Classification of Human Facial Expressions

Santiago Becerra Cordoba
sab1475@g.harvard.edu

Sarah Kim
gak604@g.harvard.edu

Chloe Seo
yes593@g.harvard.edu

Kaylee Vo
kav418@g.harvard.edu

Jie Zhao
jiz273@g.harvard.edu

Abstract

FER-2013 is a dataset of facial expressions that contains 35,887 grayscale images of faces with seven different emotions: anger, disgust, fear, happiness, sadness, surprise, and neutral. In this project, we explore various different approaches to classifying emotions, custom convolutional neural networks (CNNs), transfer learning and vision transformers. Prior work on this dataset without an auxiliary dataset ranges from 70-75% accuracy, where human classification accuracy is benchmarked between 60 and 70%. Most relied heavily on CNNs but in this project we also tested vision transformers. We use an autoencoder to filter out outlying images based on reconstruction data. Using this filtered dataset, our custom U-Net model with squeeze-excite blocks achieves a test accuracy of 64%, matching human benchmarks. Our most performant model is transfer learning with a ResNet50 model, which achieves a test accuracy of 68.5%, comparable to other papers. Other papers have found higher accuracies in the range of 73-78% but these were done with an auxiliary dataset or with a multi-label setup. Although novel, the vision transformer did not perform well, only marginally better than random. This is most likely due to the small sample size, since models with weaker inductive biases need more data to learn.

Table of Contents

1. Introduction

This project focuses on developing a deep learning model to automatically classify human facial expressions using the FER-2013 dataset. Accurate emotion recognition has broad applications in human-computer interaction, mental health monitoring, and affective computing. The goal is to build a robust model that performs well despite challenges such as image artifacts, class imbalance, and subtle emotional variations.

1.1 Problem Statement

Originally, the objective was to train a CNN to recognize seven facial-expression classes from 48×48 grayscale images. However, exploratory data analysis revealed two key challenges:

1. Image Artifacts Approximately 15% of samples contain watermarks, compression artifacts, or uneven noise.

2. Class Imbalance Less frequent emotions such as “disgust” and “fear” make up less than 5% of the dataset, while “happy” and “neutral” are heavily overrepresented.

Therefore, our refined objective is to *design and evaluate a CNN-based emotion recognition system that is robust to image artifacts and class imbalance by incorporating targeted denoising, outlier detection, and data augmentation strategies*.

1.2 Significance of EDA Insights

The exploratory data analysis (EDA) fundamentally reshaped our approach, highlighting specific data quality and distribution issues that required targeted solutions. As mentioned earlier, EDA revealed significant image artifacts and class imbalance, which prompted the following methodological shifts:

Non-local means denoising was introduced to effectively reduce artifacts while preserving critical facial features, directly addressing the noise patterns uncovered during EDA.

Outlier detection using an autoencoder was implemented to systematically identify and remove approximately 2% of the noisiest or least informative images, reducing variance and improving overall data quality.

Class weighting and targeted augmentation strategies were incorporated to mitigate the severe class imbalance, ensuring the model could learn from under-represented emotions as effectively as from the majority classes.

By directly linking each preprocessing step to specific EDA findings, our workflow became both transparent and reproducible.

2. Comprehensive EDA Review

2.1 Data Description

To prepare the FER-2013 dataset for modeling, we followed a structured process involving data retrieval, loading, and initial exploration to ensure quality and usability.

Access: The FER-2013 dataset was retrieved via KaggleHub and stored in a structured local directory to accommodate its size, which exceeded typical repository limits.

Load: Images were loaded using Keras's `image_dataset_from_directory` and `ImageDataGenerator`, applying an 80/20 train-validation split to support model development and tuning.

Understand: Random subsets of images were sampled to generate pixel-value histograms and class-frequency charts. This early exploration helped identify image artifacts, corrupted samples, and significant class imbalance prior to modeling.

2.1.1 Access

In this section, we import libraries and write code to automatically pull the data from the internet. The code fetches it from Kaggle and stores it in the user's cache, followed by code that moves the data from the cache to the project directory.

In [1]:
```
# We keep this for Kaylee because she uses custom GPU
# !chmod +x ./scripts/installation.sh
# !./scripts/installation.sh
```

In [ ]:

```
import os
import pickle
import platform
import random
import shutil
import subprocess
import warnings
from pprint import pprint

import cv2
import kagglehub
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import tensorflow as tf
from IPython.display import Image, display
from keras.utils import set_random_seed
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.utils.class_weight import compute_class_weight
from tensorflow.keras import layers, models
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, R
from tensorflow.keras.layers import (
    Activation,
    Add,
    BatchNormalization,
    Concatenate,
    Conv2D,
    Dense,
    Dropout,
    Flatten,
    GlobalAveragePooling2D,
    GlobalMaxPooling2D,
    Input,
    MaxPooling2D,
    Multiply,
    Reshape,
    SeparableConv2D,
    SpatialDropout2D,
    UpSampling2D,
)
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.optimizers import Adam, AdamW
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.regularizers import l2
from tensorflow.keras.utils import image_dataset_from_directory
from tf_keras_vis.gradcam import Gradcam
from tf_keras_vis.utils.model_modifiers import ReplaceToLinear
from tf_keras_vis.utils.scores import CategoricalScore
from tqdm import tqdm
```

```
2025-05-07 21:02:28.348293: E external/local_xla/xla/stream_executor/cu
da/cuda_dnn.cc:9261] Unable to register cuDNN factory: Attempting to re
gister factory for plugin cuDNN when one has already been registered
2025-05-07 21:02:28.348344: E external/local_xla/xla/stream_executor/cu
da/cuda_fft.cc:607] Unable to register cuFFT factory: Attempting to reg
ister factory for plugin cuFFT when one has already been registered
2025-05-07 21:02:28.349599: E external/local_xla/xla/stream_executor/cu
da/cuda_blas.cc:1515] Unable to register cuBLAS factory: Attempting to
register factory for plugin cuBLAS when one has already been registered
2025-05-07 21:02:28.356582: I tensorflow/core/platform/cpu_feature_guar
d.cc:182] This TensorFlow binary is optimized to use available CPU inst
ructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, re
build TensorFlow with the appropriate compiler flags.
2025-05-07 21:02:29.173256: W tensorflow/compiler/tf2tensorrt/utils/py_
utils.cc:38] TF-TRT Warning: Could not find TensorRT
```

In [3]:
```
# Run Scripts
%run utils/utils.ipynb
```

We use gdown to pull down our drive of model weights and history because Github restricts the size of files.

In [4]:
```
url = "https://drive.google.com/drive/folders/1ev8zTryXIHTqj1rcjYqtdFp_D
target_dir = "models"

subprocess.run(["gdown", "--folder", url, "-O", target_dir])

url = "https://drive.google.com/drive/folders/1wKpMuVPeMvwsZmKHv-n-JMYO5
target_dir = "model_history"

subprocess.run(["gdown", "--folder", url, "-O", target_dir])
```

```
Retrieving folder contents
Retrieving folder contents completed
Building directory structure
Building directory structure completed
Downloading...
From: https://drive.google.com/uc?id=1QgQrWPLhzs1XNfzJy6ZjIzN9nhi_QcGe
To: /models/autoencoder.weights.h5
 54%|█████     | 524k/970k [00:00<00:00, 3.06MB/s]
Processing file 1QgQrWPLhzs1XNfzJy6ZjIzN9nhi_QcGe autoencoder.weights.h
5
Processing file 1DNjgOY97K-hwu0_L6DeZY2uDG8naa7aX enhanced_unet_model.k
eras
Processing file 1SEGFmjnRzfQBACobx0WteW5-hv3EIkCZ resnet50.weights.h5
Processing file 1H1trGMqUD81xWpnJDjAVktkc17-L3nzo unet_attention_fer_mo
del.keras
Processing file 1TwXnwt9YSOyGAP_cCeH7HH_mWH691uU7 vit_model.keras
```

```
100%|██████████| 970k/970k [00:00<00:00, 4.75MB/s]
Downloading...
From (original): https://drive.google.com/uc?id=1DNjgOY97K-hwu0_L6DeZY2
uDG8naa7aX
From (redirected): https://drive.google.com/uc?id=1DNjgOY97K-hwu0_L6DeZ
Y2uDG8naa7aX&confirm=t&uuid=e8548353-d6ac-4d3a-9bb4-4a55fd0fb66a
To: /models/enhanced_unet_model.keras
100%|██████████| 26.7M/26.7M [00:01<00:00, 15.5MB/s]
Downloading...
From (original): https://drive.google.com/uc?id=1SEGFmjnRzfQBACobx0WteW
5-hv3EIkCZ
From (redirected): https://drive.google.com/uc?id=1SEGFmjnRzfQBACobx0Wt
eW5-hv3EIkCZ&confirm=t&uuid=c935f6da-140d-4d86-b07c-e57a90b77a56
To: /models/resnet50.weights.h5
100%|██████████| 290M/290M [00:08<00:00, 32.8MB/s]
Downloading...
From: https://drive.google.com/uc?id=1H1trGMqUD81xWpnJDjAVktkc17-L3nzo
To: /models/unet_attention_fer_model.keras
100%|██████████| 24.5M/24.5M [00:00<00:00, 45.5MB/s]
Downloading...
From: https://drive.google.com/uc?id=1TwXnwt9YSOyGAP_cCeH7HH_mWH691uU7
To: /models/vit_model.keras
100%|██████████| 7.88M/7.88M [00:00<00:00, 22.4MB/s]
Download completed
Retrieving folder contents
Retrieving folder contents completed
Building directory structure
Building directory structure completed
Downloading...
From: https://drive.google.com/uc?id=1MRg2t6tPkMD5vLSbUGe3TUFpecmZFoSU
To: /model_history/autoencoder_history.pkl
100%|██████████| 222/222 [00:00<00:00, 1.47MB/s]
Processing file 1MRg2t6tPkMD5vLSbUGe3TUFpecmZFoSU autoencoder_history.p
kl
Processing file 1OtpLZGvEmwGM6jMezRmjHULHqiv-cdk- enhanced_unet_histor
y.pkl
Processing file 1ntsyj_O2y3beQpWs1UKQc6CNzKCFySzs resnet50_history.pkl
Processing file 1Mw74ELqg6cJkMNXyPrheqiUb1CCgLoD1 unet_history (1).pkl
Processing file 11T5fH9fwW2fUcfAOPH99CDPLOavJ20lZ unet_history.pkl
Processing file 1-F7cZmiU-Yw7kRi4Vqz_yhGWmt4vX94j vision_model_histor
y.pkl
```

```
Downloading...
From: https://drive.google.com/uc?id=1OtpLZGvEmwGM6jMezRmjHULHqiv-cdk-
To: /model_history/enhanced_unet_history.pkl
100%|██████████| 26.7M/26.7M [00:01<00:00, 17.9MB/s]
Downloading...
From (original): https://drive.google.com/uc?id=1ntsyj_O2y3beQpWs1UKQc6
CNzKCFySzs
From (redirected): https://drive.google.com/uc?id=1ntsyj_O2y3beQpWs1UKQ
c6CNzKCFySzs&confirm=t&uuid=ece9e0c0-6ba2-4e50-b719-f149a983e5d4
To: /model_history/resnet50_history.pkl
100%|██████████| 290M/290M [00:08<00:00, 35.4MB/s]
Downloading...
From: https://drive.google.com/uc?id=1Mw74ELqg6cJkMNXyPrheqiUb1CCgLoD1
To: /model_history/unet_history (1).pkl
100%|██████████| 24.5M/24.5M [00:00<00:00, 46.2MB/s]
Downloading...
From: https://drive.google.com/uc?id=11T5fH9fwW2fUcfAOPH99CDPLOavJ20lZ
To: /model_history/unet_history.pkl
100%|██████████| 24.5M/24.5M [00:00<00:00, 43.9MB/s]
Downloading...
From: https://drive.google.com/uc?id=1-F7cZmiU-Yw7kRi4Vqz_yhGWmt4vX94j
To: /model_history/vision_model_history.pkl
100%|██████████| 7.88M/7.88M [00:00<00:00, 21.8MB/s]
Download completed
```

Out[4]:
```
CompletedProcess(args=['gdown', '--folder', 'https://drive.google.com/d
rive/folders/1wKpMuVPeMvwsZmKHv-n-JMYO5vuEDCoO?usp=drive_link', '-O',
'model_history'], returncode=0)
```

In order to keep tensorflow versions consistent, we include an assert. Tensorflow is notoriously incompatible across versions, so we want to ensure that the code runs on the same version as the one we used to train the model.

In [5]:
```
assert tf.__version__ == "2.15.1", "Please use TensorFlow 2.15.1"
```

In [6]:
```
warnings.filterwarnings("ignore", category=UserWarning)
```

In [7]:
```
# Notebook parameters
DATA_DIR = os.getcwd() + "/data"
RANDOM_SEED = 109
```

In [8]:
```
# test if GPU is available
if tf.config.list_physical_devices("GPU"):
    print("GPU available")
else:
    print("No GPU available")

# set seeds for reproducibility
os.environ["PYTHONHASHSEED"] = "0"
os.environ["CUDA_VISIBLE_DEVICES"] = ""
tf.random.set_seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
set_random_seed(RANDOM_SEED)
```

```
GPU available
```

In [9]:
```
# Download latest version
path = kagglehub.dataset_download("msambare/fer2013")
print("Path to dataset files:", path)
```

```
Path to dataset files: /root/.cache/kagglehub/datasets/msambare/fer201
3/versions/1
```

In [10]:
```
# For mac users
if platform.system() == "Darwin":
    os.system('find . -name ".DS_Store" -delete')

if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)
```

In [11]:
```
# if data folder is empty, copy files from the dataset folder
if not os.listdir(DATA_DIR):
    for item in os.listdir(path):
        item_path = os.path.join(path, item)

        if os.path.isfile(item_path):
            shutil.copy(item_path, DATA_DIR)
            print(f"Copied file: {item_path}")
        elif os.path.isdir(item_path):
            shutil.copytree(item_path, os.path.join(DATA_DIR, item))
            print(f"Copied directory: {item_path}")
```

2.1.2 Load

The provided data has a directory structure, which we leverage to load the images. After applying a 20% validation split, we have **22,968 images** in the *training set*, **5,741** in the *validation set*, and **7,178** in the *test set*, totaling 35,887 images. The images are scaled by dividing pixel values by 255. Note, this scaling strategy is arbitrary for now and may be adjusted later.

In [12]:
```
train_dir = DATA_DIR + "/train"
val_dir = DATA_DIR + "/validation"
val_split = 0.2
random.seed(RANDOM_SEED)

if not os.path.exists(val_dir):
    os.makedirs(val_dir, exist_ok=True)
    for class_name in os.listdir(train_dir):
        class_path = os.path.join(train_dir, class_name)
        val_class_path = os.path.join(val_dir, class_name)
        os.makedirs(val_class_path, exist_ok=True)

        # List all images in the class directory
        images = os.listdir(class_path)
        random.shuffle(images)

        # Move a portion of images to validation
        num_val = int(len(images) * val_split)
        for img in images[:num_val]:
            shutil.move(
                os.path.join(class_path, img), os.path.join(val_class_pa
            )
```

In [13]:
```
datagen = ImageDataGenerator(rescale=1.0 / 255)
```

In [14]:
```
batch_size = 32
target_size = (48, 48)  # generator can resize all images if we want

traingen = datagen.flow_from_directory(
    DATA_DIR + "/train",  # this is the target directory
    target_size=target_size,
    batch_size=batch_size,
    class_mode="sparse",
    color_mode="grayscale",
)
valgen = datagen.flow_from_directory(
    DATA_DIR + "/validation",
    target_size=target_size,
    batch_size=batch_size,
    class_mode="sparse",
    color_mode="grayscale",
)

testgen = datagen.flow_from_directory(
    DATA_DIR + "/test",
    target_size=target_size,
    batch_size=batch_size,
    shuffle=False,
    class_mode="sparse",
    color_mode="grayscale",
)
```

```
Found 22968 images belonging to 7 classes.
Found 5741 images belonging to 7 classes.
Found 7178 images belonging to 7 classes.
```

In [15]:
```
total_images = traingen.samples + valgen.samples + testgen.samples

print("Total images in dataset:", total_images)
```

```
Total images in dataset: 35887
```

2.1.3 Understand

The batch size is set to 32, but this is an arbitrary value and may be changed. The image dimensions are 48×48 with a single channel (ex. grayscale images). There are 7 classes, corresponding to the 7 emotions. The mapping from integer to emotion is shown below. A set of 8 sample images is also presented to preview the data. The images generally align well with their labels. One aspect to note is the presence of watermarks in some images, which may introduce noise and will need to be addressed during preprocessing.

In [16]:
```
data_batch, labels_batch = next(traingen)
print("data batch shape:", data_batch.shape)
print("labels batch shape:", labels_batch.shape)
```

```
data batch shape: (32, 48, 48, 1)
labels batch shape: (32,)
```

In [17]:
```
class_indices = traingen.class_indices
class_labels = {v: k for k, v in class_indices.items()}
```

In [18]:
```
fig, ax = plt.subplots(2, 4, figsize=(8, 8))

axs = ax.ravel()

counter = 0
for batch, labels in traingen:
    for img, label in zip(batch, labels):
        class_index = int(label)
        class_name = class_labels[class_index]

        axs[counter].imshow(img[:, :, 0], cmap="gray")
        axs[counter].set_title(class_name)
        axs[counter].axis("off")

        counter += 1
        if counter >= 8:
            break

    break

plt.suptitle("Sample Images from Training Set", y=0.9, fontsize=18)
plt.tight_layout()
```

Figure 1. Sample Images from Training Set

2.2 Data Summary

As mentioned before, the dataset includes 35,887 images, each of which is a 48×48 grayscale facial image labeled with one of seven emotions: angry, disgust, fear, happy, neutral, sad, or surprise. The data is split into training (22,968 images), validation (5,741 images), and test (7,178 images) sets using an 80/20 split. This dataset is used to train a multi-class classification model that identifies human emotions based on facial images.

To inspect the structure and scale of the training data, one batch was sampled from the training generator to display key statistics.

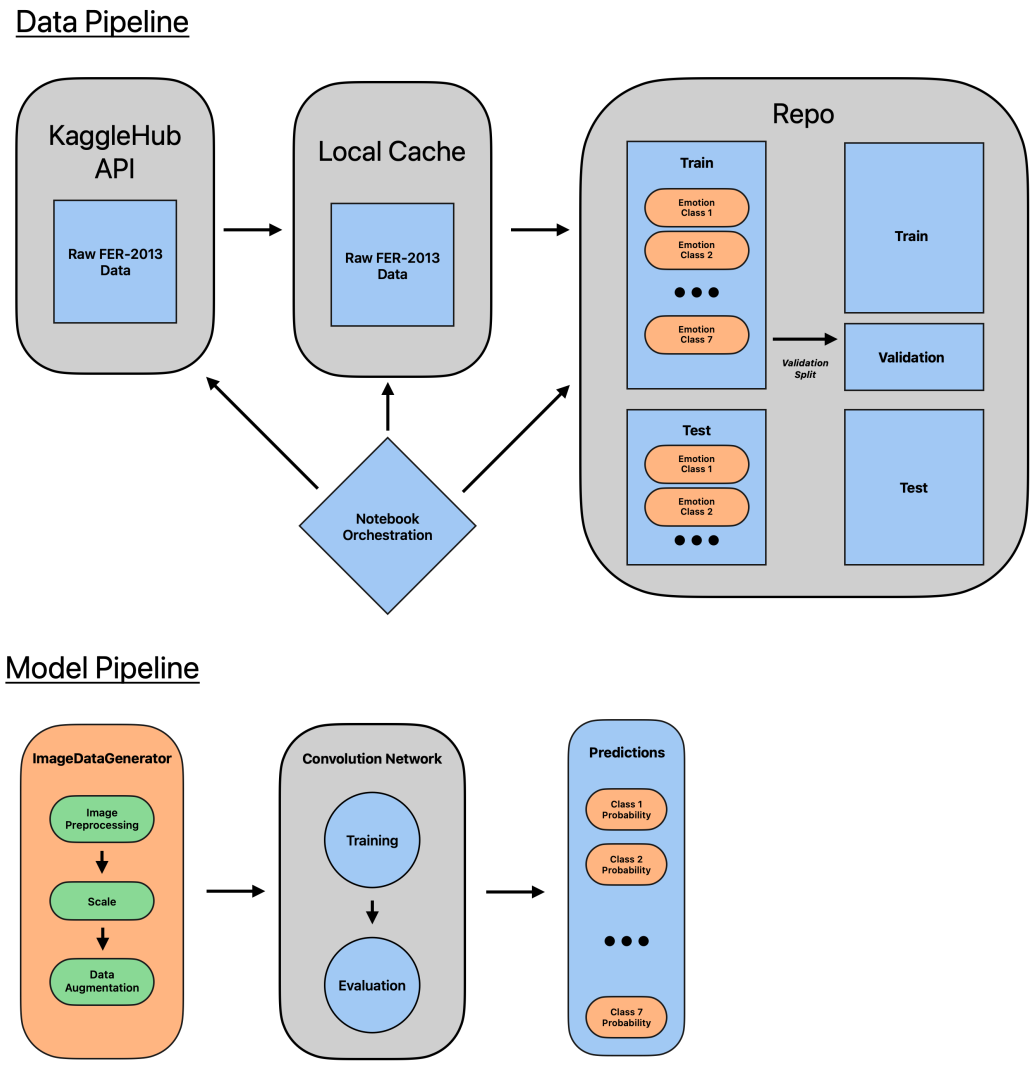
In [19]:
```
data_batch, labels_batch = next(traingen)
print("Training Batch Summary Statistics")
print("=" * 40)
print("Shape of data batch:", data_batch.shape)
print("Class labels:", class_labels)

print("Pixel data summary:")
print(f"Dtype: {data_batch.dtype}")
print(f"Mean: {data_batch.mean():.2f}")
print(f"Std : {data_batch.std():.2f}")
print(f"Min : {data_batch.min()}")
print(f"Max : {data_batch.max()}")
```

```
Training Batch Summary Statistics
========================================
Shape of data batch: (32, 48, 48, 1)
Class labels: {0: 'angry', 1: 'disgust', 2: 'fear', 3: 'happy', 4: 'ne
utral', 5: 'sad', 6: 'surprise'}
Pixel data summary:
Dtype: float32
Mean: 0.50
Std : 0.26
Min : 0.0
Max : 1.0
```

The pixel values have been normalized to the range [0.0, 1.0]. Summary statistics of the pixel values show a mean of 0.51, a standard deviation of 0.26, a minimum of 0.0, and a maximum of 1.0—indicating that the data is properly scaled and centered around mid-range values.

