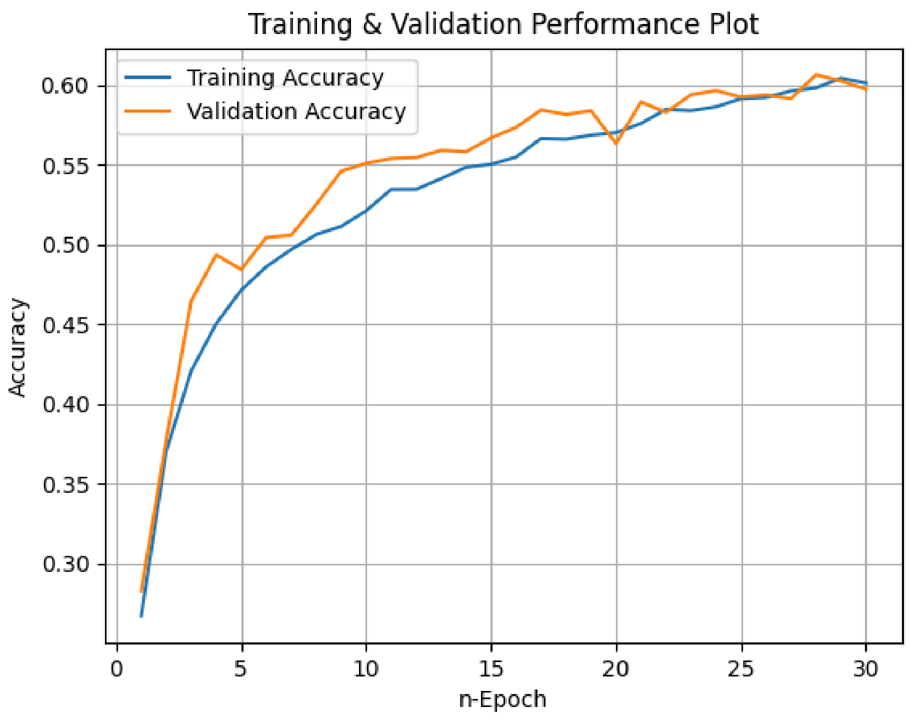
1. **Pre-Processing:**
    - Image dimension: 48x48
    - Image transformation:
        + RandomPad(padding=4, padding_mode='reflect')
        + RandomResizedCrop(size=(48,48), scale=(0.5, 1.0))
        + RandomHorizontalFlip()
        + Normalize(mean=[0.5], std=[0.5])
2. **Hyperparameters:**
    - Batch size: 128
    - N-Epochs: 30
    - Optimizer: Adam(model.parameters(), lr=0.001, weight_decay=1e-4)
    - Scheduler: ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=3)
3. **Model:**

    The model goes in the following sequence
    - <u>Convolution Layer 1:</u> (32 x 24 x 24 = **18432 neurons**)
        + Conv2D(1, 32, kernel_size=3, padding=1)
        + BatchNorm(32)
        + ReLU()
        + MaxPool2D(kernel_size=2, stride=2)
        + Dropout(p=0.5)
    - <u>Convolution Layer 2:</u> (64 x 12 x 12 = **9216 neurons**)
        + Conv2D(32, 64, kernel_size=3, padding=1)
        + BatchNorm(64)
        + ReLU()
        + MaxPool2D(kernel_size=2, stride=2)
        + Dropout(p=0.3)
    - <u>Convolution Layer 3:</u> (128 x 6 x 6 = **4608 neurons**)
        + Conv2D(64, 128, kernel_size=3, padding=1)
        + BatchNorm(128)
        + ReLU()
        + MaxPool2D(kernel_size=2, stride=2)
        + Dropout(p=0.5)
    - <u>Convolution Layer 4:</u> (256 x 3 x 3 = **2304 neurons**)
        + Conv2D(128, 256, kernel_size=3, padding=1)
        + BatchNorm(256)
        + ReLU()
        + MaxPool2D(kernel_size=2, stride=2)

        + Dropout(p=0.3)
- Flatten **(2304 neurons)**
- Linear Layer 1: **(256 neurons)**
    + Linear(256*3*3, 256)
    + BatchNorm(256)
    + ReLU()
    + Dropout(p=0.5)
- Linear Layer 2: **(7 neurons)**
    + Linear(256, 7)

4. **Training:**
    - Training goes through the generic steps described on Canvas
    - We modified the training procedure to save epoch performance:
        + The accuracies of training per epochs get saved in a list
        + The accuracies of validating per epochs get saved in a list
        + Changed the two saved list into a json data format
        + Save the formatted json data to a file epoch_accs.json
        + The file can used to graph performance later
    - We modified the training procedure to step the scheduler by passing the valid_loss into its parameter.