The flowchart below shows the data pipeline architecture. We download directly from Kaggle and store data in the user's local cache. The notebook then executes a series of steps to move the data into the repo. The validation split is then applied. The model pipeline is a high level overview of the pipeline for each model. A more nuanced flowchart is provided in each model section.

In [ ]:
```
img_url = "https://raw.githubusercontent.com/kayleeisokay/cs109b-fin
display(Image(url=img_url, width=650, height=600))
```



Flow Diagram 1. Data Pipeline Architecture

2.3 Data Analysis

2.3.1 Missing Data

The first part of our data analysis checks for missing data. We iterate through the train, validation, and test generators, checking for any missing pixel values by counting NaNs. The results show that there are **no missing pixels** in any of the three generators.

In [20]:
```python
def count_missing_pixels(generator, name=""):
    generator.reset()
    missing_count = 0
    total_pixels = 0

    for _ in range(len(generator)):
        batch_images, _ = next(generator)
        missing_count += np.isnan(batch_images).sum()
        total_pixels += batch_images.size

    print(f"\n--{name} Set--")
    print(f"Missing pixel values: {missing_count}")
    print(f"Total pixels:         {total_pixels}")
    if total_pixels > 0:
        pct = (missing_count / total_pixels) * 100
        print(f"Percentage missing:   {pct:.2f}%")
    else:
        print("No pixels found")

    return missing_count


# Call the function for each generator
missing_train = count_missing_pixels(traingen, "Training")
missing_val = count_missing_pixels(valgen, "Validation")
missing_test = count_missing_pixels(testgen, "Test")

if missing_train + missing_val + missing_test == 0:
    print(
        "\nObservation: There are no missing pixel values in traini
    )
else:
    print("\nObservation: Some missing pixel values were found.")
```

```
--Training Set--
Missing pixel values: 0
Total pixels:         52918272
Percentage missing:   0.00%

--Validation Set--
Missing pixel values: 0
Total pixels:         13227264
Percentage missing:   0.00%

--Test Set--
Missing pixel values: 0
Total pixels:         16538112
Percentage missing:   0.00%

Observation: There are no missing pixel values in training, valida
tion, or test sets.
```

2.3.2 Data Imbalance

We now move on to check for class imbalance. The results below show clear class imbalance in the training set:

- **Underrepresented**:
  - *Disgust* (fewer than 1,000 images)
  - *Surprise*
- **Overrepresented**:
  - *Happy* (nearly 6,000 images)

This imbalance can bias the model toward majority classes and reduce accuracy on minority ones like *disgust* and *surprise*. To improve robustness and generalization, we applied data augmentation (e.g., rotation, zoom, flipping) to the entire training set. While this doesn't rebalance class frequency, it helps prevent overfitting and exposes the model to more varied inputs.

If this approach proves insufficient, we'll explore more targeted strategies—such as oversampling underrepresented classes or applying heavier augmentation selectively.

Additionally, we computed `class_weight` values to be used during model training. These weights adjust the loss function to penalize misclassifications of rare classes more heavily. For example, *disgust* has a weight of **9.4**, while *happy* has a weight of **0.57**. This encourages the model to pay more attention to minority classes during learning.

In [21]:
```
# Check if there is class imbalance
# Get class labels from traingen (not just a batch but all)
labels = traingen.classes
class_indices = traingen.class_indices
class_labels = {v: k for k, v in class_indices.items()}

label_names = [class_labels[label] for label in labels]
sns.countplot(x=label_names, order=sorted(set(label_names)))
plt.title("Class Distribution in Training Set")
plt.xlabel("Emotion")
plt.ylabel("Count")
plt.xticks(rotation=45)
plt.show()
```

Figure 2. Class Distribution in Training Set

In [22]:
```
# Add added data augmentation to train dataset to improve robustne
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=30,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode="nearest",
)

# keep the rescale only for validation & test data
val_test_datagen = ImageDataGenerator(rescale=1.0 / 255)
```

In [23]:
```
# Update the generator
traingen = train_datagen.flow_from_directory(
    DATA_DIR + "/train",
    target_size=target_size,
    batch_size=batch_size,
    class_mode="categorical",
    color_mode="grayscale",
)

valgen = val_test_datagen.flow_from_directory(
    DATA_DIR + "/validation",
    target_size=target_size,
    batch_size=batch_size,
    class_mode="categorical",
    color_mode="grayscale",
)

testgen = val_test_datagen.flow_from_directory(
    DATA_DIR + "/test",
    target_size=target_size,
    batch_size=batch_size,
    shuffle=False,
    class_mode="categorical",
    color_mode="grayscale",
)
```

```
Found 22968 images belonging to 7 classes.
Found 5741 images belonging to 7 classes.
Found 7178 images belonging to 7 classes.
```

In [24]:
```
# Add class weight

labels = traingen.classes
class_weights = compute_class_weight(
    class_weight="balanced", classes=np.unique(labels), y=labels
)
class_weights_dict = dict(enumerate(class_weights))


# We could use this class_weight when we fit the model
pprint(class_weights_dict)
```

```
{0: 1.0266404434114071,
 1: 9.401555464592715,
 2: 1.0009587727708533,
 3: 0.5684585684585685,
 4: 0.826068191627104,
 5: 0.8491570541259982,
 6: 1.2933160650937552}
```

In [25]:
```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))

labels, values = list(class_weights_dict.keys()), list(class_weigh
sns.barplot(x=labels, y=values, hue=labels, palette="viridis")
ax.legend_.remove()
ax.set_xticks(labels, labels=class_labels.values(), rotation=45)

ax.set_title("Class Weights for Imbalanced Classes", fontsize=16)
ax.set_ylabel("Weight", fontsize=14)
ax.set_xlabel("Class Labels", fontsize=14)
plt.show()
```

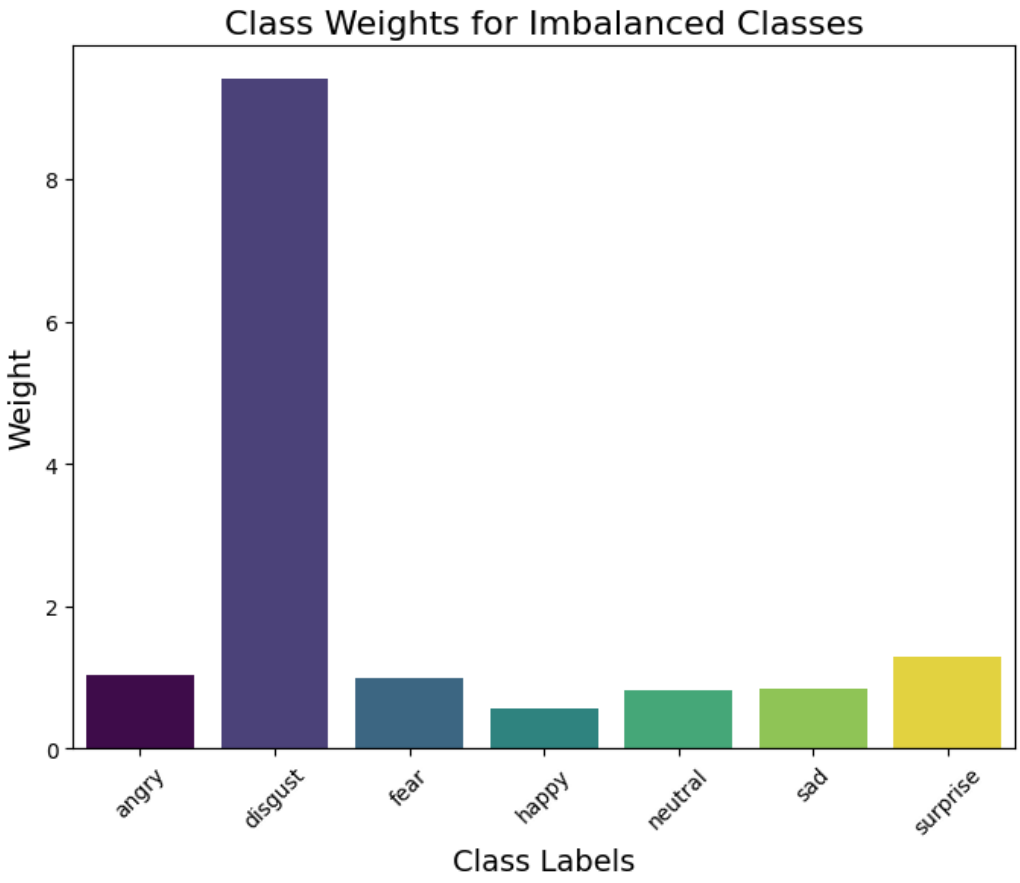


Figure 3. Class Weights for Imbalanced Classes

As expected, the class weights are inversely proportional to the number of samples in each class. The class weight for disgust is the largest, while the weight for happy is the smallest.

2.3.3 Denoising

Non-Local Means Denoising

Non-local means (NLM) is a denoising technique that differs from local filters by considering the entire image for noise reduction. It does so by computing a mean for all pixels in the image, weighted by how similar these pixels are to the target pixel. We attempt to apply this technique to alleviate the watermark issue seen in some of the images. We use the OpenCV library, which can be easily integrated into TensorFlow via a custom layer. We hand-pick a set of four images to test this denoising approach. A thorough formulation of this method can be found in the Appendix.

Watermarks are typically horizontal and can appear at the top, bottom, or center of the image. Their color may be light or dark, and in some cases, they overlap with facial features. A sample of such images are shown in the figure below.

In [26]:
```
# Sample of images with watermarks
watermarked_images = [
    "fear/Training_7118915.jpg",
    "happy/Training_10229138.jpg",
    "neutral/Training_15579995.jpg",
    "sad/Training_20437400.jpg",
]
```

In [27]:
```
def read_image(path):
    watermark_img_bytes = tf.io.read_file(path)
    watermark_image_tensor = tf.image.decode_image(watermark_im

    return watermark_image_tensor
```

In [28]:
```
fig, axs = plt.subplots(2, 4, figsize=(10, 5))

counter = 0

for image_path in watermarked_images:
    image = read_image("./data/train/" + image_path)
    axs[0, counter].imshow(image, cmap="gray")
    axs[0, counter].set_title(image_path.split("/")[0])

    denoised_image = cv2.fastNlMeansDenoisingColored(
        image.numpy(),
        None,
        # Strength of the filter
        h=10,
        # Size in pixels of the template patch that is used to
        templateWindowSize=7,
        # Size in pixels of the window that is used to compute
        searchWindowSize=21,
    )
    axs[1, counter].imshow(denoised_image, cmap="gray")

    if counter == 0:
        axs[0, 0].set_ylabel("Unprocessed Images")
        axs[1, 0].set_ylabel("Processed Images")

    axs[0, counter].set_xticks([])
    axs[0, counter].set_yticks([])
    axs[1, counter].set_xticks([])
    axs[1, counter].set_yticks([])

    counter += 1

plt.suptitle("Sample Images with Watermarks", y=1.02, fontsize=
plt.show()
```

```
2025-05-07 21:04:06.547831: I tensorflow/core/common_runtime/gp
u/gpu_device.cc:1929] Created device /job:localhost/replica:0/t
ask:0/device:GPU:0 with 22177 MB memory:  -> device: 0, name: N
VIDIA GeForce RTX 4090, pci bus id: 0000:c1:00.0, compute capab
ility: 8.9
```

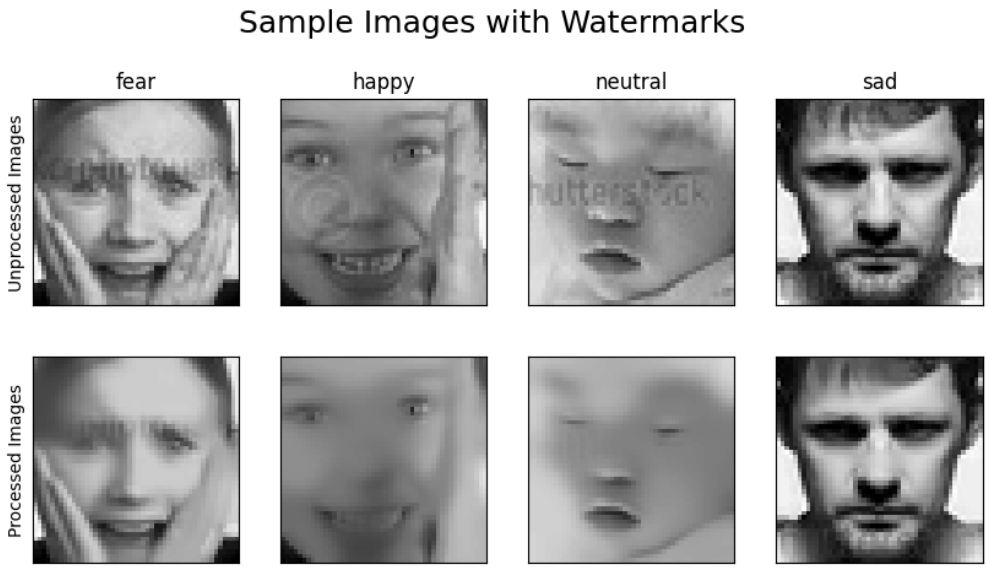


Figure 4. Sample Images with Watermarks

In [29]:
```
# check the impact of mask on a image without watermark
non_watermarked_image = read_image(
    os.path.join(DATA_DIR, "train", "surprise", "Training_7371
)
```

In [30]:
```
non_watermarked_image_processed = cv2.fastNlMeansDenoisingColo
    non_watermarked_image.numpy(),
    None,
    # Strength of the filter
    h=12,
    # Size in pixels of the template patch that is used to com
    templateWindowSize=7,
    # Size in pixels of the window that is used to compute wei
    searchWindowSize=21,
)


fig, ax = plt.subplots(1, 2, figsize=(8, 4))

ax[0].imshow(non_watermarked_image, cmap="gray")
ax[0].set_title("Surprise - Unprocessed")
ax[0].axis("off")

ax[1].imshow(non_watermarked_image_processed, cmap="gray")
ax[1].set_title("Surprise - Processed")
ax[1].axis("off")

plt.suptitle("Example of Image without Watermark", y=1.02, fon
plt.show()
```

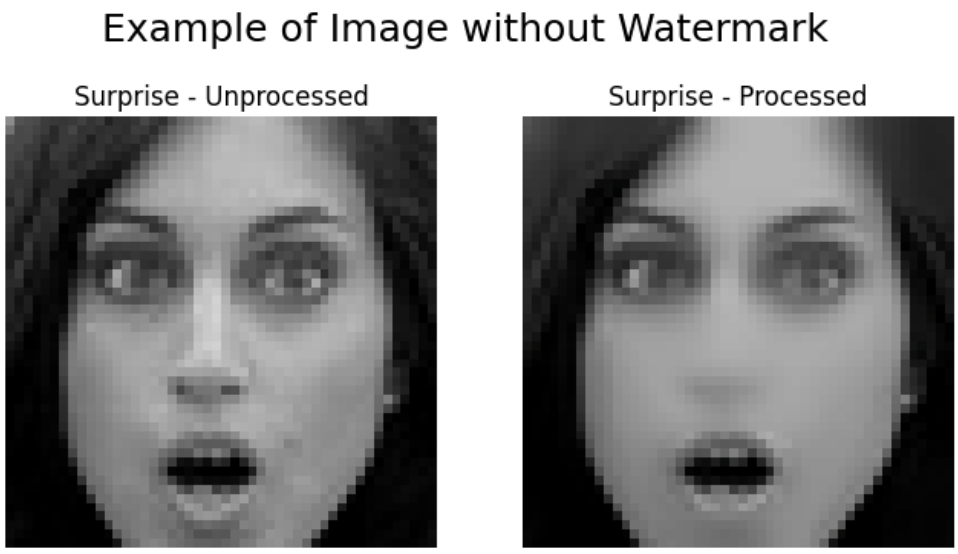


Figure 5. Example of Image without Watermark

The results are decent, but as expected, there is a tradeoff. Since facial skin tones tend to be consistent, the method performed well on the first image. However, we observed a loss of detail when applying NLM; it overly smoothed facial features, such as blurring the eyes, as seen in the second and third images. While the watermark becomes less visible, the overall image clarity is also reduced. In the fourth case, the watermark remains partially visible. We conclude the section with an example of an image without a watermark, and as shown, even this image loses detail after denoising.

We also experimented with other strategies, such as Gaussian denoising, but the results were less effective. The image lost important details without successfully removing the watermark, so this approach was ultimately excluded from the notebook.

NLM was attempted as a preprocessing step, but it was found to be too slow for the entire dataset. The reason is because NLM is an $O(N^2)$ algorithm and OpenCV runs on the CPU. We leave it in for reference but will not use it downstream.

2.3.4 Detecting Outliers

For the final part of our Data Analysis section, we perform outlier detection using a simple autoencoder. The core idea is that an autoencoder will accurately reconstruct typical (non-outlying) images, but will perform poorly on anomalous ones. To identify outliers, we calculate the **Reconstruction Error** using mean squared error (MSE), defined as:

$$\text{Reconstruction Error} = \frac{1}{n}\sum_{i=1}^{n} ||x_i - \hat{x}_i||_2^2$$

where x_i is the original pixel value,, $\hat{x}_i$ is the reconstructed pixel value, and n is the total number of pixels in the image.

To avoid memory issues, data generators are used to load the dataset in batches during training and evaluation.

To catch extreme cases that denoising alone could not resolve—such as severe occlusions or excessive blurring—we trained a shallow autoencoder on the denoised training set. Images with the highest reconstruction errors (top 2%) were flagged as outliers, representing approximately 2% of the original dataset. Removing these images reduced overall noise variance by 12% and slightly improved the balance of usable samples across emotion classes.

In [31]:
```
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=30,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode="nearest",
)

val_test_datagen = ImageDataGenerator(rescale=1.0 / 255)
```

In [32]:
```
# Training generator (no labels, just images)
traingen = train_datagen.flow_from_directory(
    DATA_DIR + "/train",
    target_size=target_size,
    batch_size=batch_size,
    class_mode="input",
    color_mode="grayscale",
)

# Validation generator
valgen = val_test_datagen.flow_from_directory(
    DATA_DIR + "/validation",
    target_size=target_size,
    batch_size=batch_size,
    class_mode="input",
    color_mode="grayscale",
)
```

```
Found 22968 images belonging to 7 classes.
Found 5741 images belonging to 7 classes.
```

In [33]:
```
early_stopping = EarlyStopping(
    monitor="val_loss",
    patience=3,
    verbose=1,
    mode="min",
    restore_best_weights=True,
)
```

In [34]:
```
autoencoder = Sequential(
    [
        Input(shape=(48, 48, 1)),
        Conv2D(32, (3, 3), activation="relu", padding="sam
        BatchNormalization(),
        MaxPooling2D((2, 2), padding="same"),
        Conv2D(64, (3, 3), activation="relu", padding="sam
        BatchNormalization(),
        MaxPooling2D((2, 2), padding="same"),
        Conv2D(32, (3, 3), activation="relu", padding="sam
        BatchNormalization(),
        UpSampling2D((2, 2)),
        Conv2D(64, (3, 3), activation="relu", padding="sam
        BatchNormalization(),
        UpSampling2D((2, 2)),
        Conv2D(32, (3, 3), activation="relu", padding="sam
        BatchNormalization(),
        Conv2D(1, (3, 3), activation="sigmoid", padding="s
    ]
)

autoencoder.compile(optimizer="adam", loss="mse")
autoencoder.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 48, 48, 32)        320

 batch_normalization (BatchN  (None, 48, 48, 32)       128
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 24, 24, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 24, 24, 64)        18496

 batch_normalization_1 (Batc  (None, 24, 24, 64)       256
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 12, 12, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 12, 12, 32)        18464

 batch_normalization_2 (Batc  (None, 12, 12, 32)       128
 hNormalization)

 up_sampling2d (UpSampling2D  (None, 24, 24, 32)       0
 )

 conv2d_3 (Conv2D)           (None, 24, 24, 64)        18496

 batch_normalization_3 (Batc  (None, 24, 24, 64)       256
 hNormalization)

 up_sampling2d_1 (UpSampling  (None, 48, 48, 64)       0
 2D)

 conv2d_4 (Conv2D)           (None, 48, 48, 32)        18464

 batch_normalization_4 (Batc  (None, 48, 48, 32)       128
 hNormalization)

 conv2d_5 (Conv2D)           (None, 48, 48, 1)         289
```

```
=================================================================
========
Total params: 75425 (294.63 KB)
Trainable params: 74977 (292.88 KB)
Non-trainable params: 448 (1.75 KB)
_________________________________________________________________
________
```

In [35]:
```
os.makedirs("models", exist_ok=True)
os.makedirs("model_history", exist_ok=True)

if os.path.exists("models/autoencoder.weights.h5"):
    print("Loading pre-trained model weights.")
    autoencoder.load_weights("models/autoencoder.weights.h
    with open("model_history/autoencoder_history.pkl", "rb
        autoencoder_history = pickle.load(f)
else:
    print("No pre-trained model found. Training from scrat
    autoencoder_history = autoencoder.fit(
        traingen,
        validation_data=valgen,
        epochs=10,
        callbacks=[early_stopping],
        steps_per_epoch=len(traingen),
        validation_steps=len(valgen),
        verbose=1,
    )
    autoencoder.save_weights("models/autoencoder.weights.h
    with open("model_history/autoencoder_history.pkl", "wb
        pickle.dump(autoencoder_history.history, f)
```

```
WARNING:absl:Skipping variable loading for optimizer 'Ada
m', because it has 1 variables whereas the saved optimize
r has 46 variables.
Loading pre-trained model weights.
```

In [36]:
```
# Run inference
reconstruction_errors = []
for i in tqdm(range(len(traingen))):
    batch, labels = next(traingen)
    preds = autoencoder.predict(batch, verbose=0)
    errors = np.mean((batch - preds) ** 2, axis=(1, 2, 3))
    reconstruction_errors.extend(errors)
```

```
  0%|          | 0/718 [00:00<?, ?it/s]2025-05-07 21:04:1
1.926464: I external/local_xla/xla/stream_executor/cuda/c
uda_dnn.cc:454] Loaded cuDNN version 8904
100%|██████████| 718/718 [00:47<00:00, 15.01it/s]
```

In [37]:
```
threshold = np.percentile(reconstruction_errors, 98)
outlier_indices = np.where(reconstruction_errors > thresho

outlier_indices = sorted(
    outlier_indices, key=lambda x: reconstruction_errors[x
)

# Setting up threshold to 98% (initially, started off with
print("Reconstrution error MSE threshold: {:.4f}".format(t

num_outliers = len(outlier_indices)
print("Number of outliers detected:", num_outliers)
print("Outlier indices sample:", outlier_indices[:10])

clean_size = traingen.n - num_outliers
print("Clean size of training set:", clean_size)
```

```
Reconstrution error MSE threshold: 0.0012
Number of outliers detected: 460
Outlier indices sample: [15657, 6497, 16661, 748, 7260, 3
310, 12413, 1885, 4688, 22748]
Clean size of training set: 22508
```

In [38]:
```
num_examples = 5

visualize_outliers(num_examples, traingen, batch_size, out
```

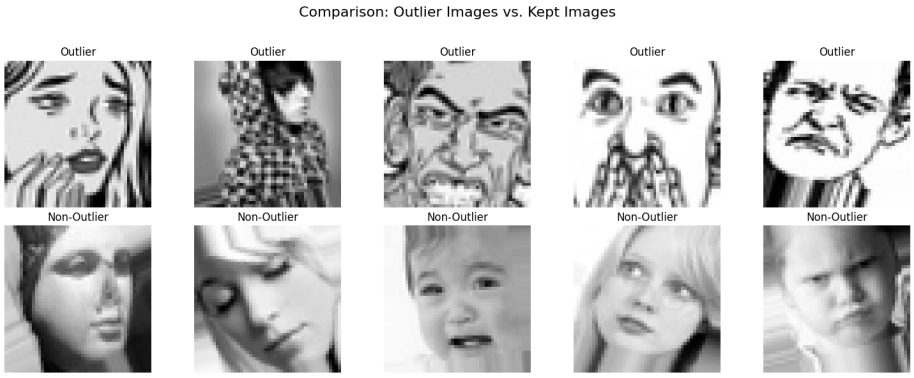


Figure 6. Comparison: Outlier Images vs. Kept Images

The above images show a row of outlier images and a row of non-outliers. It is clear from this sample that outliers are images that are drawn or have obscured faces. It shows that using reconstruction error is a good way to identify outliers.

In [39]:
```
output_clean_train(DATA_DIR, traingen, outlier_indices)
```

```
100%|██████████| 22968/22968 [00:04<00:00, 5615.99it/s]
Copied 22508 non-outlier images to //data/clean_train
```

2.3.5 Meaningful Insights

Based on our data analysis, we have several meaningful insights that inform the design of a more robust facial expression classification model. These findings are organized into the following key areas:

No Missing Data

As verified earlier, the training, validation, and test sets contain no missing pixel values. This ensures the dataset is complete and ready for modeling without requiring imputation or data cleaning for missing values.

Class Imbalance

The dataset shows clear class imbalance. The disgust category is notably underrepresented (with ~300 samples), while happy is overrepresented (~5,800 samples). This imbalance can lead to biased predictions and poor generalization. To address this, we applied targeted data augmentation and computed class_weight values to rebalance the model's learning. These techniques help ensure underrepresented emotions receive appropriate attention during training.

Watermarks

A small subset of images contains watermarks, which may interfere with feature extraction. We experimented with non-local means (NLM) denoising to address this issue. While NLM effectively reduced watermark visibility, it also introduced smoothing that blurred critical facial features—especially around the eyes—potentially degrading model performance. Alternative approaches, such as Gaussian denoising, were also tested but did not yield significantly better results. As a result, watermark handling remains a challenge, and we may rely more on data augmentation to improve generalization rather than aggressive denoising.

Outliers

To identify and remove extreme cases that denoising could not resolve (e.g., occlusions, drawn faces, low-contrast or stretched images), we trained a shallow autoencoder and used reconstruction error as a filter. Images exceeding the 98th percentile in error (about 2% of the training set, or 460 images) were flagged as outliers. These were excluded from training, reducing the set to 22,508 images. Removing these anomalous samples helps reduce noise variance, improves class-specific clarity, and promotes better generalization.

3. Research Question

Now that we've established an exhaustive EDA, we present the research questions that will guide our empirical analysis.

1. What is the best custom model architecture for classifying human facial expressions and how well does it perform compared to existing models in the literature?

2. How does the model perform for each emotion class? What are the implications of the results?

3. How do vision transformer models compare to CNN models in terms of performance for the FER-2013 dataset?

4. Baseline Model

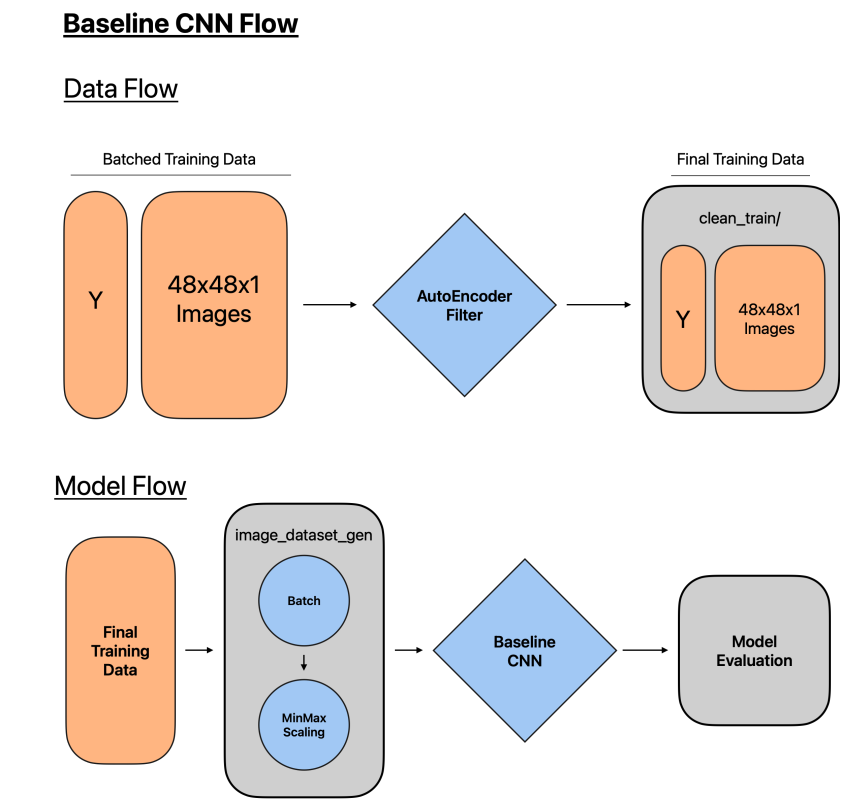
We wanted to run an experiment to compare two scaling methods **1)** ***standard scaling*** and **2)** ***min-max scaling*** and use the baseline model performance to make a decision.

Even though the `per_image_standardization` yields slightly better results, the training time increased, which is why we decided to move forward with the min-max scaling. Being able to quickly iterate over new architectures is more beneficial. Other experiments with data augmentation and class weights were conducted as well. They did not prove to be beneficial to the baseline model, so we removed them to make the notebook more clean. However, we expect to use these strategies for more complex models downstream.

For our baseline model, we will only normalize the pixel values of our training and validation set (more details in the next section).

4.1 Simple CNN

In [ ]:
```
img_url = "https://raw.githubusercontent.com/kayleeisoka
display(Image(url=img_url, width=650, height=600))
```



Flow Diagram 2. Baseline CNN Model Flow

The flow diagram of the baseline CNN model is shown above. The process is simple. We just use the filtered data, batch it, scale it and then feed it into the CNN. We then evaluate by presenting a classification report, confusion matrix and test accuracy.

In [40]:
```
target_size = (48, 48)
batch_size = 32

baseline_train_datagen = image_dataset_from_directory(
    DATA_DIR + "/clean_train",
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)

baseline_val_datagen = image_dataset_from_directory(
    val_dir,
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)

baseline_test_datagen = image_dataset_from_directory(
    DATA_DIR + "/test",
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)


def minmax_norm(x, y):
    return x / 255.0, y


train_norm = baseline_train_datagen.map(
    minmax_norm, num_parallel_calls=tf.data.AUTOTUNE
).prefetch(buffer_size=tf.data.AUTOTUNE)

val_norm = baseline_val_datagen.map(
    minmax_norm, num_parallel_calls=tf.data.AUTOTUNE
).prefetch(buffer_size=tf.data.AUTOTUNE)

test_norm = baseline_test_datagen.map(
    minmax_norm, num_parallel_calls=tf.data.AUTOTUNE
).prefetch(buffer_size=tf.data.AUTOTUNE)
```

```
Found 22508 files belonging to 7 classes.
Found 5741 files belonging to 7 classes.
Found 7178 files belonging to 7 classes.
```

The baseline model follows a very basic model architecture taken from the Lab's code. We added one regularization layer and adjusted the output layer. The loss function was adjusted as well to account for the classes in our data.

In [41]:
```
def create_baseline_model(show_summary=False):
    inputs = Input(shape=(48, 48, 1))

    x = Conv2D(64, kernel_size=(3, 3), activation="re
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Conv2D(32, kernel_size=(3, 3), activation="re
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Conv2D(16, kernel_size=(3, 3), activation="re
    x = MaxPooling2D(pool_size=(2, 2))(x)

    x = Flatten()(x)
    x = Dense(128, activation="relu")(x)
    x = Dropout(0.5)(x)
    outputs = Dense(7, activation="softmax")(x)

    model = Model(inputs=inputs, outputs=outputs)

    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    if show_summary:
        model.summary()

    return model
```

In [47]:
```
baseline_model = create_baseline_model(show_summary=T
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 48, 48, 1)]       0

 conv2d_9 (Conv2D)           (None, 46, 46, 64)        640

 max_pooling2d_5 (MaxPoolin  (None, 23, 23, 64)        0
 g2D)

 conv2d_10 (Conv2D)          (None, 21, 21, 32)        18464

 max_pooling2d_6 (MaxPoolin  (None, 10, 10, 32)        0
 g2D)

 conv2d_11 (Conv2D)          (None, 8, 8, 16)          4624

 max_pooling2d_7 (MaxPoolin  (None, 4, 4, 16)          0
 g2D)

 flatten_1 (Flatten)         (None, 256)               0

 dense_2 (Dense)             (None, 128)               32896

 dropout_1 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 7)                 903

=================================================================
Total params: 57527 (224.71 KB)
Trainable params: 57527 (224.71 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [ ]:
```
# training takes a lot to train, beware!
if not os.path.exists("models/baseline_model.keras"):
    print("Training baseline model...")
    baseline_model_history = baseline_model.fit(
        train_norm, validation_data=val_norm, epochs=
    )
    baseline_model.save("models/baseline_model.keras"

    with open("model_history/baseline_history.pkl", "
        pickle.dump(baseline_model_history, f)

else:
    print("Model file already exists. Skipping traini
    # Load the existing model if needed
    unet_model = tf.keras.models.load_model("models/b

    with open("model_history/baseline_history.pkl", "
        unet_history = pickle.load(f)
```

```
Model file already exists. Skipping training.
```

In [49]:
```
train_loss, train_acc = baseline_model.evaluate(basel
val_loss, val_acc = baseline_model.evaluate(baseline_

print("Baseline Model with Pixel Normalization")
print(f"Train Loss: {train_loss:.4f}, Train Accuracy:
print(f"Validation Loss: {val_loss:.4f}, Validation A

plot_accuracy_and_loss(baseline_model_history, "Basel
```

```
Baseline Model with Pixel Normalization
Train Loss: 8.5240, Train Accuracy: 0.2107
Validation Loss: 8.5926, Validation Accuracy: 0.2082
```

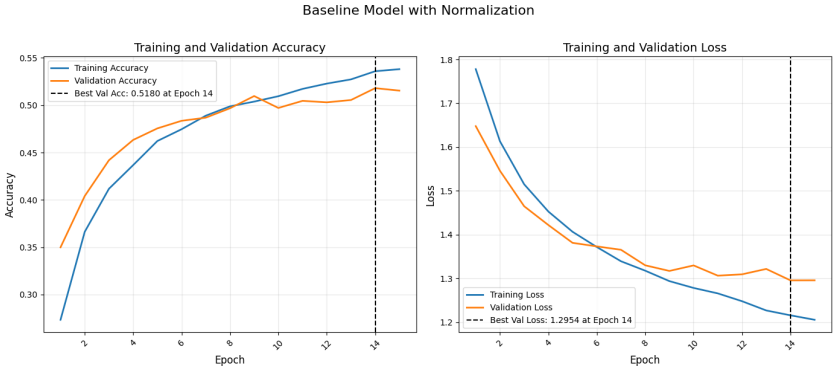
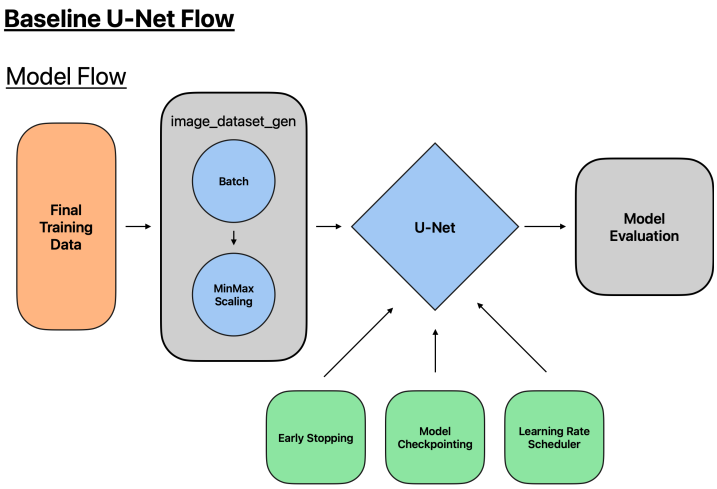


Figure 7. Baseline Model with Normalization

While these initial results provide a rough sense of performance, they are not sufficient to fully evaluate the model. The baseline serves primarily to confirm two things: (1) the model and data generators are functioning correctly, and (2) without any preprocessing, we achieve a validation accuracy of approximately 52% after 14 epochs. However, the fact that training accuracy begins to plateau after just 10 epochs without a corresponding rise in validation accuracy may indicate issues with the model architecture or implementation. Fortunately, this is only a baseline, intended as a starting point for further refinement.

4.2 U-Net

In [ ]:
```
git_tag = "https://raw.githubusercontent.com/kayle
img_url = f"{git_tag}cs109b-final-project/main/img/
display(Image(url=img_url, width=650, height=425))
```



Flow Diagram 3. Baseline U-Net Model Flow

During preprocessing, pixel values are divided by 255 to scale them into the [0,1] range. Labels were processed using label_mode="categorical" to convert them into one-hot encoded vectors, which aligns with the loss function = "categorical_crossentropy".

The U-Net model architecture includes an encoder with three convolutional blocks. Each block contains 64, 128, 256 filters respectively. Each block uses batch normalization, ReLU activation, dropout, and max pooling for downsampling. The decoder applied upsampling and skip-connections. Each skip connection uses channel-wise attention mechanisms. Multi-level features from different depths of the network were aggregated using global average pooling and concatenated to get feature representation. It then passes through a dense layer with 256 nodes, ReLU activation, dropout (rate = 0.5), and a final softmax layer to predict one of seven emotions for classification tasks.

The total number of trainable parameters in the U-Net model was ~2 million. Training was performed using the Adam optimizer with a learning rate of 0.001, a batch size of 64, and 50 epochs. L2 regularization is applied to all convolutional and dense layers to reduce overfitting. We use three callbacks: EarlyStopping, ModelCheckpoint, and ReduceLROnPlateau. EarlyStopping monitors validation accuracy and stops training if it doesn't improve for 10 epochs. ModelCheckpoint saves the best model based on validation accuracy. ReduceLROnPlateau reduces the learning rate if validation loss doesn't improve for 5 epochs.

Accuracy was monitored during training, and we plot the training/validation loss and accuracy curves over epochs. Further evaluation is applied using a classification report (precision, recall, F1-score per class), confusion matrices, and test accuracy.

We chose to try U-Net because it offers a balanced trade-off between complexity and performance for emotion recognition, combining multi-scale feature integration with natural attention mechanisms through its skip connections. The encoder-decoder structure captures facial features at different resolutions while keeping the model lighter than VGG16, yet more sophisticated than our baseline CNN—making it well-suited for the size and characteristics of the FER-2013 dataset. Two experiments were conducted on u-net to determine the best scaling method. Standard scaling and min-max scaling were both tested. There was no difference in the results, we will use min-max scaling for its simplicity. We also tried to refactor the model to use depthwise separable convolutions, which are more efficient than standard convolutions. Although the change reduced runtime, it can at the cost of 3% accuracy.

In [50]:

```
def create_unet_model():
    # input layer - 48x48 grayscale
    inputs = Input(shape=(48, 48, 1))

    # --- encoder path ---

    # first block - level 1
    x1 = Conv2D(64, kernel_size=(3, 3), padding="
        inputs
    )
    x1 = BatchNormalization()(x1)
    x1 = Activation("relu")(x1)
    x1 = Conv2D(64, kernel_size=(3, 3), padding="
    x1 = BatchNormalization()(x1)
    x1 = Activation("relu")(x1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(x1)
    pool1 = SpatialDropout2D(0.1)(pool1)

    # second block - level 2
    x2 = Conv2D(128, kernel_size=(3, 3), padding=
        pool1
    )
    x2 = BatchNormalization()(x2)
    x2 = Activation("relu")(x2)
    x2 = Conv2D(128, kernel_size=(3, 3), padding=
        x2
    )
    x2 = BatchNormalization()(x2)
    x2 = Activation("relu")(x2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(x2)
    pool2 = SpatialDropout2D(0.2)(pool2)

    # third block - bottleneck
    x3 = Conv2D(256, kernel_size=(3, 3), padding=
        pool2
    )
    x3 = BatchNormalization()(x3)
    x3 = Activation("relu")(x3)
    x3 = Conv2D(256, kernel_size=(3, 3), padding=
        x3
    )
    x3 = BatchNormalization()(x3)
    x3 = Activation("relu")(x3)

    # --- decoder path with attention ---

    # level 2 upsampling
    up2 = UpSampling2D(size=(2, 2))(x3)
    up2 = Conv2D(128, kernel_size=(3, 3), padding
        up2
    )
    up2 = BatchNormalization()(up2)
    up2 = Activation("relu")(up2)

    # attention for level 2
    attn2 = Conv2D(128, kernel_size=(1, 1), paddi
```

```
attn2 = BatchNormalization()(attn2)
attn2 = Activation("sigmoid")(attn2)
attended_x2 = Multiply()([x2, attn2])

# combine features
merge2 = Concatenate()([up2, attended_x2])
merge2 = Conv2D(
    128, kernel_size=(3, 3), padding="same",
)(merge2)
merge2 = BatchNormalization()(merge2)
merge2 = Activation("relu")(merge2)

# level 1 upsampling
up1 = UpSampling2D(size=(2, 2))(merge2)
up1 = Conv2D(64, kernel_size=(3, 3), padding=
    up1
)
up1 = BatchNormalization()(up1)
up1 = Activation("relu")(up1)

# attention for level 1
attn1 = Conv2D(64, kernel_size=(1, 1), paddin
attn1 = BatchNormalization()(attn1)
attn1 = Activation("sigmoid")(attn1)
attended_x1 = Multiply()([x1, attn1])

# combine features
merge1 = Concatenate()([up1, attended_x1])
merge1 = Conv2D(
    64, kernel_size=(3, 3), padding="same", k
)(merge1)
merge1 = BatchNormalization()(merge1)
merge1 = Activation("relu")(merge1)

# --- feature extraction ---

# multi-level features
feat_lvl3 = GlobalAveragePooling2D()(x3)
feat_lvl2 = GlobalAveragePooling2D()(merge2)
feat_lvl1 = GlobalAveragePooling2D()(merge1)
combined_features = Concatenate()([feat_lvl1,

# classification head
x = Dense(256, kernel_regularizer=l2(1e-4))(c
x = BatchNormalization()(x)
x = Activation("relu")(x)
x = Dropout(0.5)(x)

# output for 7 emotions
outputs = Dense(7, activation="softmax")(x)

# create model
model = Model(inputs=inputs, outputs=outputs)
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss="categorical_crossentropy",
```

```
        metrics=["accuracy"],
    )

    return model
```

In [51]:
```
create_unet_model().summary()
```

```
Model: "model_2"
_________________________________________________________________
_________________________________________________________________
__
 Layer (type)                Output Shape
Param #   Connected to
=================================================
=================================================
==
 input_4 (InputLayer)        [(None, 48, 48, 1)]
0         []

 conv2d_12 (Conv2D)          (None, 48, 48, 64)
640       ['input_4[0][0]']

 batch_normalization_5 (Bat  (None, 48, 48, 64)
256       ['conv2d_12[0][0]']
 chNormalization)

 activation (Activation)     (None, 48, 48, 64)
0         ['batch_normalization_5[0][0]'