V. Training and Validation Plot:



VI. Performance of best model:

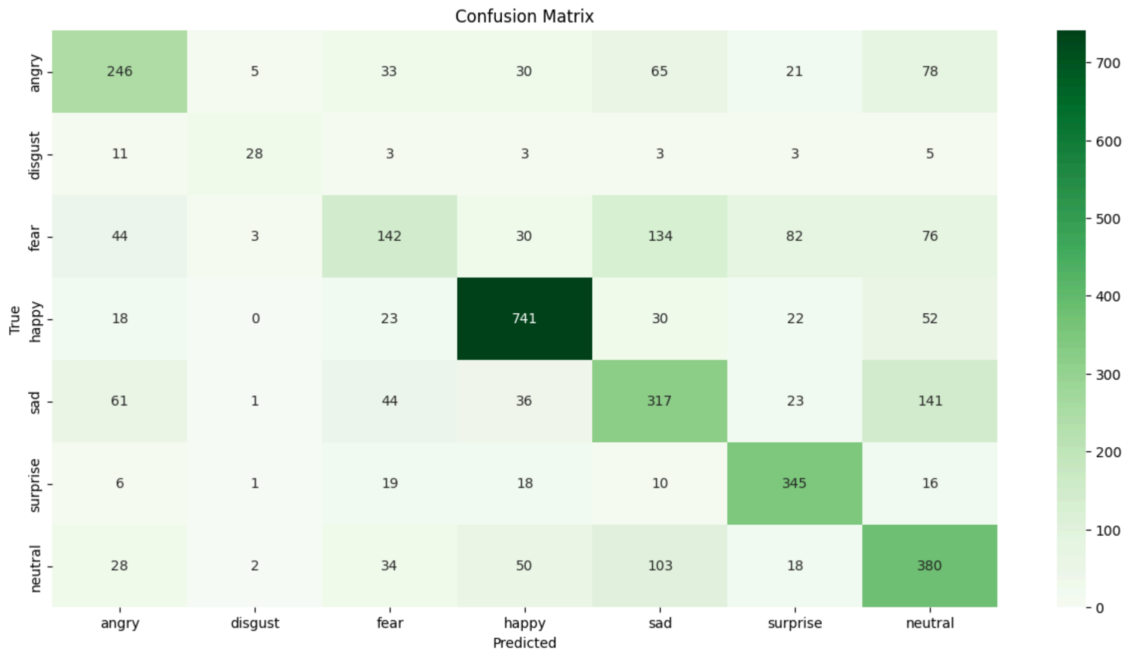
1. Performance overview

For the ease of reading, we will shortened the numbers to 5 significant figures

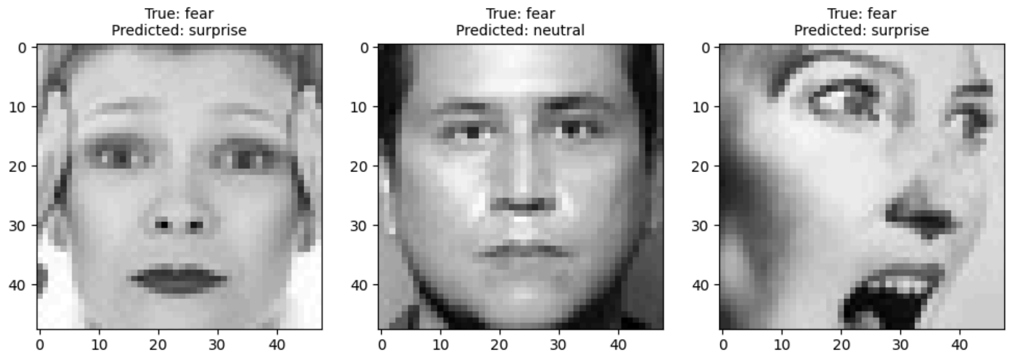
- **Overall accuracy**: 61.356 -%

- **angry**:	precision (59.420-%),	recall (51.464-%)
- **disgust**:	precision (70.0-%),	recall (50.0-%)
- **fear**:	precision (47.651-%),	recall (27.788-%)
- **happy**:	precision (81.607-%),	recall (83.634-%)
- **sad**:	precision (47.885-%),	recall (50.882-%)
- **surprise**:	precision (67.120-%),	recall (83.132-%)
- **neutral**:	precision (50.802-%),	recall (61.788-%)

2. Confusion matrix



3. Visualization of misclassified images.



- We think these images are misclassified because they resemble the expression of other expressions, and the program mistook it.
- The program also fails to have higher precision/accuracy in predicting certain expressions such as fear and disgust because there is much less data available for it to train on. Meaning it will generalize these lesser-trained expressions toward a more well-trained expression.