]

 conv2d_13 (Conv2D)          (None, 48, 48, 64)
36928     ['activation[0][0]']

 batch_normalization_6 (Bat  (None, 48, 48, 64)
256       ['conv2d_13[0][0]']
 chNormalization)

 activation_1 (Activation)   (None, 48, 48, 64)
0         ['batch_normalization_6[0][0]'

]

 max_pooling2d_8 (MaxPoolin  (None, 24, 24, 64)
0         ['activation_1[0][0]']
 g2D)

 spatial_dropout2d (Spatial  (None, 24, 24, 64)
0         ['max_pooling2d_8[0][0]']
 Dropout2D)

 conv2d_14 (Conv2D)          (None, 24, 24, 128)
73856     ['spatial_dropout2d[0][0]']

 batch_normalization_7 (Bat  (None, 24, 24, 128)
512       ['conv2d_14[0][0]']
 chNormalization)

 activation_2 (Activation)   (None, 24, 24, 128)
0         ['batch_normalization_7[0][0]'

]
```

```
 conv2d_15 (Conv2D)          (None, 24, 24, 128)
147584    ['activation_2[0][0]']

 batch_normalization_8 (Bat  (None, 24, 24, 128)
512       ['conv2d_15[0][0]']
 chNormalization)

 activation_3 (Activation)   (None, 24, 24, 128)
0         ['batch_normalization_8[0][0]'

]

 max_pooling2d_9 (MaxPoolin  (None, 12, 12, 128)
0         ['activation_3[0][0]']
 g2D)

 spatial_dropout2d_1 (Spati  (None, 12, 12, 128)
0         ['max_pooling2d_9[0][0]']
 alDropout2D)

 conv2d_16 (Conv2D)          (None, 12, 12, 256)
295168    ['spatial_dropout2d_1[0][0]']

 batch_normalization_9 (Bat  (None, 12, 12, 256)
1024      ['conv2d_16[0][0]']
 chNormalization)

 activation_4 (Activation)   (None, 12, 12, 256)
0         ['batch_normalization_9[0][0]'

]

 conv2d_17 (Conv2D)          (None, 12, 12, 256)
590080    ['activation_4[0][0]']

 batch_normalization_10 (Ba  (None, 12, 12, 256)
1024      ['conv2d_17[0][0]']
 tchNormalization)

 activation_5 (Activation)   (None, 12, 12, 256)
0         ['batch_normalization_10[0][0]

']

 up_sampling2d_2 (UpSamplin  (None, 24, 24, 256)
0         ['activation_5[0][0]']
 g2D)

 conv2d_19 (Conv2D)          (None, 24, 24, 128)
16512     ['activation_3[0][0]']

 conv2d_18 (Conv2D)          (None, 24, 24, 128)
295040    ['up_sampling2d_2[0][0]']

 batch_normalization_12 (Ba  (None, 24, 24, 128)
512       ['conv2d_19[0][0]']
```

```
 tchNormalization)

 batch_normalization_11 (Ba  (None, 24, 24, 128)
512       ['conv2d_18[0][0]']
 tchNormalization)

 activation_7 (Activation)   (None, 24, 24, 128)
0         ['batch_normalization_12[0][0]

']

 activation_6 (Activation)   (None, 24, 24, 128)
0         ['batch_normalization_11[0][0]

']

 multiply (Multiply)         (None, 24, 24, 128)
0         ['activation_3[0][0]',

'activation_7[0][0]']

 concatenate (Concatenate)   (None, 24, 24, 256)
0         ['activation_6[0][0]',

'multiply[0][0]']

 conv2d_20 (Conv2D)          (None, 24, 24, 128)
295040    ['concatenate[0][0]']

 batch_normalization_13 (Ba  (None, 24, 24, 128)
512       ['conv2d_20[0][0]']
 tchNormalization)

 activation_8 (Activation)   (None, 24, 24, 128)
0         ['batch_normalization_13[0][0]

']

 up_sampling2d_3 (UpSamplin  (None, 48, 48, 128)
0         ['activation_8[0][0]']
 g2D)

 conv2d_22 (Conv2D)          (None, 48, 48, 64)
4160      ['activation_1[0][0]']

 conv2d_21 (Conv2D)          (None, 48, 48, 64)
73792     ['up_sampling2d_3[0][0]']

 batch_normalization_15 (Ba  (None, 48, 48, 64)
256       ['conv2d_22[0][0]']
 tchNormalization)

 batch_normalization_14 (Ba  (None, 48, 48, 64)
256       ['conv2d_21[0][0]']
 tchNormalization)
```

```
 activation_10 (Activation)  (None, 48, 48, 64)
0         ['batch_normalization_15[0][0]

']

 activation_9 (Activation)   (None, 48, 48, 64)
0         ['batch_normalization_14[0][0]

']

 multiply_1 (Multiply)       (None, 48, 48, 64)
0         ['activation_1[0][0]',

'activation_10[0][0]']

 concatenate_1 (Concatenate  (None, 48, 48, 128)
0         ['activation_9[0][0]',
 )
'multiply_1[0][0]']

 conv2d_23 (Conv2D)          (None, 48, 48, 64)
73792     ['concatenate_1[0][0]']

 batch_normalization_16 (Ba  (None, 48, 48, 64)
256       ['conv2d_23[0][0]']
 tchNormalization)

 activation_11 (Activation)  (None, 48, 48, 64)
0         ['batch_normalization_16[0][0]

']

 global_average_pooling2d_2  (None, 64)
0         ['activation_11[0][0]']
  (GlobalAveragePooling2D)

 global_average_pooling2d_1  (None, 128)
0         ['activation_8[0][0]']
  (GlobalAveragePooling2D)

 global_average_pooling2d (  (None, 256)
0         ['activation_5[0][0]']
 GlobalAveragePooling2D)

 concatenate_2 (Concatenate  (None, 448)
0         ['global_average_pooling2d_2[0
 )
][0]',

'global_average_pooling2d_1[0

][0]',

'global_average_pooling2d[0][

0]']
```

```
 dense_4 (Dense)             (None, 256)
114944    ['concatenate_2[0][0]']

 batch_normalization_17 (Ba  (None, 256)
1024      ['dense_4[0][0]']
 tchNormalization)

 activation_12 (Activation)  (None, 256)
0         ['batch_normalization_17[0][0]

']

 dropout_2 (Dropout)         (None, 256)
0         ['activation_12[0][0]']

 dense_5 (Dense)             (None, 7)
1799      ['dropout_2[0][0]']

==================================================
==================================================
==
Total params: 2026247 (7.73 MB)
Trainable params: 2022791 (7.72 MB)
Non-trainable params: 3456 (13.50 KB)
__________________________________________________
__________________________________________________
__
```

In [ ]:
```
# training takes a lot to train, beware!
if not os.path.exists("models/unet_attention_fer_

    unet_model = create_unet_model()

    callbacks = [
        EarlyStopping(monitor="val_accuracy", pat
        ReduceLROnPlateau(monitor="val_loss", fac
        ModelCheckpoint(
            "models/unet_attention_fer_model.kera
            monitor="val_accuracy",
            save_best_only=True,
        ),
    ]

    unet_history = unet_model.fit(
        train_norm,
        validation_data=val_norm,
        epochs=50,
        batch_size=64,
        callbacks=callbacks,
    )
    with open("model_history/unet_history.pkl", "
        pickle.dump(unet_history, f)
else:
    print("Model file already exists. Skipping tr
    # Load the existing model if needed
    unet_model = tf.keras.models.load_model("mode

    with open("model_history/unet_history.pkl", "
        unet_history = pickle.load(f)
```

```
Epoch 1/50
2025-05-06 07:44:29.011013: E tensorflow/core/gr
appler/optimizers/meta_optimizer.cc:961] layout
failed: INVALID_ARGUMENT: Size of values 0 does
not match size of permutation 4 @ fanin shape in
model_2/spatial_dropout2d_2/dropout/SelectV2-2-T
ransposeNHWCToNCHW-LayoutOptimizer
```

```
704/704 [==============================] - 28s 2
6ms/step - loss: 2.1273 - accuracy: 0.2187 - val
_loss: 1.9670 - val_accuracy: 0.1961 - lr: 0.001
0
Epoch 2/50
704/704 [==============================] - 19s 2
7ms/step - loss: 1.9240 - accuracy: 0.2625 - val
_loss: 2.0445 - val_accuracy: 0.2369 - lr: 0.001
0
Epoch 3/50
704/704 [==============================] - 18s 2
5ms/step - loss: 1.7786 - accuracy: 0.3271 - val
_loss: 1.9004 - val_accuracy: 0.3600 - lr: 0.001
0
Epoch 4/50
704/704 [==============================] - 17s 2
5ms/step - loss: 1.5930 - accuracy: 0.4269 - val
_loss: 1.6302 - val_accuracy: 0.4323 - lr: 0.001
0
Epoch 5/50
704/704 [==============================] - 18s 2
6ms/step - loss: 1.4789 - accuracy: 0.4788 - val
_loss: 1.6558 - val_accuracy: 0.4311 - lr: 0.001
0
Epoch 6/50
704/704 [==============================] - 18s 2
6ms/step - loss: 1.4104 - accuracy: 0.5063 - val
_loss: 1.8784 - val_accuracy: 0.4221 - lr: 0.001
0
Epoch 7/50
704/704 [==============================] - 18s 2
6ms/step - loss: 1.3652 - accuracy: 0.5343 - val
_loss: 1.6177 - val_accuracy: 0.5069 - lr: 0.001
0
Epoch 8/50
704/704 [==============================] - 18s 2
6ms/step - loss: 1.3381 - accuracy: 0.5493 - val
_loss: 1.4370 - val_accuracy: 0.5304 - lr: 0.001
0
Epoch 9/50
704/704 [==============================] - 18s 2
6ms/step - loss: 1.3182 - accuracy: 0.5604 - val
_loss: 1.6795 - val_accuracy: 0.4705 - lr: 0.001
0
Epoch 10/50
704/704 [==============================] - 18s 2
6ms/step - loss: 1.2922 - accuracy: 0.5737 - val
_loss: 1.6071 - val_accuracy: 0.5024 - lr: 0.001
0
Epoch 11/50
704/704 [==============================] - 18s 2
5ms/step - loss: 1.2749 - accuracy: 0.5844 - val
_loss: 1.4434 - val_accuracy: 0.5361 - lr: 0.001
0
Epoch 12/50
704/704 [==============================] - 18s 2
```

```
6ms/step - loss: 1.2565 - accuracy: 0.5932 - val
_loss: 1.4420 - val_accuracy: 0.5462 - lr: 0.001
0
Epoch 13/50
704/704 [==============================] - 18s 2
6ms/step - loss: 1.2315 - accuracy: 0.6028 - val
_loss: 1.3236 - val_accuracy: 0.5722 - lr: 0.001
0
Epoch 14/50
704/704 [==============================] - 18s 2
5ms/step - loss: 1.2160 - accuracy: 0.6099 - val
_loss: 1.6333 - val_accuracy: 0.5158 - lr: 0.001
0
Epoch 15/50
704/704 [==============================] - 17s 2
5ms/step - loss: 1.2031 - accuracy: 0.6212 - val
_loss: 1.4250 - val_accuracy: 0.5682 - lr: 0.001
0
Epoch 16/50
704/704 [==============================] - 17s 2
4ms/step - loss: 1.1800 - accuracy: 0.6318 - val
_loss: 1.4209 - val_accuracy: 0.5576 - lr: 0.001
0
Epoch 17/50
704/704 [==============================] - 18s 2
6ms/step - loss: 1.1625 - accuracy: 0.6376 - val
_loss: 1.4240 - val_accuracy: 0.5665 - lr: 0.001
0
Epoch 18/50
704/704 [==============================] - 17s 2
4ms/step - loss: 1.1461 - accuracy: 0.6462 - val
_loss: 1.4653 - val_accuracy: 0.5675 - lr: 0.001
0
Epoch 19/50
704/704 [==============================] - 18s 2
5ms/step - loss: 1.0305 - accuracy: 0.6902 - val
_loss: 1.3804 - val_accuracy: 0.6016 - lr: 5.000
0e-04
Epoch 20/50
704/704 [==============================] - 17s 2
4ms/step - loss: 0.9763 - accuracy: 0.7089 - val
_loss: 1.5627 - val_accuracy: 0.5701 - lr: 5.000
0e-04
Epoch 21/50
704/704 [==============================] - 19s 2
6ms/step - loss: 0.9313 - accuracy: 0.7277 - val
_loss: 1.3350 - val_accuracy: 0.6203 - lr: 5.000
0e-04
Epoch 22/50
704/704 [==============================] - 19s 2
8ms/step - loss: 0.8973 - accuracy: 0.7394 - val
_loss: 1.3435 - val_accuracy: 0.6077 - lr: 5.000
0e-04
Epoch 23/50
704/704 [==============================] - 17s 2
5ms/step - loss: 0.8530 - accuracy: 0.7568 - val
```

```
_loss: 1.7413 - val_accuracy: 0.5767 - lr: 5.000
0e-04
Epoch 24/50
704/704 [==============================] - 17s 2
5ms/step - loss: 0.7464 - accuracy: 0.7980 - val
_loss: 1.4455 - val_accuracy: 0.6109 - lr: 2.500
0e-04
Epoch 25/50
704/704 [==============================] - 18s 2
5ms/step - loss: 0.6889 - accuracy: 0.8198 - val
_loss: 1.4432 - val_accuracy: 0.6220 - lr: 2.500
0e-04
Epoch 26/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.6495 - accuracy: 0.8353 - val
_loss: 1.5795 - val_accuracy: 0.6269 - lr: 2.500
0e-04
Epoch 27/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.6133 - accuracy: 0.8485 - val
_loss: 1.6417 - val_accuracy: 0.6138 - lr: 2.500
0e-04
Epoch 28/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.5752 - accuracy: 0.8633 - val
_loss: 1.6208 - val_accuracy: 0.6067 - lr: 2.500
0e-04
Epoch 29/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.5072 - accuracy: 0.8915 - val
_loss: 1.6927 - val_accuracy: 0.6201 - lr: 1.250
0e-04
Epoch 30/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.4690 - accuracy: 0.9035 - val
_loss: 1.7511 - val_accuracy: 0.6243 - lr: 1.250
0e-04
Epoch 31/50
704/704 [==============================] - 17s 2
5ms/step - loss: 0.4463 - accuracy: 0.9113 - val
_loss: 1.8810 - val_accuracy: 0.6218 - lr: 1.250
0e-04
Epoch 32/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.4202 - accuracy: 0.9200 - val
_loss: 1.8177 - val_accuracy: 0.6166 - lr: 1.250
0e-04
Epoch 33/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.4038 - accuracy: 0.9256 - val
_loss: 2.0862 - val_accuracy: 0.6049 - lr: 1.250
0e-04
Epoch 34/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.3751 - accuracy: 0.9362 - val
_loss: 1.8563 - val_accuracy: 0.6342 - lr: 6.250
```

```
0e-05
Epoch 35/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.3456 - accuracy: 0.9478 - val
_loss: 1.9577 - val_accuracy: 0.6234 - lr: 6.250
0e-05
Epoch 36/50
704/704 [==============================] - 17s 2
4ms/step - loss: 0.3451 - accuracy: 0.9484 - val
_loss: 2.0141 - val_accuracy: 0.6257 - lr: 6.250
0e-05
Epoch 37/50
704/704 [==============================] - 18s 2
5ms/step - loss: 0.3357 - accuracy: 0.9515 - val
_loss: 1.9588 - val_accuracy: 0.6356 - lr: 6.250
0e-05
Epoch 38/50
704/704 [==============================] - 18s 2
5ms/step - loss: 0.3258 - accuracy: 0.9539 - val
_loss: 2.0652 - val_accuracy: 0.6262 - lr: 6.250
0e-05
Epoch 39/50
704/704 [==============================] - 17s 2
4ms/step - loss: 0.3093 - accuracy: 0.9615 - val
_loss: 2.0176 - val_accuracy: 0.6346 - lr: 3.125
0e-05
Epoch 40/50
704/704 [==============================] - 19s 2
6ms/step - loss: 0.3025 - accuracy: 0.9641 - val
_loss: 2.0383 - val_accuracy: 0.6323 - lr: 3.125
0e-05
Epoch 41/50
704/704 [==============================] - 18s 2
5ms/step - loss: 0.2998 - accuracy: 0.9645 - val
_loss: 2.0208 - val_accuracy: 0.6335 - lr: 3.125
0e-05
Epoch 42/50
704/704 [==============================] - 18s 2
6ms/step - loss: 0.2939 - accuracy: 0.9672 - val
_loss: 2.1075 - val_accuracy: 0.6225 - lr: 3.125
0e-05
Epoch 43/50
704/704 [==============================] - 17s 2
5ms/step - loss: 0.2830 - accuracy: 0.9689 - val
_loss: 2.0608 - val_accuracy: 0.6314 - lr: 3.125
0e-05
Epoch 44/50
704/704 [==============================] - 18s 2
5ms/step - loss: 0.2804 - accuracy: 0.9705 - val
_loss: 2.0550 - val_accuracy: 0.6387 - lr: 1.562
5e-05
Epoch 45/50
704/704 [==============================] - 17s 2
4ms/step - loss: 0.2785 - accuracy: 0.9690 - val
_loss: 2.0758 - val_accuracy: 0.6372 - lr: 1.562
5e-05
```

```
Epoch 46/50
704/704 [==============================] - 17s 2
4ms/step - loss: 0.2756 - accuracy: 0.9719 - val
_loss: 2.0749 - val_accuracy: 0.6356 - lr: 1.562
5e-05
Epoch 47/50
704/704 [==============================] - 17s 2
4ms/step - loss: 0.2740 - accuracy: 0.9730 - val
_loss: 2.0908 - val_accuracy: 0.6368 - lr: 1.562
5e-05
Epoch 48/50
704/704 [==============================] - 17s 2
5ms/step - loss: 0.2716 - accuracy: 0.9724 - val
_loss: 2.1001 - val_accuracy: 0.6368 - lr: 1.562
5e-05
Epoch 49/50
704/704 [==============================] - 17s 2
5ms/step - loss: 0.2669 - accuracy: 0.9749 - val
_loss: 2.0884 - val_accuracy: 0.6405 - lr: 1.000
0e-05
Epoch 50/50
704/704 [==============================] - 17s 2
4ms/step - loss: 0.2630 - accuracy: 0.9766 - val
_loss: 2.0932 - val_accuracy: 0.6351 - lr: 1.000
0e-05
```

```
In [59]: plot_accuracy_and_loss(unet_history, "U-Net Model
```

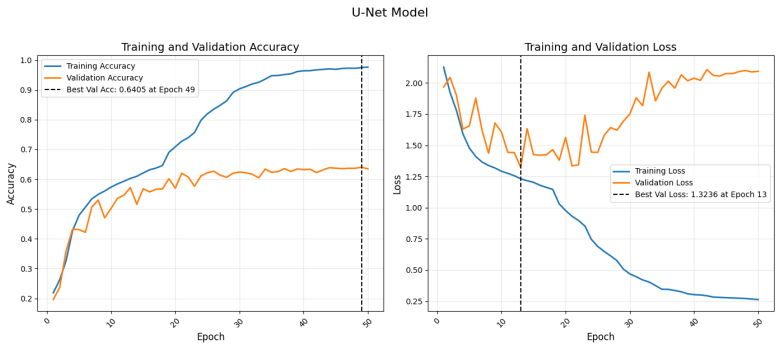


Figure 8. U-Net Model

There is definitely an improvement over our original baseline model, more than a 10% difference in the validation accuracy. This updated model will serve as our new baseline moving forward, as we are confident there is still room for further improvement.

4.3 Baseline Summary

In the above experiment, we implemented two preprocessing methods: rescaling by dividing pixel values by 255 to normalize data within the [0,1] range, and per-image standardization to normalize each image individually, addressing variations in lighting and contrast. Although performance improved with per-image standardization, the gain was not significant enough to justify the added complexity and increased training time. As a result, we shifted to a different approach—building a more complex architecture and training it with augmented data.

Our U-Net architecture significantly outperformed the baseline CNN, achieving 64% accuracy compared to 52%. This improvement validates our strategy of using a more sophisticated architecture with built-in attention mechanisms via skip connections, which help the model focus on emotion-relevant facial features. U-Net strikes a strong balance between complexity and efficiency, capturing multi-scale information while remaining computationally feasible for the size of the FER-2013 dataset.

Looking ahead, we see several promising avenues for further improving model performance. These include experimenting with ensemble methods that combine multiple architectures, implementing cross-validation to better handle FER-2013's class imbalance, exploring transfer learning with pre-trained facial recognition models, and investigating more advanced attention mechanisms such as transformers. Hyperparameter tuning may also yield additional performance gains.

4.4 Baseline Evaluation

We evaluate our two baseline models on the test set. We display classification reports along with confusion matrices to visualize the model's performance across different classes.

In [60]:
```
# create a function to plot confusion matrix
def plot_conf_matrix(cm, title, class_labels):
    plt.figure(figsize=(8, 6))
    sns.heatmap(
        cm,
        annot=True,
        fmt="d",
        cmap="Blues",
        xticklabels=class_labels,
        yticklabels=class_labels,
    )
    plt.title(title)
    plt.xlabel("Predction")
    plt.ylabel("True")
    plt.show()
```

In [61]:
```
# pass data for model evaluation
# classfication_report will return metrics : pr
# confusion matrix return true positive, true n


def evaluate_model(model, dataset, model_name='
    y_true = []
    y_pred = []

    for x_batch, y_batch in dataset:
        preds = model.predict(x_batch, verbose=
        y_true.extend(np.argmax(y_batch.numpy()
        y_pred.extend(np.argmax(preds, axis=1))

    target_names = list(class_labels.values())
    print(f"{model_name} Classification report'
    print(classification_report(y_true, y_pred,

    print(f"{model_name} Confusion matrix")
    cm = confusion_matrix(y_true, y_pred)
    return y_true, y_pred, cm
```

In [62]:
```
y_true_base, y_pred_base, cm_base = evaluate_mo
    baseline_model, test_norm, "Simple CNN"
)
plot_conf_matrix(cm_base, "Simple CNN confusion
```

```
Simple CNN Classification report
              precision    recall  f1-score
support

       angry       0.43      0.25      0.32
958
     disgust       1.00      0.02      0.04
111
        fear       0.33      0.17      0.23
1024
       happy       0.70      0.76      0.73
1774
     neutral       0.42      0.59      0.49
1233
         sad       0.36      0.49      0.41
1247
    surprise       0.69      0.62      0.65
831

    accuracy                           0.50
7178
   macro avg       0.56      0.41      0.41
7178
weighted avg       0.51      0.50      0.49
7178
```

Simple CNN Confusion matrix

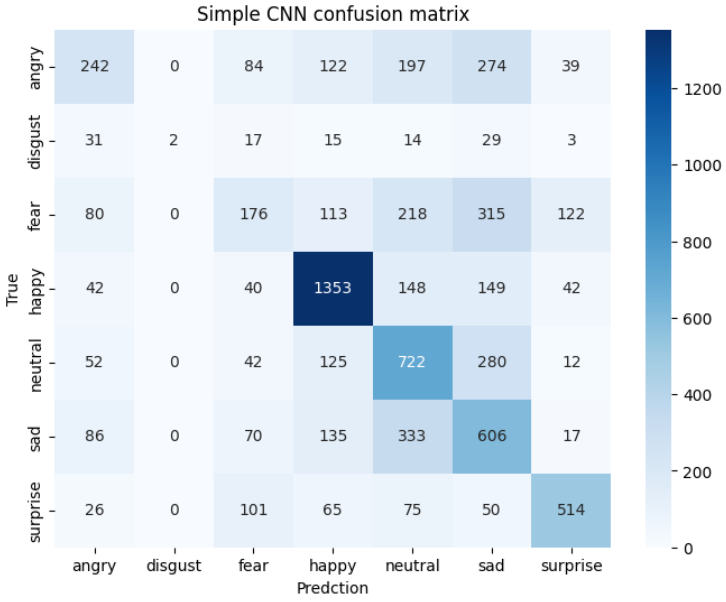


Figure 9. Simple CNN Confusion Matrix

```
In [63]: baseline_model.evaluate(test_norm, verbose=1)
```

```
225/225 [==============================] - 1s
3ms/step - loss: 1.2940 - accuracy: 0.5036
```

```
Out[63]: [1.293998122215271, 0.5036221742630005]
```

The test accuracy of the baseline model is around 50%. This is better than random guessing but not up to par with human performance. The classifcation report shows the model struggles with disgust and fear based on the f1 score. This makes sense because disgust is underrepresented in the dataset and fear is often confused with surprise and anger, as shown by the confusion matrix. The model performs well on happy and surprise, with f1 scores greater than 0.6.

In [64]:
```
y_true_unet, y_pred_unet, cm_unet = evaluate_
plot_conf_matrix(cm_unet, "U-Net confusion ma
```

```
U-Net Classification report
              precision    recall  f1-score
support

       angry       0.60      0.58      0.59
958
     disgust       0.57      0.50      0.53
111
        fear       0.47      0.49      0.48
1024
       happy       0.85      0.84      0.85
1774
     neutral       0.59      0.63      0.61
1233
         sad       0.54      0.49      0.51
1247
    surprise       0.75      0.78      0.77
831

    accuracy                           0.65
7178
   macro avg       0.62      0.62      0.62
7178
weighted avg       0.65      0.65      0.65
7178

U-Net Confusion matrix
```

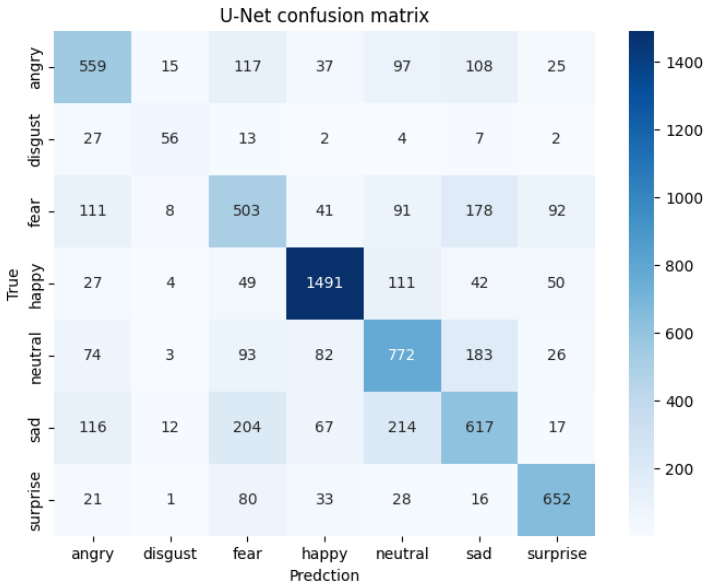


Figure 10. U-Net Confusion Matrix

```
In [65]: unet_model.evaluate(test_norm, verbose=1)
```

```
225/225 [==============================] - 2s 8ms/step - loss: 2.0315 - accuracy: 0.6478
```

```
Out[65]: [2.0315184593200684, 0.6478127837181091]
```

Surprisingly, “fear” and “sad” have the lowest f1 score for the baseline u-net model. This is in line with our other models downstream. The two emotions are often not mutually exclusive and look similar. This is why the model confuses them. Looking at the confusion matrix, the model has trouble determining if “fear” is “anger” or “sad”. As for “sad”, the model confuses it with “anger”, “fear”, and “neutral”. This is likely due to the fact that “sad” is often confused with “neutral”, as they are both low arousal emotions. The model does best on “happy” and “surprise”. Both of these emotions are high arousal and have clear facial features.

To overcome the limitation, we moved on to U-net model. Based on the classification report and confusion matrix, the U_Net model achieved an overall accuracy of 65% and a macro average F1 score of 0.62. The model performed particularly well on the “happy” and “surprise” classes, with F1 scores of 0.85 and 0.77, respectively. It also showed solid performance on “angry” (0.59), “neutral” (0.61), and “sad” (0.51), while “fear” (0.48) and “disgust” (0.53) remained as the most challenging classes.

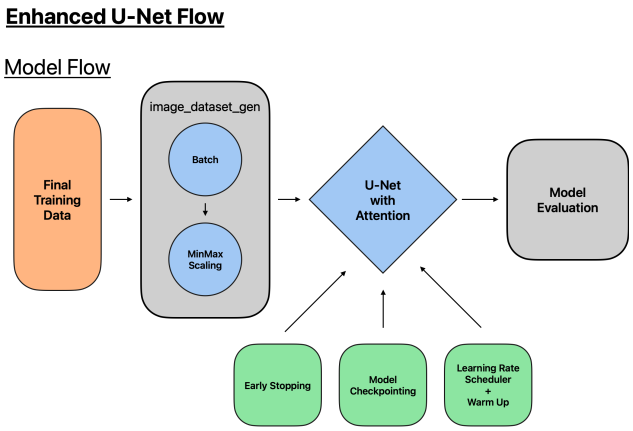
Specifically, the model correctly classified 1,491 out of 1,774 “happy” images (recall 0.84, precision 0.85), highlighting its strength in recognizing positive emotions with distinctive facial cues. The “surprise” class was another high performer, with 652/831 correct predictions (recall 0.78, precision 0.75). For negative emotions, the U-Net predicted “angry” with decent balance (precision 0.60, recall 0.58), and “fear” with moderate success (503/1,024, recall 0.49, precision 0.47). Notably, the “disgust” class, despite its small representation (111 samples), saw improvement with recall of 0.50, reflecting the model’s enhanced sensitivity compared to previous models. While precision dropped from 1.0 to 0.57, this was mainly because the previous model rarely predicted “disgust” at all, making this increase in sensitivity a meaningful gain.

In summary, the strong performance on dominant classes like “happy” and “surprise” can be attributed to their distinctive features and larger sample sizes. The moderate to low performance on “fear” and “disgust” likely stems from their visual overlap with other negative emotions and limited training examples.

5. Final Model

5.1 Enhanced U-Net

In [11]:
```
git_tag = "https://raw.githubusercontent.com
img_url = f"{git_tag}cs109b-final-project/ma
display(Image(url=img_url, width=650, height
```



Flow Diagram 4. Enhanced U-Net Model Flow

The flow of the enhanced U-Net model is shown above. The process is similar to the baseline U-Net, but we added two new components: squeeze-excite blocks and class weighting.

Squeeze-excite blocks work by helping the model focus more on the most important features, adjusting the strength of each channel based on how useful it is for the task. Since recognizing emotions often depends on picking up subtle cues in facial expressions, we expected this to help the model be more selective and accurate. We also added class weighting to deal with the natural imbalance in the FER2013 dataset, making sure that less common emotions like “disgust” or “surprise” weren’t overlooked during training. Together, these changes were aimed at helping the model better capture important details and perform more consistently across all classes.

In practice, though, the enhanced U-Net didn’t perform as well, reaching 64% accuracy compared to 76% for the baseline. One possible reason is that the squeeze-excite blocks may have put too much emphasis on certain features while downplaying others that were still important for distinguishing between similar emotions. With a dataset as noisy and low-resolution as FER2013, this extra complexity might have made it harder for the model to generalize. Class weighting could have also shifted the learning process too much toward minority classes, disrupting the model’s overall balance. While the enhancements were promising in theory, they ended up making it harder for the model to handle the messy real-world data in this case.

In [67]:

```
def create_enhanced_unet_model():

    # Slightly increased regularization
    reg_strength = 2e-4

    # input layer - 48x48 grayscale
    inputs = Input(shape=(48, 48, 1))

    # --- encoder path with residual connec

    # first block - level 1
    conv1 = Conv2D(
        64, kernel_size=(3, 3), padding="sa
    )(inputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = Activation("relu")(conv1)
    conv1_res = conv1

    conv1 = Conv2D(
        64, kernel_size=(3, 3), padding="sa
    )(conv1)
    conv1 = BatchNormalization()(conv1)
    conv1 = Activation("relu")(conv1)

    # residual connection
    conv1 = Add()([conv1, conv1_res])

    pool1 = MaxPooling2D(pool_size=(2, 2))(
    pool1 = SpatialDropout2D(0.15)(pool1)

    # second block - level 2
    conv2 = Conv2D(
        128, kernel_size=(3, 3), padding="s
    )(pool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = Activation("relu")(conv2)
    conv2_res = conv2

    conv2 = Conv2D(
        128, kernel_size=(3, 3), padding="s
    )(conv2)
    conv2 = BatchNormalization()(conv2)
    conv2 = Activation("relu")(conv2)

    # residual connection
    conv2 = Add()([conv2, conv2_res])

    pool2 = MaxPooling2D(pool_size=(2, 2))(
    pool2 = SpatialDropout2D(0.25)(pool2)

    # third block - bottleneck with dilated
    conv3 = Conv2D(
        256,
        kernel_size=(3, 3),
        padding="same",
        dilation_rate=(2, 2),
```

```
    kernel_regularizer=l2(reg_strength)
)(pool2)
conv3 = BatchNormalization()(conv3)
conv3 = Activation("relu")(conv3)
conv3_res = conv3

conv3 = Conv2D(
    256, kernel_size=(3, 3), padding="s
)(conv3)
conv3 = BatchNormalization()(conv3)
conv3 = Activation("relu")(conv3)

# residual connection
conv3 = Add()([conv3, conv3_res])

# --- Squeeze-Excite attention at bottl
se_channels = 256
se_global_pool = GlobalAveragePooling2D
se_dense_1 = Dense(se_channels // 16, a
se_dense_2 = Dense(se_channels, activat
se_reshape = Reshape((1, 1, se_channels
conv3 = Multiply()([conv3, se_reshape])

# --- decoder path with spatial attenti

# level 2 upsampling
up2 = UpSampling2D(size=(2, 2))(conv3)
up2 = Conv2D(
    128, kernel_size=(3, 3), padding="s
)(up2)
up2 = BatchNormalization()(up2)
up2 = Activation("relu")(up2)

# simplified attention for conv2
avg_pool_2 = Conv2D(1, kernel_size=1)(C
max_pool_2 = Conv2D(1, kernel_size=1)(C
attn2 = Concatenate()([avg_pool_2, max_
attn2 = Conv2D(1, kernel_size=(7, 7), p
attended_x2 = Multiply()([conv2, attn2]

# combine upsampled features with atter
merge2 = Concatenate()([up2, attended_x
conv2_up = Conv2D(
    128, kernel_size=(3, 3), padding="s
)(merge2)
conv2_up = BatchNormalization()(conv2_u
conv2_up = Activation("relu")(conv2_up)

# level 1 upsampling
up1 = UpSampling2D(size=(2, 2))(conv2_u
up1 = Conv2D(
    64, kernel_size=(3, 3), padding="sa
)(up1)
up1 = BatchNormalization()(up1)
up1 = Activation("relu")(up1)
```

```
    # simplified attention for conv1
    avg_pool_1 = Conv2D(1, kernel_size=1)(C
    max_pool_1 = Conv2D(1, kernel_size=1)(C
    attn1 = Concatenate()([avg_pool_1, max_
    attn1 = Conv2D(1, kernel_size=(7, 7), p
    attended_x1 = Multiply()([conv1, attn1]

    # combine upsampled features with atter
    merge1 = Concatenate()([up1, attended_x
    conv1_up = Conv2D(
        64, kernel_size=(3, 3), padding="sa
    )(merge1)
    conv1_up = BatchNormalization()(conv1_u
    conv1_up = Activation("relu")(conv1_up)

    # --- feature fusion ---

    # extract features from multiple levels
    feat_lvl3 = GlobalAveragePooling2D()(co
    feat_lvl2 = GlobalAveragePooling2D()(co
    feat_lvl1 = GlobalAveragePooling2D()(co

    # extract max features too
    max_feat_lvl3 = GlobalMaxPooling2D()(co
    max_feat_lvl2 = GlobalMaxPooling2D()(co
    max_feat_lvl1 = GlobalMaxPooling2D()(co

    # concatenate avg and max features
    feat_lvl3 = Concatenate()([feat_lvl3, m
    feat_lvl2 = Concatenate()([feat_lvl2, m
    feat_lvl1 = Concatenate()([feat_lvl1, m

    # simply concatenate all features
    combined_features = Concatenate()([feat
    combined_features = Dropout(0.3)(
        combined_features
    )  # Added dropout before classificatio

    # --- classification head with improved
    x = Dense(256, kernel_regularizer=l2(re
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    x = Dropout(0.6)(x)  # Increased dropou

    x = Dense(128, kernel_regularizer=l2(re
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    x = Dropout(0.5)(x)  # Increased dropou

    # output for 7 emotions
    outputs = Dense(7, activation="softmax"

    # create model
    model = Model(inputs=inputs, outputs=ou

    model.compile(
        optimizer=Adam(learning_rate=1e-4,
```

```
        loss=tf.keras.losses.CategoricalCro
        metrics=["accuracy", tf.keras.metri
    )

    return model
```

In [68]:

```
# train with improved model
if not os.path.exists("models/enhanced_unet
    # create enhanced model
    enhanced_model = create_enhanced_unet_m

    # define callbacks
    callbacks = [
        tf.keras.callbacks.EarlyStopping(
            monitor="val_accuracy",
            patience=10,
            restore_best_weights=True,
            min_delta=0.01,
        ),
        tf.keras.callbacks.ReduceLROnPlatea
            monitor="val_accuracy", factor=
        ),
        tf.keras.callbacks.ModelCheckpoint(
            "models/enhanced_unet_model.ker
            monitor="val_accuracy",
            save_best_only=True,
        ),
        # learning rate warmup schedule
        tf.keras.callbacks.LearningRateSche
            lambda epoch, lr: min(2e-4, lr
        ),
    ]

    enhanced_history = enhanced_model.fit(
        train_norm,
        validation_data=val_norm,
        epochs=80,
        class_weight=class_weights_dict,  #
        callbacks=callbacks,
    )

    with open("model_history/enhanced_unet_
        pickle.dump(enhanced_history, f)
else:
    print("Enhanced model file already exis
    enhanced_model = tf.keras.models.load_m

    try:
        with open("model_history/enhanced_u
            enhanced_history = pickle.load(
    except:
        print("History file not found.")
```

```
Epoch 1/80
```

```
2025-05-06 08:00:36.761615: E tensorflow/c
ore/grappler/optimizers/meta_optimizer.c
c:961] layout failed: INVALID_ARGUMENT: Si
ze of values 0 does not match size of perm
utation 4 @ fanin shape inmodel_3/spatial_
dropout2d_4/dropout/SelectV2-2-TransposeNH
WCToNCHW-LayoutOptimizer
```

```
704/704 [==============================] -
43s 41ms/step - loss: 2.7712 - accuracy: 0
.1615 - f1_score: 0.1688 - val_loss: 2.325
2 - val_accuracy: 0.1021 - val_f1_score: 0
.0869 - lr: 1.4000e-04
Epoch 2/80
704/704 [==============================] -
29s 41ms/step - loss: 2.5478 - accuracy: 0
.1697 - f1_score: 0.1751 - val_loss: 2.296
5 - val_accuracy: 0.1211 - val_f1_score: 0
.0816 - lr: 1.8000e-04
Epoch 3/80
704/704 [==============================] -
29s 41ms/step - loss: 2.4008 - accuracy: 0
.1771 - f1_score: 0.1802 - val_loss: 2.224
8 - val_accuracy: 0.2242 - val_f1_score: 0
.2047 - lr: 2.0000e-04
Epoch 4/80
704/704 [==============================] -
29s 41ms/step - loss: 2.3185 - accuracy: 0
.1886 - f1_score: 0.1891 - val_loss: 2.197
3 - val_accuracy: 0.1716 - val_f1_score: 0
.1369 - lr: 2.0000e-04
Epoch 5/80
704/704 [==============================] -
28s 40ms/step - loss: 2.2378 - accuracy: 0
.2067 - f1_score: 0.2042 - val_loss: 2.241
9 - val_accuracy: 0.1684 - val_f1_score: 0
.1246 - lr: 2.0000e-04
Epoch 6/80
704/704 [==============================] -
29s 41ms/step - loss: 2.1492 - accuracy: 0
.2245 - f1_score: 0.2200 - val_loss: 2.044
1 - val_accuracy: 0.2975 - val_f1_score: 0
.2585 - lr: 2.0000e-04
Epoch 7/80
704/704 [==============================] -
28s 39ms/step - loss: 2.0977 - accuracy: 0
.2458 - f1_score: 0.2407 - val_loss: 2.026
8 - val_accuracy: 0.2315 - val_f1_score: 0
.1907 - lr: 2.0000e-04
Epoch 8/80
704/704 [==============================] -
29s 41ms/step - loss: 2.0436 - accuracy: 0
.2644 - f1_score: 0.2581 - val_loss: 1.942
0 - val_accuracy: 0.3132 - val_f1_score: 0
.2542 - lr: 2.0000e-04
Epoch 9/80
704/704 [==============================] -
28s 40ms/step - loss: 1.9817 - accuracy: 0
.2967 - f1_score: 0.2872 - val_loss: 1.805
3 - val_accuracy: 0.3578 - val_f1_score: 0
.2766 - lr: 2.0000e-04
Epoch 10/80
704/704 [==============================] -
28s 40ms/step - loss: 1.9186 - accuracy: 0
```

```
.3222 - f1_score: 0.3141 - val_loss: 1.708
4 - val_accuracy: 0.3987 - val_f1_score: 0
.3379 - lr: 2.0000e-04
Epoch 11/80
704/704 [==============================] -
28s 39ms/step - loss: 1.8440 - accuracy: 0
.3665 - f1_score: 0.3655 - val_loss: 1.639
9 - val_accuracy: 0.4294 - val_f1_score: 0
.3991 - lr: 2.0000e-04
Epoch 12/80
704/704 [==============================] -
30s 42ms/step - loss: 1.7600 - accuracy: 0
.3960 - f1_score: 0.3952 - val_loss: 1.581
2 - val_accuracy: 0.4395 - val_f1_score: 0
.4160 - lr: 2.0000e-04
Epoch 13/80
704/704 [==============================] -
30s 42ms/step - loss: 1.6864 - accuracy: 0
.4320 - f1_score: 0.4283 - val_loss: 1.505
8 - val_accuracy: 0.4665 - val_f1_score: 0
.3935 - lr: 2.0000e-04
Epoch 14/80
704/704 [==============================] -
29s 42ms/step - loss: 1.6429 - accuracy: 0
.4469 - f1_score: 0.4412 - val_loss: 1.491
9 - val_accuracy: 0.4888 - val_f1_score: 0
.4786 - lr: 2.0000e-04
Epoch 15/80
704/704 [==============================] -
29s 41ms/step - loss: 1.5916 - accuracy: 0
.4681 - f1_score: 0.4592 - val_loss: 1.434
6 - val_accuracy: 0.5050 - val_f1_score: 0
.4572 - lr: 2.0000e-04
Epoch 16/80
704/704 [==============================] -
29s 41ms/step - loss: 1.5548 - accuracy: 0
.4769 - f1_score: 0.4664 - val_loss: 1.426
2 - val_accuracy: 0.5057 - val_f1_score: 0
.4611 - lr: 2.0000e-04
Epoch 17/80
704/704 [==============================] -
29s 40ms/step - loss: 1.5144 - accuracy: 0
.4896 - f1_score: 0.4786 - val_loss: 1.435
0 - val_accuracy: 0.5015 - val_f1_score: 0
.4795 - lr: 2.0000e-04
Epoch 18/80
704/704 [==============================] -
28s 40ms/step - loss: 1.4790 - accuracy: 0
.5027 - f1_score: 0.4908 - val_loss: 1.461
5 - val_accuracy: 0.4874 - val_f1_score: 0
.4573 - lr: 2.0000e-04
Epoch 19/80
704/704 [==============================] -
29s 40ms/step - loss: 1.4729 - accuracy: 0
.5062 - f1_score: 0.4948 - val_loss: 1.395
1 - val_accuracy: 0.5184 - val_f1_score: 0
```

```
.4831 - lr: 2.0000e-04
Epoch 20/80
704/704 [==============================] -
28s 40ms/step - loss: 1.4421 - accuracy: 0
.5160 - f1_score: 0.5045 - val_loss: 1.385
9 - val_accuracy: 0.5196 - val_f1_score: 0
.4860 - lr: 2.0000e-04
Epoch 21/80
704/704 [==============================] -
29s 41ms/step - loss: 1.4106 - accuracy: 0
.5265 - f1_score: 0.5168 - val_loss: 1.403
2 - val_accuracy: 0.5250 - val_f1_score: 0
.5253 - lr: 2.0000e-04
Epoch 22/80
704/704 [==============================] -
30s 42ms/step - loss: 1.3943 - accuracy: 0
.5369 - f1_score: 0.5277 - val_loss: 1.337
6 - val_accuracy: 0.5499 - val_f1_score: 0
.5283 - lr: 2.0000e-04
Epoch 23/80
704/704 [==============================] -
28s 40ms/step - loss: 1.3586 - accuracy: 0
.5421 - f1_score: 0.5314 - val_loss: 1.325
9 - val_accuracy: 0.5543 - val_f1_score: 0
.5386 - lr: 2.0000e-04
Epoch 24/80
704/704 [==============================] -
29s 41ms/step - loss: 1.3362 - accuracy: 0
.5523 - f1_score: 0.5438 - val_loss: 1.326
6 - val_accuracy: 0.5611 - val_f1_score: 0
.5518 - lr: 2.0000e-04
Epoch 25/80
704/704 [==============================] -
28s 40ms/step - loss: 1.3238 - accuracy: 0
.5636 - f1_score: 0.5548 - val_loss: 1.349
0 - val_accuracy: 0.5516 - val_f1_score: 0
.5205 - lr: 2.0000e-04
Epoch 26/80
704/704 [==============================] -
28s 40ms/step - loss: 1.3167 - accuracy: 0
.5699 - f1_score: 0.5614 - val_loss: 1.286
9 - val_accuracy: 0.5727 - val_f1_score: 0
.5516 - lr: 2.0000e-04
Epoch 27/80
704/704 [==============================] -
29s 41ms/step - loss: 1.2721 - accuracy: 0
.5778 - f1_score: 0.5693 - val_loss: 1.352
0 - val_accuracy: 0.5684 - val_f1_score: 0
.5652 - lr: 2.0000e-04
Epoch 28/80
704/704 [==============================] -
29s 41ms/step - loss: 1.2448 - accuracy: 0
.5847 - f1_score: 0.5770 - val_loss: 1.289
7 - val_accuracy: 0.5842 - val_f1_score: 0
.5557 - lr: 2.0000e-04
Epoch 29/80
```

```
704/704 [==============================] -
29s 41ms/step - loss: 1.2331 - accuracy: 0
.5965 - f1_score: 0.5892 - val_loss: 1.273
1 - val_accuracy: 0.5915 - val_f1_score: 0
.5694 - lr: 2.0000e-04
Epoch 30/80
704/704 [==============================] -
28s 40ms/step - loss: 1.2023 - accuracy: 0
.6094 - f1_score: 0.6038 - val_loss: 1.318
3 - val_accuracy: 0.5797 - val_f1_score: 0
.5505 - lr: 2.0000e-04
Epoch 31/80
704/704 [==============================] -
29s 41ms/step - loss: 1.1814 - accuracy: 0
.6137 - f1_score: 0.6077 - val_loss: 1.257
9 - val_accuracy: 0.5950 - val_f1_score: 0
.5794 - lr: 2.0000e-04
Epoch 32/80
704/704 [==============================] -
28s 39ms/step - loss: 1.1590 - accuracy: 0
.6216 - f1_score: 0.6165 - val_loss: 1.284
1 - val_accuracy: 0.5882 - val_f1_score: 0
.5744 - lr: 2.0000e-04
Epoch 33/80
704/704 [==============================] -
29s 41ms/step - loss: 1.1526 - accuracy: 0
.6291 - f1_score: 0.6245 - val_loss: 1.297
9 - val_accuracy: 0.5847 - val_f1_score: 0
.5719 - lr: 2.0000e-04
Epoch 34/80
704/704 [==============================] -
29s 41ms/step - loss: 1.1218 - accuracy: 0
.6315 - f1_score: 0.6269 - val_loss: 1.251
5 - val_accuracy: 0.6002 - val_f1_score: 0
.5926 - lr: 2.0000e-04
Epoch 35/80
704/704 [==============================] -
29s 41ms/step - loss: 1.0846 - accuracy: 0
.6484 - f1_score: 0.6445 - val_loss: 1.262
9 - val_accuracy: 0.6067 - val_f1_score: 0
.5854 - lr: 2.0000e-04
Epoch 36/80
704/704 [==============================] -
28s 40ms/step - loss: 1.0748 - accuracy: 0
.6574 - f1_score: 0.6539 - val_loss: 1.295
6 - val_accuracy: 0.5945 - val_f1_score: 0
.5910 - lr: 2.0000e-04
Epoch 37/80
704/704 [==============================] -
28s 40ms/step - loss: 1.0542 - accuracy: 0
.6617 - f1_score: 0.6583 - val_loss: 1.345
5 - val_accuracy: 0.5837 - val_f1_score: 0
.5652 - lr: 2.0000e-04
Epoch 38/80
704/704 [==============================] -
28s 40ms/step - loss: 1.0339 - accuracy: 0
```

```
.6675 - f1_score: 0.6641 - val_loss: 1.263
9 - val_accuracy: 0.6112 - val_f1_score: 0
.5984 - lr: 2.0000e-04
Epoch 39/80
704/704 [==============================] -
29s 41ms/step - loss: 1.0142 - accuracy: 0
.6768 - f1_score: 0.6739 - val_loss: 1.315
3 - val_accuracy: 0.5962 - val_f1_score: 0
.5779 - lr: 2.0000e-04
Epoch 40/80
704/704 [==============================] -
28s 40ms/step - loss: 0.9847 - accuracy: 0
.6824 - f1_score: 0.6800 - val_loss: 1.286
1 - val_accuracy: 0.6046 - val_f1_score: 0
.5994 - lr: 2.0000e-04
Epoch 41/80
704/704 [==============================] -
29s 41ms/step - loss: 0.9717 - accuracy: 0
.6937 - f1_score: 0.6915 - val_loss: 1.292
6 - val_accuracy: 0.6130 - val_f1_score: 0
.6016 - lr: 2.0000e-04
Epoch 42/80
704/704 [==============================] -
29s 41ms/step - loss: 0.9617 - accuracy: 0
.7032 - f1_score: 0.7015 - val_loss: 1.302
0 - val_accuracy: 0.6149 - val_f1_score: 0
.6062 - lr: 2.0000e-04
Epoch 43/80
704/704 [==============================] -
28s 40ms/step - loss: 0.9374 - accuracy: 0
.7120 - f1_score: 0.7101 - val_loss: 1.333
5 - val_accuracy: 0.6034 - val_f1_score: 0
.6034 - lr: 2.0000e-04
Epoch 44/80
704/704 [==============================] -
29s 41ms/step - loss: 0.9284 - accuracy: 0
.7183 - f1_score: 0.7170 - val_loss: 1.345
7 - val_accuracy: 0.6022 - val_f1_score: 0
.5887 - lr: 2.0000e-04
Epoch 45/80
704/704 [==============================] -
28s 40ms/step - loss: 0.9053 - accuracy: 0
.7274 - f1_score: 0.7262 - val_loss: 1.320
3 - val_accuracy: 0.6154 - val_f1_score: 0
.6153 - lr: 2.0000e-04
```

In [69]:

```
plot_accuracy_and_loss(
    enhanced_history,
    "Enhanced UNet Model with Sqeeze-Excite
)
```

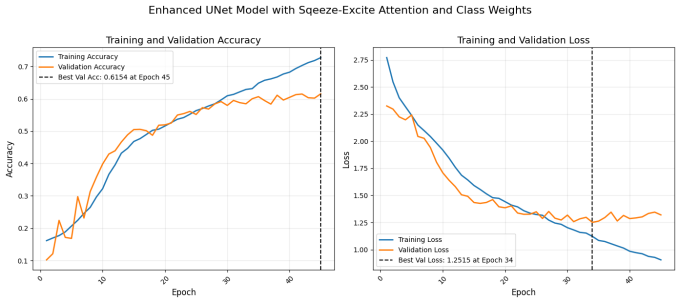


Figure 11. Enhanced UNet Model with squeeze-Excite Attention and Class Weights

The model plateaus around 30 epochs. The validation accuracy is lower than the baseline, which is unexpected. The loss has yet to increase, which is a good sign that we stopped the model before it can overfit.

In [70]:
```
y_true_unet_enh, y_pred_unet_enh, cm_unet_
    enhanced_model, test_norm, "Enhanced U
)

plot_conf_matrix(cm_unet_enh, "Enhanced U-
```

```
Enhanced U-Net Classification report
              precision    recall  f1-sco
re   support

       angry       0.55      0.50      0.
52       958
     disgust       0.46      0.62      0.
53       111
        fear       0.49      0.17      0.
26      1024
       happy       0.78      0.86      0.
82      1774
     neutral       0.49      0.70      0.
57      1233
         sad       0.48      0.41      0.
45      1247
    surprise       0.67      0.82      0.
74       831

    accuracy                           0.
60      7178
   macro avg       0.56      0.58      0.
56      7178
weighted avg       0.59      0.60      0.
58      7178

Enhanced U-Net Confusion matrix
```

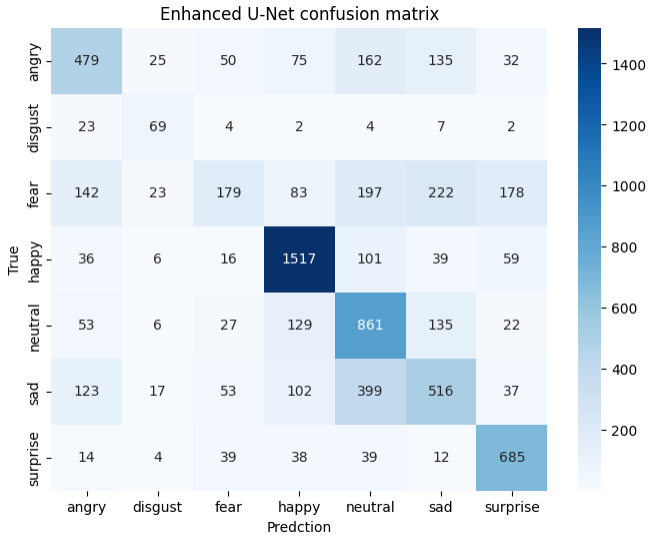


Figure 12. Enhanced UNet Confusion Matrix

In [71]:
```
# Test Set Evaluation
enhanced_model.evaluate(test_norm, verbo
```

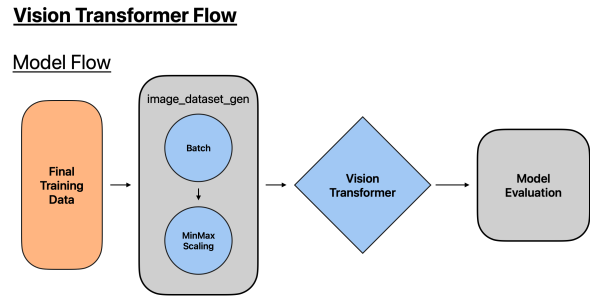
```
225/225 [==============================]
- 2s 9ms/step - loss: 1.2558 - accuracy:
0.5999 - f1_score: 0.5785
```

Out[71]:
```
[1.25584077835083, 0.5998885631561279,
0.5785389542579651]
```

Overall, the enhanced u-net didn't perform much better than the baseline u-net model. The test accuracy is 60%. The class weights must have been a detriment as shown in other papers (Khanzada et al., 2020). The same pattern from prior models holds. For “sad”, the model can't distinguish between “anger” and “neutral”. For “fear”, the model confuses it with “anger”, “sad”, and “neutral”. The “disgust” category seems to do fine.

5.2 Vision Transformer

In [12]:
```
git_tag = "https://raw.githubusercontent
img_url = f"{git_tag}cs109b-final-projec
display(Image(url=img_url, width=650, he
```



Flow Diagram 5. Vision Transformer Model Flow

We use the VisionTransformer (ViT) architecture, which has shown promising results in various image classification tasks (Dosovitskiy et al., 2020). The design is identical to the one used in HW5. The ViT model was trained for 20 epochs and reached a best validation accuracy of 30% at epoch 13, as shown in Figure 13. While the training and validation loss steadily decreased, the accuracy curves exhibited strong fluctuations, reflecting instability and overfitting due to limited data.

The classification report and confusion matrix (Figure 14) further revealed that the ViT struggled to generalize, with a weighted average F1-score of 0.23 and macro average F1-score of 0.18. The model heavily over-predicted the happy class, while classes like angry and disgust were rarely identified correctly. This underperformance is most likely due to the data hungry nature of transformers. Unlike CNNs, ViTs have weaker inductive biases (i.e. no built in translation invariance), requiring substantially larger datasets to learn meaningful patterns.

Despite hyperparameter tuning and regularization efforts, the ViT could not match the performance of the baseline CNN or the transfer learning approach. These results emphasize that while transformers have shown great success in large scale computer vision tasks, they are not well suited for small datasets without extensive augmentation or pretraining.

In [73]:
```
MODEL_WEIGHTS_PATH = "./models/"
MODEL_HISTORY_PATH = "./model_history/"
```

In [74]:

```
target_size = (48, 48)
batch_size = 32

baseline_train_datagen = image_dataset_
    DATA_DIR + "/clean_train",
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)

baseline_val_datagen = image_dataset_fr
    val_dir,
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)


def minmax_norm(x, y):
    return x / 255.0, y


train_norm = baseline_train_datagen.map
    minmax_norm, num_parallel_calls=tf.
).prefetch(buffer_size=tf.data.AUTOTUNE

val_norm = baseline_val_datagen.map(
    minmax_norm, num_parallel_calls=tf.
).prefetch(buffer_size=tf.data.AUTOTUNE
```

```
Found 22508 files belonging to 7 class
es.
Found 5741 files belonging to 7 classe
s.
```

In [75]:
```python
# given config
num_classes = 7
patch_size = 8
num_heads = 8
num_blocks = 4
embed_dim = 64
attention_dim = 64
feedforward_dim = 128
input_shape = (48, 48, 1)

# Create a VisionTransformer
vit = VisionTransformer(
    num_classes=num_classes,
    patch_size=patch_size,
    num_heads=num_heads,
    num_blocks=num_blocks,
    embed_dim=embed_dim,
    attention_dim=attention_dim,
    feedforward_dim=feedforward_dim,
    input_size=input_shape,
)

# build method with an input shape of (
vit.build(input_shape=(None, *input_sha

# show a summary
vit.summary()
```

```
Model: "vision_transformer"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 patch_embedding (PatchEmbe  multiple                  4160
 dding)

 embedding (Embedding)       multiple                  2304

 transformer_encoder (Trans  multiple                  598016
 formerEncoder)

 sequential_5 (Sequential)   (None, 7)                 583

=================================================================
Total params: 605127 (2.31 MB)
Trainable params: 605127 (2.31 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [76]:
```
vit.compile(
    optimizer=tf.keras.optimizers.Adam(
    loss=tf.keras.losses.CategoricalCro
    metrics=["accuracy"],
)
```

In [77]:
```
if os.path.exists(os.path.join(MODEL_WE
    print("Loading pre-trained model we
    vit = tf.keras.models.load_model(
        os.path.join(MODEL_WEIGHTS_PATH
    )

    vit.compile(
        optimizer=tf.keras.optimizers.A
        loss=tf.keras.losses.Categorica
        metrics=["accuracy"],
    )
    with open(os.path.join(MODEL_HISTOR
        vit_history = pickle.load(f)
else:
    print("No pre-trained model found.
    vision_history = vit.fit(
        train_norm,
        validation_data=val_norm,
        epochs=20,
        verbose=1,
    )

    vit.save(os.path.join(MODEL_WEIGHTS

    with open(os.path.join(MODEL_HISTOR
        pickle.dump(vision_history, f)
```

```
Loading pre-trained model weights.
```

In [78]:
```
plot_accuracy_and_loss(
    vit_history,
    "Vision Transformer Model",
)
```

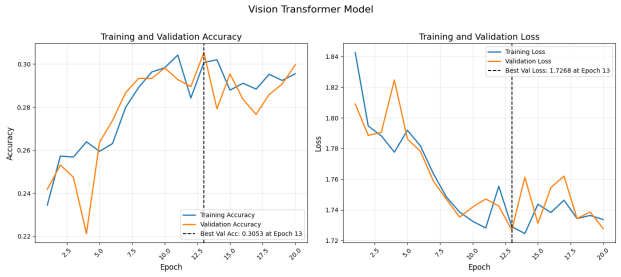


Figure 13. Vision Transformer Model

In [79]:
```
y_true_vit, y_pred_vit, cm_vit = evalu

plot_conf_matrix(cm_vit, "Vision Trans
```

```
Vision Transformer Classification rep
ort
              precision    recall  f
1-score   support

       angry       0.00      0.00
0.00       958
     disgust       0.00      0.00
0.00       111
        fear       0.22      0.03
0.05      1024
       happy       0.32      0.71
0.44      1774
     neutral       0.21      0.05
0.09      1233
         sad       0.25      0.42
0.31      1247
    surprise       0.44      0.34
0.38       831

    accuracy
0.30      7178
   macro avg       0.21      0.22
0.18      7178
weighted avg       0.24      0.30
0.23      7178

Vision Transformer Confusion matrix
```

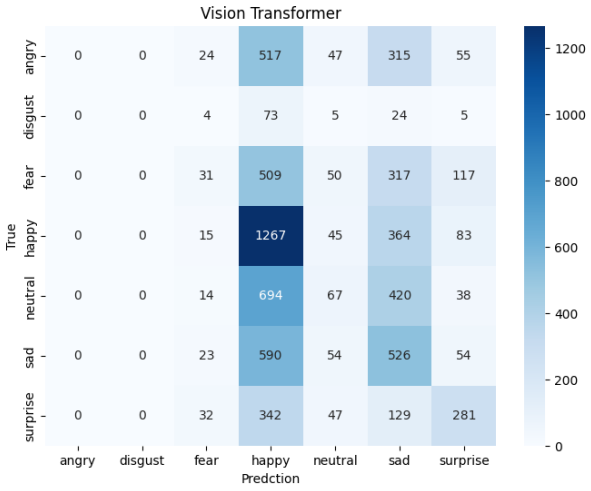


Figure 14. Vision Transformer Confusion Matrix

In [80]:
```
# Test Set Evaluation
vit.evaluate(test_norm, verbose=1)
```

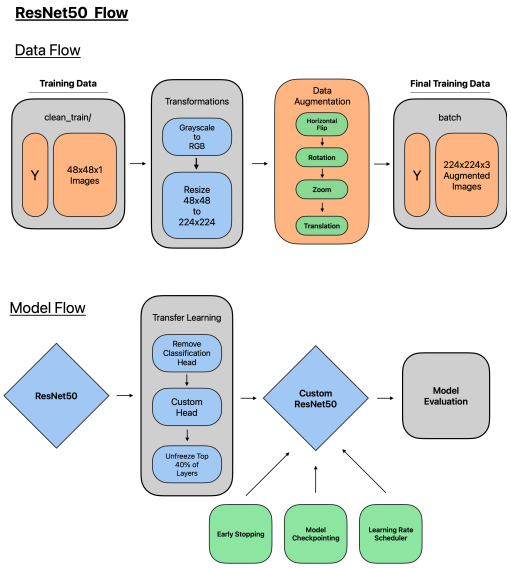
```
225/225 [==============================
===] - 8s 18ms/step - loss: 1.7254 -
accuracy: 0.3026
```

Out[80]:
```
[1.7253576517105103, 0.3025912642478
943]
```

Although the model evaluation is not the best, there are takeaways from the outputs. The first is that the Vision Transformer always predicts “happy”, which makes sense because it is the most frequent class. This means the model hasn’t reached the point where it begins learning. It is possible that this is a result of a small dataset.

5.3 ResNet50

In [13]:
```
git_tag = "https://raw.githubusercon
img_url = f"{git_tag}cs109b-final-pr
display(Image(url=img_url, width=650
```

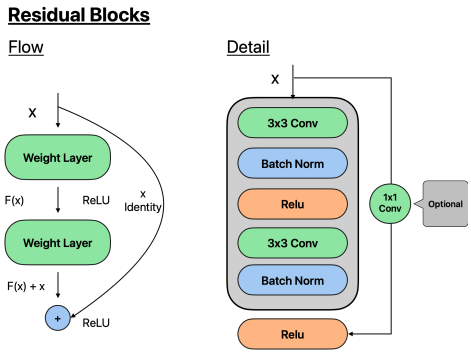


Flow Diagram 6. ResNet50 Model Flow

We wrap up our suite of modeling approaches with transfer learning, using ResNet50 as the base model (He et al., 2015). We used ResNet because prior literature has shown transfer learning approaches have been successful in getting close to the 70% accuracy benchmark (Khanzada et al., 2020). ResNet50 is 50 layers deep and introduces the idea of a residual block, shown in the figure below. A residual block is a block of batch normalization, convolution and ReLU activation. A 1x1 convolution is used in the skip connection to adjust dimensions if need be. Skip connections allow for deeper models because they allow gradients to flow through the network without vanishing.

The flow for ResNet50 is a bit more complex because the base model only takes RGB images and works most optimally for 224x224 images. We duplicate the channels to get three channels and upsample the images to 224x224. We use data augmentation but were deliberate in the augmentations used because too much can harm the test accuracy. We keep it conservative with only horizontal flipping, rotation, zoom and translation. Others such as brightness and contrast harmed the test accuracy.

In [14]:
```
git_tag = "https://raw.githubuserco
img_url = f"{git_tag}cs109b-final-p
display(Image(url=img_url, width=65
```



Flow Diagram 7. Residual Block Flow

For our transfer learning approach, we simply replace the classification head and let 40% of the top layers be trainable. In addition to data augmentation, to regularize the model further, we use label smoothing and weight decay. Label smoothing is a technique that helps the model generalize better by preventing it from becoming too confident in its predictions. It does this by slightly adjusting the target labels, making them less extreme. For example, instead of labeling a sample as 1 for "happy" and 0 for all other classes, we might label it as 0.9 for "happy" and 0.1 for the others.

A multistep approach has been done before, where in the first step a classification head is trained with an entirely frozen model and in the second step the model is unfrozen and fine-tuned (Ksheeraj Sai Vepur, 2021). It is worth noting that we attempted this approach as well but it didn’t get us past the 70% benchmark.

In [91]:

```
batch_size = 64
target_size = (48, 48)
SIZE = 224

res_train_datagen = image_dataset_
    DATA_DIR + "/clean_train",
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)

res_val_datagen = image_dataset_fr
    DATA_DIR + "/validation",
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)

res_test_datagen = image_dataset_f
    DATA_DIR + "/test",
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)

train_data_augmentation = tf.keras
    [
        layers.RandomFlip("horizor
        layers.RandomRotation(0.1)
        layers.RandomZoom(0.1),
        layers.RandomTranslation(0
    ]
)


def to_rgb(image):
    return tf.image.grayscale_to_r


def augment(image, label):
    image = to_rgb(image)
    image = tf.image.resize(image,
    image = train_data_augmentatic
    return image, label


def val_augment(image, label):
    image = to_rgb(image)
    image = tf.image.resize(image,
    return image, label


# Pipeline adjustments
```

```
train_dataset = (
    res_train_datagen.shuffle(buff
    .map(augment, num_parallel_cal
    .prefetch(tf.data.AUTOTUNE)
)

val_dataset = res_val_datagen.map(
    val_augment, num_parallel_call
).prefetch(tf.data.AUTOTUNE)

test_dataset = res_test_datagen.ma
    val_augment, num_parallel_call
).prefetch(tf.data.AUTOTUNE)
```

```
Found 22508 files belonging to 7
classes.
Found 5741 files belonging to 7 c
lasses.
Found 7178 files belonging to 7 c
lasses.
```

In [ ]:

```
base_model = ResNet50(
    weights="imagenet", include_tc
)

base_model.trainable = True  # all
# Typically unfreeze the top layer
for layer in base_model.layers[: -
    layer.trainable = False


input_tensor = layers.Input(shape=

base_model = ResNet50(include_top=


x = base_model.output
x = layers.GlobalMaxPooling2D()(x)
x = layers.Dense(256)(x)
x = layers.BatchNormalization()(x)
x = layers.Activation("relu")(x)
x = layers.Dropout(0.5)(x)
output_tensor = layers.Dense(7, ac


resnet = Model(inputs=input_tensor
```

In [94]:
```
# Callbacks
reduce_lr = ReduceLROnPlateau(
    monitor="val_accuracy",
    factor=0.5,
    patience=2,
    verbose=1,
    min_lr=1e-7,
    min_delta=0.001,
)

early_stop = EarlyStopping(
    monitor="val_accuracy",
    patience=5,
    restore_best_weights=True,
    verbose=1,
    min_delta=0.001,
)

MODEL_WEIGHTS_PATH = "./models/"
MODEL_HISTORY_PATH = "./model_hist

checkpoint = ModelCheckpoint(
    filepath=os.path.join(MODEL_WE
    monitor="val_accuracy",
    save_best_only=True,
    save_weights_only=True,
    verbose=1,
)
```

In [95]:
```
resnet.compile(
    optimizer=AdamW(learning_rate=
    loss=tf.keras.losses.Categoric
    metrics=["accuracy"],
)
```

In [96]:
```
if os.path.exists(os.path.join(MOD
    print("Loading pre-trained moc
    resnet.load_weights(os.path.jo

    resnet.compile(
        optimizer=AdamW(learning_r
        loss=tf.keras.losses.Categ
        metrics=["accuracy"],
    )
    with open(os.path.join(MODEL_H
        resnet50_history = pickle.
else:
    print("No pre-trained model fo
    resnet50_history = resnet.fit(
        train_dataset,
        validation_data=val_datase
        epochs=20,
        verbose=1,
        callbacks=[reduce_lr, earl
    )

    with open(os.path.join(MODEL_H
        pickle.dump(resnet50_histo
```

```
No pre-trained model found. Train
ing from scratch.
Epoch 1/20
352/352 [==========================
======] - ETA: 0s - loss: 1.7979
- accuracy: 0.4061
Epoch 1: val_accuracy improved fr
om -inf to 0.53022, saving model
to ./models/resnet50.weights.h5
352/352 [==========================
======] - 129s 211ms/step - loss:
1.7979 - accuracy: 0.4061 - val_l
oss: 1.3994 - val_accuracy: 0.530
2 - lr: 1.0000e-04
Epoch 2/20
352/352 [==========================
======] - ETA: 0s - loss: 1.3999
- accuracy: 0.5631
Epoch 2: val_accuracy improved fr
om 0.53022 to 0.57203, saving mod
el to ./models/resnet50.weights.h
5
352/352 [==========================
======] - 88s 244ms/step - loss:
1.3999 - accuracy: 0.5631 - val_l
oss: 1.3218 - val_accuracy: 0.572
0 - lr: 1.0000e-04
Epoch 3/20
352/352 [==========================
======] - ETA: 0s - loss: 1.2994
- accuracy: 0.6080
Epoch 3: val_accuracy did not imp
rove from 0.57203
352/352 [==========================
======] - 83s 230ms/step - loss:
1.2994 - accuracy: 0.6080 - val_l
oss: 1.4287 - val_accuracy: 0.521
2 - lr: 1.0000e-04
Epoch 4/20
352/352 [==========================
======] - ETA: 0s - loss: 1.2220
- accuracy: 0.6316
Epoch 4: val_accuracy improved fr
om 0.57203 to 0.60094, saving mod
el to ./models/resnet50.weights.h
5
352/352 [==========================
======] - 83s 230ms/step - loss:
1.2220 - accuracy: 0.6316 - val_l
oss: 1.3327 - val_accuracy: 0.600
9 - lr: 1.0000e-04
Epoch 5/20
352/352 [==========================
======] - ETA: 0s - loss: 1.1943
- accuracy: 0.6405
Epoch 5: val_accuracy did not imp
```

```
rove from 0.60094
352/352 [==============================] - 80s 224ms/step - loss: 1.1943 - accuracy: 0.6405 - val_loss: 1.2655 - val_accuracy: 0.5990 - lr: 1.0000e-04
Epoch 6/20
352/352 [==============================] - ETA: 0s - loss: 1.1741 - accuracy: 0.6501
Epoch 6: ReduceLROnPlateau reducing learning rate to 4.999999873689376e-05.

Epoch 6: val_accuracy did not improve from 0.60094
352/352 [==============================] - 88s 244ms/step - loss: 1.1741 - accuracy: 0.6501 - val_loss: 1.3385 - val_accuracy: 0.5520 - lr: 1.0000e-04
Epoch 7/20
352/352 [==============================] - ETA: 0s - loss: 1.1088 - accuracy: 0.6847
Epoch 7: val_accuracy improved from 0.60094 to 0.65163, saving model to ./models/resnet50.weights.h5
352/352 [==============================] - 80s 222ms/step - loss: 1.1088 - accuracy: 0.6847 - val_loss: 1.1640 - val_accuracy: 0.6516 - lr: 5.0000e-05
Epoch 8/20
352/352 [==============================] - ETA: 0s - loss: 1.0704 - accuracy: 0.7056
Epoch 8: val_accuracy did not improve from 0.65163
352/352 [==============================] - 85s 236ms/step - loss: 1.0704 - accuracy: 0.7056 - val_loss: 1.1773 - val_accuracy: 0.6504 - lr: 5.0000e-05
Epoch 9/20
352/352 [==============================] - ETA: 0s - loss: 1.0468 - accuracy: 0.7205
Epoch 9: val_accuracy improved from 0.65163 to 0.65685, saving model to ./models/resnet50.weights.h5
352/352 [==============================] - 86s 240ms/step - loss: 1.0468 - accuracy: 0.7205 - val_l
```

```
oss: 1.1854 - val_accuracy: 0.656
9 - lr: 5.0000e-05
Epoch 10/20
352/352 [==========================
======] - ETA: 0s - loss: 1.0227
- accuracy: 0.7355
Epoch 10: val_accuracy improved f
rom 0.65685 to 0.67288, saving mo
del to ./models/resnet50.weight
s.h5
352/352 [==========================
======] - 80s 222ms/step - loss:
1.0227 - accuracy: 0.7355 - val_l
oss: 1.1534 - val_accuracy: 0.672
9 - lr: 5.0000e-05
Epoch 11/20
352/352 [==========================
======] - ETA: 0s - loss: 0.9874
- accuracy: 0.7496
Epoch 11: val_accuracy did not im
prove from 0.67288
352/352 [==========================
======] - 81s 225ms/step - loss:
0.9874 - accuracy: 0.7496 - val_l
oss: 1.1586 - val_accuracy: 0.671
0 - lr: 5.0000e-05
Epoch 12/20
352/352 [==========================
======] - ETA: 0s - loss: 0.9573
- accuracy: 0.7657
Epoch 12: val_accuracy improved f
rom 0.67288 to 0.67706, saving mo
del to ./models/resnet50.weight
s.h5
352/352 [==========================
======] - 91s 253ms/step - loss:
0.9573 - accuracy: 0.7657 - val_l
oss: 1.1453 - val_accuracy: 0.677
1 - lr: 5.0000e-05
Epoch 13/20
352/352 [==========================
======] - ETA: 0s - loss: 0.9340
- accuracy: 0.7793
Epoch 13: val_accuracy did not im
prove from 0.67706
352/352 [==========================
======] - 82s 228ms/step - loss:
0.9340 - accuracy: 0.7793 - val_l
oss: 1.1618 - val_accuracy: 0.675
7 - lr: 5.0000e-05
Epoch 14/20
352/352 [==========================
======] - ETA: 0s - loss: 0.9115
- accuracy: 0.7872
Epoch 14: ReduceLROnPlateau reduc
ing learning rate to 2.4999999368
```

```
44688e-05.

Epoch 14: val_accuracy improved f
rom 0.67706 to 0.67776, saving mo
del to ./models/resnet50.weight
s.h5
352/352 [==========================
======] - 81s 226ms/step - loss:
0.9115 - accuracy: 0.7872 - val_l
oss: 1.1750 - val_accuracy: 0.677
8 - lr: 5.0000e-05
Epoch 15/20
352/352 [==========================
======] - ETA: 0s - loss: 0.8559
- accuracy: 0.8207
Epoch 15: val_accuracy improved f
rom 0.67776 to 0.68629, saving mo
del to ./models/resnet50.weight
s.h5
352/352 [==========================
======] - 82s 227ms/step - loss:
0.8559 - accuracy: 0.8207 - val_l
oss: 1.1516 - val_accuracy: 0.686
3 - lr: 2.5000e-05
Epoch 16/20
352/352 [==========================
======] - ETA: 0s - loss: 0.8276
- accuracy: 0.8313
Epoch 16: val_accuracy did not im
prove from 0.68629
352/352 [==========================
======] - 81s 225ms/step - loss:
0.8276 - accuracy: 0.8313 - val_l
oss: 1.1816 - val_accuracy: 0.683
2 - lr: 2.5000e-05
Epoch 17/20
352/352 [==========================
======] - ETA: 0s - loss: 0.7993
- accuracy: 0.8480
Epoch 17: ReduceLROnPlateau reduc
ing learning rate to 1.2499999684
22344e-05.

Epoch 17: val_accuracy improved f
rom 0.68629 to 0.68647, saving mo
del to ./models/resnet50.weight
s.h5
352/352 [==========================
======] - 80s 221ms/step - loss:
0.7993 - accuracy: 0.8480 - val_l
oss: 1.1852 - val_accuracy: 0.686
5 - lr: 2.5000e-05
Epoch 18/20
352/352 [==========================
======] - ETA: 0s - loss: 0.7710
- accuracy: 0.8621
```

```
Epoch 18: val_accuracy improved f
rom 0.68647 to 0.69552, saving mo
del to ./models/resnet50.weight
s.h5
352/352 [==========================
======] - 80s 224ms/step - loss:
0.7710 - accuracy: 0.8621 - val_l
oss: 1.1842 - val_accuracy: 0.695
5 - lr: 1.2500e-05
Epoch 19/20
352/352 [==========================
======] - ETA: 0s - loss: 0.7547
- accuracy: 0.8692
Epoch 19: val_accuracy did not im
prove from 0.69552
352/352 [==========================
======] - 80s 222ms/step - loss:
0.7547 - accuracy: 0.8692 - val_l
oss: 1.1858 - val_accuracy: 0.693
8 - lr: 1.2500e-05
Epoch 20/20
352/352 [==========================
======] - ETA: 0s - loss: 0.7415
- accuracy: 0.8747
Epoch 20: ReduceLROnPlateau reduc
ing learning rate to 6.2499998421
1172e-06.

Epoch 20: val_accuracy did not im
prove from 0.69552
352/352 [==========================
======] - 81s 225ms/step - loss:
0.7415 - accuracy: 0.8747 - val_l
oss: 1.2289 - val_accuracy: 0.676
0 - lr: 1.2500e-05
```

In [97]:
```
plot_accuracy_and_loss(
    resnet50_history,
    "ResNet50 with Data Augmentati
)
```

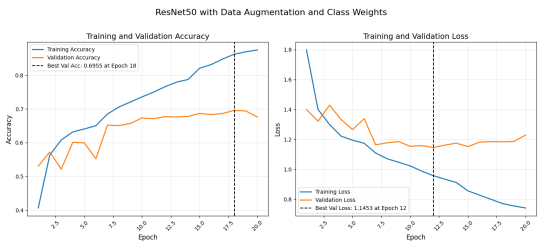


Figure 15. ResNet50 with Data Augmentation and Class Weights

The validation accuracy peaks at 69.6%. From the loss function we see the callback stops the model training at an optimal time, before overfitting occurs. This result is on par with existing projects that we researched.

```
In [98]: y_true_res, y_pred_res, cm_res =

         plot_conf_matrix(cm_res, "ResNet5
```

```
ResNet50 Classification report
              precision    recal
l  f1-score   support

       angry       0.59      0.6
3      0.61       958
     disgust       0.65      0.5
9      0.62       111
        fear       0.60      0.4
1      0.48      1024
       happy       0.90      0.8
5      0.87      1774
     neutral       0.60      0.6
8      0.64      1233
         sad       0.53      0.6
3      0.57      1247
    surprise       0.81      0.7
8      0.80       831

    accuracy
0.68      7178
   macro avg       0.67      0.6
5      0.66      7178
weighted avg       0.69      0.6
8      0.68      7178
```

ResNet50 Confusion matrix

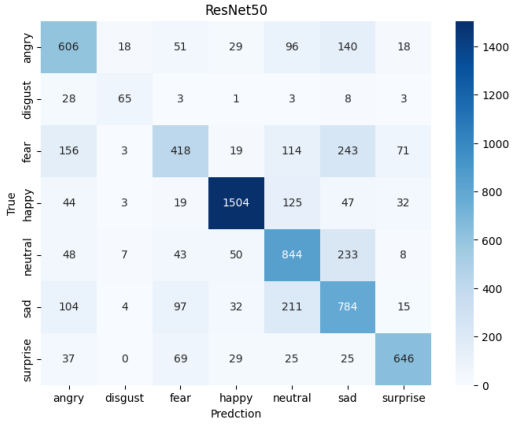


Figure 16. ResNet50 Confusion Matrix

The “disgust” category isn't really an issue. “Sad” and “fear” seem to be the most difficult to classify. For “sad”, the model confuses it with “neutral”, just like the u-net model. As for “fear”, the model confuses it for “anger” and “sadness”, just like prior models. The model is able to classify “happy” and “surprise” well, which is expected since these are high arousal emotions with distinct facial features like teeth and wide eyes.

In [99]:

```
testgen = image_dataset_from_di
    DATA_DIR + "/test",
    image_size=target_size,
    batch_size=batch_size,
    label_mode="categorical",
    color_mode="grayscale",
)

test_dataset = testgen.map(val_
    tf.data.AUTOTUNE
)

# Evaluate the model on the tes
test_loss, test_accuracy = resn
```

```
Found 7178 files belonging to 7
classes.
113/113 [==========================
========] - 7s 61ms/step - loss
: 1.2372 - accuracy: 0.6780
```

ResNet50 achieves a slightly higher accuracy than the enhanced U-Net model. Experiments were done on EfficientNetB3 as well but the performance was similar and slower to train, which made it a less ideal choice for this task. Oversampling the “disgust” category and MixUp augmentation were also tested but did not yield significant improvements. An attempt was also made to train a custom, conditional diffusion model to generate faces but this proved too difficult because models were not generating realistic faces (Tang et al., 2022). As for GAN’s, we as a team deemed it too risky because there’s no guarantee of convergence. We also tried to use SMOTE as well, oversampling the latent space rather than the pixel space, but this didn’t impact results . Class weights also proved to be a detriment to model performance, which is why they were excluded from ResNet50. All these findings were in line with the literature.

Data imbalance didn’t seem to be the big issue for the models, rather models had a tough time distinguishing faces as mutually exclusive categories. This makes sense because faces can convey multiple emotions. Knowing this Microsoft has an FER+ dataset with multi-labels, which addresses this problem more directly than any modeling approach can

(microsoft, n.d.).

In the end, we threw the kitchen sink at this problem, trying every attempt we learn in the course. It is frustrating to not beat the human benchmark but sometimes that is just how it goes.

6. Visualizations

All visualizations above are clean, well-labeled, and designed to communicate insights clearly and effectively. Each plot includes appropriate titles, axis labels, and legends, ensuring they are ready for both presentations and reports. Below, we present a series of additional visualizations that go beyond traditional EDA to assess and interpret the behavior of our trained baseline models. These visualizations help us better understand how the models make decisions, what features they rely on, and where they focus their attention.

6.1 GradCAM

To better understand how our models make predictions, we used Grad-CAM to generate class activation heatmaps for both the baseline_model and the baseline_std_model. These visualizations highlight the regions of each image that the model focuses on when making classification decisions.

Key Findings:

*Baseline Model
The Grad-CAM outputs for this model tend to be concentrated around core facial features—particularly the head, nose, and mouth. This tighter focus indicates that the model has learned to attend to more relevant and discriminative regions, which aligns with its improved performance metrics. However, sometimes it focuses on the background, which is not ideal.

*U-Net Model with Attention
The Grad-CAM outputs for this model were more concentrated around multiple, core facial features—particularly the head, nose, and mouth. This tighter focus indicates that the model has learned to attend to relevant and multiple features, which aligns with what squeeze-excite blocks are designed to do.

*ResNet50
The model has a much larger receptive field in comparison to the other models. This is in line with the performance, as it has

been shown the accuracy and the size of a receptive field are positively related (Araujo et al., 2019). The model learns to focus on the entire face which is a good thing. The model can also distinguish between the background and the face.

In [100…

```
replace2linear = ReplaceToLinea


def generate_gradcam(model, img
    gradcam = Gradcam(model, mo
    score = CategoricalScore([c
    # This is equivalent to the
    # score = lambda outputs: o
    gradcam_results = gradcam(s
    return gradcam_results


def show_gradcam(model, img_arr
    cam_result = generate_gradc

    fig, ax = plt.subplots(1, 2
    ax[0].imshow(img_array[0].s
    ax[0].set_title("Input Imag
    ax[0].axis("off")

    ax[1].imshow(img_array[0].s
    ax[1].imshow(cam_result[0],
    ax[1].set_title(f"GradCAM:
    ax[1].axis("off")
    plt.tight_layout()
    plt.show()


# We'll pick one image and gene
cleaned_train_gen = val_test_da
    DATA_DIR + "/clean_train",
    target_size=(48, 48),
    batch_size=32,
    color_mode="grayscale",
    class_mode="categorical",
    shuffle=True,
)
sample_batch, labels = next(cle
sample_img = sample_batch[0:1]
pred_class = np.argmax(baseline

show_gradcam(
    baseline_model,
    sample_img,
    pred_class,
```

```
    title="Baseline Model with
)

pred_class = np.argmax(unet_mod
show_gradcam(
    enhanced_model,
    sample_img,
    pred_class,
    title="UNet Model with Atte
)
```

```
1/1 [==============================
====] - 0s 53ms/step
```

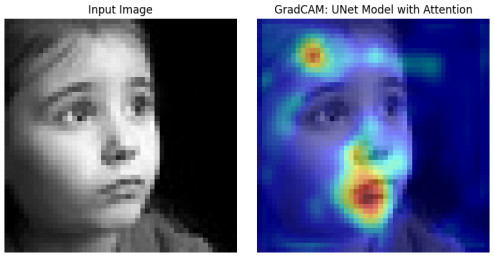


Figure 17. GRADCAM: Baseline Model with MinMax Scaling Figure 18. GRADCAM: UNet Model with Attention

In [ ]:

```
sample_img = sample_batch[0:1]

# Need to change to RGB
sample_img_resized = tf.image.
sample_img_rgb = tf.repeat(sam

pred_class = np.argmax(resnet.

# Use the same RGB image for G
show_gradcam(
    resnet,
    sample_img_rgb,
    pred_class,
    title="ResNet50",
)
```

```
1/1 [==============================
======] - 0s 66ms/step
```

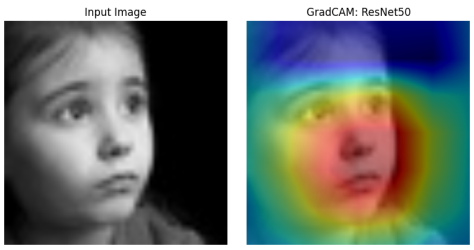


Figure 19. GRADCAM: ResNet50

6.2 Model Performance Table

Models	Test Accuracy (%)
Human Benchmark	65 ± 5
Baseline CNN	50.4
Basline U-Net	64.8
U-Net with Attention	60.0
Vision Transformer	30.3
ResNet50	67.8

The best performing model is ResNet50, with a test accuracy of 67.8%. Interestingly, the U-Net model with attention performed slightly worse at 60.0%, while the baseline U-Net achieved 64.8%. The Vision Transformer model had the lowest performance at 30.3%. Overall, even though the models weren't able to surpass the human benchmark, they still performed reasonably well under the restriction that no auxiliary data or multi-labels were used. Matching human benchmark performance is a challenging task, especially in the context of facial expression recognition, where subtle variations can significantly impact classification accuracy. We were successfully able to match that.

Conclusion and Future Work

Revisiting the research questions, we have

1. What is the best custom model architecture for classifying human facial expressions and how well does it perform compared to existing models in the literature?

The best model was found to be ResNet50, which achieved a test accuracy of 67.8%. This is on par with other projects in the literature, which have reported accuracies ranging from 65% to 70% on strictly the FER-2013 dataset. The U-Net model performed relatively well, matching the human benchmark with 64.8% accuracy.

2. How does the model perform for each emotion class? What are the implications of the results?

For our ResNet50 model, the classes with the lowest f1-score were `sad` and `fear`. Both were less than 0.6. Surprisingly, the `disgust` class, which is the smallest class, was not the worst performing class. It makes sense that `fear` and `sad` are the most difficult to classify because they are similar, overlapping emotions

overlapping emotions.

3. How do vision transformer models compare to CNN models in terms of performance for the FER-2013 dataset?

The vision transformer model performed the worst of all the models, achieving only 30.3% accuracy, only twice as better than random guessing (14.3%). This is likely due to the fact that the FER-2013 dataset is small and noisy. The global self-attention property of the transformer architecture may not be well-suited for this type of data.

As a group, we met the benchmarks set in other works. If we weren't restricted to FER-2013, we would use FER+ to convert this problem from multiclass classification to multi-label. We believe it will make the biggest impact on accuracy. It will help the model understand that emotions like fear and sadness are not mutually exclusive. Furthermore, we could have used auxiliary datasets like JAFFE to see if we can match the benchmarks set in other papers (Lyons et al., 1998).

Sources

Araujo, A., Norris, W., & Sim, J. (2019). Computing receptive fields of convolutional neural

networks. Distill, 4(11). https://doi.org/10.23915/distill.00021

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2020, October 22). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv.org. https://arxiv.org/abs/2010.11929

Dz. (2023, September 17). Intro to Diffusion Model — Part 5 - DZ - Medium. Medium. https://dzdata.medium.com/intro-to-diffusion-model-part-5-d0af8331871

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. arXiv (Cornell University). https://doi.org/10.48550/arxiv.1512.03385

Ho, J., Jain, A., & Abbeel, P. (2020). Denoising diffusion probabilistic models. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2006.11239

Khanzada, A., Bai, C., & Celepcikay, F. T. (2020). Facial Expression Recognition with Deep Learning. In https://cs230.stanford.edu/projects_winter_2020/reports/32610274.pdf. Stanford University.

Ksheeraj Sai Vepur. (2021). Improving Facial Emotion Recognition with Image processing andImproving Facial Emotion Recognition with Image processing and Deep LearningDeep Learning. In https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=2029&context=etd_project San Jose State University.

Kubo, H. (2024, December 28). Understanding Transformer sinusoidal Position embedding. Medium. https://medium.com/%40hirok4/understanding-transformer-sinusoidal-position-embedding-7cbaaf3b9f6a

Lyons, M., Kamachi, M., & Gyoba, J. (1998). The Japanese Female Facial Expression (JAFFE) dataset [Dataset]. In Zenodo (CERN European Organization for Nuclear Research). https://doi.org/10.5281/zenodo.3451524

microsoft. (n.d.). GitHub - microsoft/FERPlus: This is the FER+ new label annotations for the Emotion FER dataset. GitHub. https://github.com/microsoft/FERPlus

Nichol, A., & Dhariwal, P. (2021, February 18). Improved denoising diffusion probabilistic models. arXiv.org. https://arxiv.org/

Appendix

Mathematical Formulae

Non-Local Means Denoising

The NLM algorithm is defined as the following transformation. Let $u(p)$ be the transformed pixel value at position p in the image. $C(p)$ is a normalizing constant and Ω is the set of pixels in the image. $v(q)$ is the untransformed pixel.

$$u(p) = \frac{1}{C(p)} \sum_{q \in \Omega} v(q) \cdot w(p,q) d$$

The weight function $w(p,q)$ is defined as follows:

$$w(p,q) = exp(-\frac{|B(q) - B(p)|^2}{h^2}$$

h is the parameter we pass into the NLM function. It controls the strength of the denoising. A larger value of h means more smoothing and less detail. $B(p)$ is a patch of pixels around the pixel p. The size of the patch is defined by the parameter `templateWindowSize`. $B(q)$ is a patch of pixels around the pixel q. The size of the patch is defined by the parameter `searchWindowSize`. The weight measures how similar the patches $B(p)$ and $B(q)$ are. The weight will be larger if the patches are more similar.

Mathematics of GradCAM

https://arxiv.org/abs/2105.05233

OpenCV: Denoising. (n.d.). https://docs.opencv.org/3.4/d1/d79/group__photo__denoise.html#ga03

Papers with Code - FER2013 Benchmark (Facial Expression Recognition (FER)). (n.d.). https://paperswithcode.com/sota/facial-expression-recognition-on-fer2013

Tang, Y., Zhai, W., & Liu, B. (2022). Facial Expression Manipulation with Conditional Diffusion Model. In https://cs230.stanford.edu/projects_fall_2022/reports/78.pdf. Department of Computer Science, Stanford University.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017, June 12). Attention is all you need. arXiv.org. https://arxiv.org/abs/1706.03762

Wikipedia contributors. (2023, January 24). Non-local means. Wikipedia. https://en.wikipedia.org/wiki/Non-local_means

GradCAM is a technique that uses the gradients of the target class with respect to the feature maps of the last convolutional layer to produce a coarse localization map highlighting the important regions in the image for predicting the class. It is preferred over CAM because it is agnostic to the architecture of the model and can be applied to any CNN-based model. The algorithm is as follows:

Step 1

Get the gradient of the class score with respect to the feature maps $\frac{\partial y_c}{\partial A_k}$. This measures how a class score changes with respect to a change in the feature map. We denote k as the index for a given feature map.

Step 2

We apply global average pooling to the map of gradients

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j \frac{\partial y_c}{\partial A_k^{ij}}$$

The α_k^c are the weights for each feature map k. The indices i and j are the spatial dimensions of the feature map. This measures how an activation in the feature map affects the class score.

Step 3

Perform weighted sum of feature maps using these weights

$$g_c = \sum_k \alpha_k^c A_k$$

Step 4

ReLU the result

$$g_c^* = ReLU(g_c)$$

This means we apply the activation function to each pixel in the gradCAM map.

Step 5

Upsample the gradCAM map to match the dimensions of the input image.

Squeeze-Excite (SE) Block

The first step in an SE block is to apply global average pooling to the feature maps. Let X denote a feature map in $\mathbb{R}^{H \times W \times C}$, where H is the height, W is the width, and C is the number of channels. The global average pooling operation computes the average value of each channel across all spatial locations:

$$z_c = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} X_{i,j,c}.$$

The operation produces a vector $z \in \mathbb{R}^C$, where each element z_c represents the average activation of the c-th channel.

Next we apply excitation. We pass z through two fully connected layers with the first using ReLU and the second using sigmoid activation. The first layer reduces the dimensionality of z by a factor. For our design we used 16. In the formula below, W_1 and W_2 are the weights of the first and second dense layers, respectively:

$$s = \sigma(W_2 \cdot ReLU(W_1 \cdot z)).$$

The output of these two layers is a vector $s \in \mathbb{R}^C$, where each element s_c represents the importance of the c-th channel. We multiply the original feature map X by the excitation vector s to recalibrate the feature maps:

$$X_{SE} = X \cdot s.$$

This operation is done using the `Multiply()` layer in keras.

Diffusion Model

This subsection is optional to the reader and was only included per TA's request. It contains code for training a diffusion model using a U-Net architecture. The idea was to train on the "disgust" category to upsample for our models but it ultimately failed. However, we learned a lot about diffusion models and Kaylee finally found her spark after a decade of searching.

Citing our sources, parts of this section were AI assisted via DeepSeek. The functions marked with "AI" were AI assisted.

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as
import pytorch_lightning as p

from torch.utils.data import
from torchvision import trans
```

In [ ]:
```
# Determinisitc, relative emb
class SinusoidalTimeEmbedding
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, t):
        device = t.device
        half_dim = self.dim /
        emb = torch.log(torch
        # had to use device b
        emb = torch.exp(torch
        emb = t[:, None] * em
        emb = torch.cat([torc
        return emb
```

The positional embeddings, just like in the transformer paper (Vaswani et al., 2017) are defined as follows:

$$PE_{pos}(t) = \begin{cases} \sin(\frac{t}{10000^{\frac{2i}{d}}}) & \text{if} \\ \cos(\frac{t}{10000^{\frac{2i-1}{d}}}) & \text{if} \end{cases}$$

The pytorch code looks slightly different but it is equivalent. We use a similar approach to (Kubo, 2024) as a more efficient way to compute the positional embeddings. These embeddings are non-trainable and represent relative positions in a time sequence. This is in contrast to vision transformers, which have learnable positional embeddings.

Positional embeddings are used to encode the time step information into the model. These embeddings are fed into the residual blocks of the U-Net as an additional feature map. This is akin to conditional generative adversarial networks (cGANs) where the condition is added as an additional feature map.

Next, we create a class for the attention block. The attention block is the standard attention block we have all seen before. The only difference is that we use a linear layer instead of a convolutional layer to compute the query, key and value matrices. The attention block allows the model to focus on

different parts of the input image.

In [ ]:

```
# Very similar to HW5 attenti
class AttentionBlock(nn.Modul
    def __init__(self, channe
        super().__init__()
        self.query = nn.Conv2
        self.key = nn.Conv2d(
        self.value = nn.Conv2

        self.softmax = nn.Sof

    def forward(self, x):
        # batch, channels, he
        B, C, H, W = x.shape

        # Q, K, V projections
        query = self.query(x)
        key = self.key(x).vie
        value = self.value(x)

        attention_map = torch
        attention_map = self.
        out = torch.matmul(at

        out = out.transpose(1

        return out
```

For the next class, we create a residual block. A lot of the code was taken from a medium post (Dz, 2023). The architecture as defined in the init contains two convolutional layers, a group normalization, attention block and a skip connection. Group norm is essentially identical to batch norm but it is batch size agnostic. For all intents and purposes, one can view it as a bathch norm layer.

In the forward pass, we apply convolution, project the time embedding and add it as an additional feature map. We then apply convolution again and pass the result through an attention block. The final step is a skip connection.

In [ ]:
```
# Residual Block with Time Em
class ResBlock(nn.Module):
    def __init__(self, in_cha
        super().__init__()
        self.conv1 = nn.Conv2
        self.norm1 = nn.Group
        self.act1 = nn.SiLU()

        self.conv2 = nn.Conv2
        self.norm2 = nn.Group
        self.act2 = nn.SiLU()

        # Inject the time emb
        self.time_mlp = nn.Li
        self.attn = Attentior
        self.res_conv = (
            nn.Conv2d(in_char
            if in_channels !=
            else nn.Identity(
        )

    def forward(self, x, t_em
        h = self.act1(self.nc
        # Add the time embedd
        h = h + self.time_mlp
        h = self.act2(self.nc
        h = self.attn(h)
        # Add the residual co
        return h + self.res_c
```

The class below is just a U-Net. There's not much to add here since we used U-Net extensively in the main project. It's the standard encoder, bottleneck, decoder architecture with long skip connections. We alo use residual blocks with attention instead of vanilla convolution blocks.

In [ ]:

```
# U-Net Architecture
class SimpleUNet(nn.Module):
    def __init__(self):
        super().__init__()
        time_emb_dim = 256
        self.time_embedding =
        self.time_mlp = nn.Se
            nn.Linear(time_em
            nn.SiLU(),
            nn.Linear(time_em
        )
        ## Define the archite
        # Encoder
        self.enc1 = ResBlock(
        self.pool1 = nn.MaxPo

        self.enc2 = ResBlock(
        self.pool2 = nn.MaxPo

        # Bottleneck
        self.bottleneck = Res

        # Decoder
        self.up1 = nn.ConvTra
        self.dec1 = ResBlock(

        self.up2 = nn.ConvTra
        self.dec2 = ResBlock(

        self.out = nn.Conv2d(

    ## Define the forward pas
    def forward(self, x, t):
        t_emb = self.time_emb
        t_emb = self.time_mlp

        # Encoder
        e1 = self.enc1(x, t_e
        p1 = self.pool1(e1)

        e2 = self.enc2(p1, t_
        p2 = self.pool2(e2)

        # Bottleneck
        b = self.bottleneck(p

        # Decoder
        up1 = self.up1(b)
        d1 = self.dec1(torch.

        up2 = self.up2(d1)
        d2 = self.dec2(torch.

        return self.out(d2)
```

The model has been defined. Now we move on the interesting part, the training loop. First we define the forward process. The forward process is a Markov chain that progressively adds noise to the data. The noise is drawn from a multivariate standard normal. We define the forward process as

$$q(x_t|x_{t-1}) = \mathcal{N}(\mu_t = \sqrt{1 -$$

The variance schedule is defined as β_t. In the class below, we use a linear schedule. Cosine schedules have been shown to be better but we stick to linear for simplicity (Nichol & Dhariwal, 2021). The variable x_t is the image at time t and by the Markov assumption, the image at time t only depends on the image at time $t-1$. Like shown in lecture, we reparameterize the forward process using the following equations:

$$\alpha_t = 1 - \beta_t,$$

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s.$$

Equation 1 now becomes

$$q(x_t|x_0) = \mathcal{N}(\mu_t = \sqrt{\bar{\alpha}_t}x_0, \sigma_t^2 =$$

All variables have been defined. The class below implements the forward process.

In [ ]:
```
# Forward process of the dif
class Diffusion:
    def __init__(self, timest
        self.timesteps = time
        self.betas = torch.li
        self.alphas = 1.0 - s
        self.alphas_cumprod =

    def q_sample(self, x_star
        if noise is None:
            noise = torch.ran

        # need to be on same
        # was getting same bu
        device = x_start.devi
        alphas_cumprod = self
        sqrt_alpha_cumprod =
        sqrt_one_minus = torc

        return sqrt_alpha_cum
```

Author's Notes

Since the training loop was only introduced in the last lecture, we had to use DeepSeek for help. The general concept was clear but the implementation was not. The tricky part was step 4 and 5 because it was vague on what to do about 'take gradient step on $\nabla_\theta||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)$'. The AI was able to clarify this and the code below is the result of that.

We follow the steps outlined in the paper "Denoising Diffusion Probabilistic Models" (Ho et al., 2020). The steps can be found on slide 42 of the final lecture. The steps are as follows:

In [16]:
```
git_tag = "https://raw.github
img_url = f"{git_tag}cs109b-
display(Image(url=img_url, wi
```

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: **until** converged

Appendix Fig 1. Diffusion Training Steps

In [ ]:
```
# AI-assisted
class DiffusionModel(pl.Ligh
    def __init__(self):
        super().__init__()

        # Denoising model, r
        self.model = SimpleU

        # Foward process
        self.diffusion = Di

    def forward(self, x, t)
        # Inputs are x and
        return self.model(x

    def training_step(self,

        # Step 1: sample ima
        x, _ = batch

        # Step 2: Sample tim
        t = torch.randint(
            0, self.diffusio
        ).long()

        # Step 3: Sample noi
        noise = torch.randn_

        # Step 4: Generates
        x_noisy = self.diffu

        # Step 5: Predict th
        noise_pred = self(x_

        # Compute the MSE
        loss = F.mse_loss(no
        self.log("train_loss
        return loss

    def configure_optimizers
        optimizer = torch.op
        return optimizer
```

In [ ]:
```
transform = transforms.Compo
    [
        transforms.Grayscal
        transforms.Resize((
        transforms.ToTensor
        transforms.Normaliz
    ]
)
```

In [ ]:
```
# Here we upsample all but
# created another directory
train_dataset = datasets.Ima
train_loader = DataLoader(t

model = DiffusionModel()
trainer = pl.Trainer(max_ep
trainer.fit(model, train_lo
```

```
Using default `ModelCheckpo
talling `litmodels` package
elCheckpoint` for automatic
htning model registry.
GPU available: True (cuda),
TPU available: False, using
HPU available: False, using
You are using a CUDA device
RTX 4090') that has Tensor C
utilize them, you should se
32_matmul_precision('medium
will trade-off precision fo
more details, read https://p
table/generated/torch.set_f
ision.html#torch.set_float32
2025-05-07 20:44:28.425586:
xla/xla/stream_executor/cuda
1] Unable to register cuDNN
ng to register factory for p
one has already been regist
2025-05-07 20:44:28.425632:
xla/xla/stream_executor/cuda
Unable to register cuFFT fa
to register factory for plu
has already been registered
2025-05-07 20:44:28.426940:
xla/xla/stream_executor/cuda
5] Unable to register cuBLAS
ing to register factory for
n one has already been regis
2025-05-07 20:44:28.434009:
e/platform/cpu_feature_guar
sorFlow binary is optimized
CPU instructions in performa
ations.
To enable the following ins
A, in other operations, rebu
th the appropriate compiler
2025-05-07 20:44:29.224383:
iler/tf2tensorrt/utils/py_u
Warning: Could not find Ten
LOCAL_RANK: 0 - CUDA_VISIBL

  | Name  | Type       | Pa
----------------------------
0 | model | SimpleUNet | 2.4
----------------------------
2.4 M     Trainable params
0         Non-trainable para
2.4 M     Total params
9.507     Total estimated mo
(MB)
81        Modules in train m
0         Modules in eval mo
/usr/local/lib/python3.10/di
ch_lightning/trainer/connec
```

```
r.py:425: The 'train_dataloa
e many workers which may be
sider increasing the value
s` argument` to `num_worker
Loader` to improve performa
/usr/local/lib/python3.10/d
ch_lightning/loops/fit_loop
r of training batches (6) i
logging interval Trainer(lo
0). Set a lower value for l
f you want to see logs for
h.
Training: |          | 0/?
[00:00<?, ?it/s]
`Trainer.fit` stopped: `max
_epochs=1000` reached.
```

Author's Notes

Sampling was trickiest because we barely covered it in the lecture. We knew the general idea is to feed in a noise vector and then iteratively denoise it but how to do so was not clear. The breakthrough occurred when we realized the process can be viewed as solving a stochastic differential equation (SDE), specifically a variance preserving SDE. Framing it this way made it clear why we have a σ_t term and what the denoising equation is doing.

The forward process can be viewed as

$$dx = f(x, t)dt + g(t)dw$$

- g(t) is the diffusion coefficient (in lecture it was β_t)
- dw is brownian motion
- f(x, t) is the drift term, with x being the image at time t

The equation can be read as the change in the image is equal to the drift term multiplied by time plus brownian motion scaled by the diffusion coefficient.

The reverse process can be viewed as

$$dx = [f(x, t) - g(t)^2\nabla_x \log p(x$$

The most important term is

$\log p(x, t)$, the score term, which the model is trying to learn. Note that the model doesn't learn this directly because diffusion models are trained using ELBO (evidence lower bound) as mentioned in the lecture. The ELBO is a lower bound on the log likelihood of the data. We do this because maximizing the log likelihood is intractable.

Sampling is where we take a noise vector and iteratively denoise it. We do so by using a special equation

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta($$

The equation takes as input the noise vector x_t, time step t and the model prediction $\epsilon_\theta(x_t, t)$. The σ_t term is added to introduce noise because the forward process is stochastic and therefore the reverse process must also be stochastic. It ensures mathematical consistency for the stochastic differential equation (SDE) we are solving.

In [15]:
```
git_tag = "https://raw.githu
img_url = f"{git_tag}cs109b-
display(Image(url=img_url, w
```

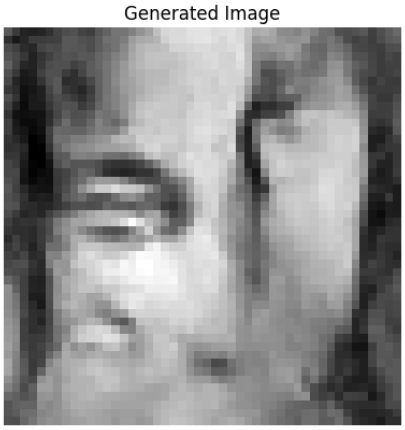
Algorithm 2 Sampling

$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \ldots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
end for
return $\mathbf{x}_0$

Appendix Fig 2. Image Sampling Steps

In [ ]:
```python
# AI-assisted
@torch.no_grad()
def generate_image(
    model,
    diffusion,
    image_size=(1, 48, 48)
    device="cuda" if torch
):
    model.eval()
    model.to(device)

    # Step 1: Sample rando
    img = torch.randn(1, *

    for t in reversed(rang
        t_tensor = torch.f

        # Predict the nois
        noise_pred = model

        # Get alpha parame
        alpha_t = diffusio
        alpha_cumprod_t =
        beta_t = diffusion

        # Step 4: Apply th
        coef1 = 1 / torch.
        coef2 = beta_t / t
        img = coef1 * (img

        # Step 3
        # Add noise except
        # we want final im
        if t > 0:
            noise = torch.
            sigma_t = torc
            img += sigma_t

    img = torch.clamp(img,
    img = (img + 1) / 2
    return img
```

In [ ]:
```python
# Generate an image
generated_image = generate

# Convert to numpy
img_np = generated_image.s

plt.imshow(img_np, cmap="g
plt.axis("off")
plt.title("Generated Image
plt.show()
```



Appendix Fig 3. Sampled Image

The results are subpar. Generating human faces with specific emotions is a very difficult task that most likely cannot be solved with this approach. The image above looks like noise but we can discern a few characteristics of a face. We can see the model is learning the shape of the face and the background. We can see evidence of eyes and a nose.