Ravirajpurohit Follow
Apr 2 · 9 min read · Listen

Facial Expression Classifier

Link to the Jupyter Notebook

Brief about the problem statement

In recent years, there has been a significant increase in the use of machine learning algorithms to analyze and understand images. One of the most common tasks in image analysis is image classification, which involves categorizing images into predefined classes. In this project, the goal is to build a neural network classifier algorithm using the TensorFlow library for image classification of facial expressions. The model will be trained on a dataset of facial images and will learn to classify these images based on the expressions they portray.

About the dataset

The dataset used for this project is a collection of facial expression images that are labeled into 7 different classes: angry, happy, sad, disgust, neutral, fear, and surprise. Each image in the dataset has a dimension of 48 by 48 pixels, and the images are grayscale. The dataset is already split into training and validation sets. The training set is used to train the neural network classifier algorithm, while the validation set is used to evaluate the performance of the model. The dataset is a widely used benchmark dataset in the field of computer vision, and it has been used in several previous studies related to facial expression recognition. With the availability of this dataset, researchers can train and test their models on a standardized dataset, making it easier to compare the performance of different approaches.

Reference

I have used the following notebook and explanation for reference -

SamanehEslamifar (2021). Facial Emotion Expressions [Data set]. Kaggle. https://www.kaggle.com/datasets/samaneheslamifar/facial-emotion-expressions

Data Exploration and Preparation

Import the libraries required to run the program

```
## import libraries
import numpy as np
import pandas as pd
import os
from tqdm import tqdm
import time

import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
import glob
import cv2

from tensorflow.keras.utils import to_categorical
from tensorflow.keras.utils import load_img
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D
from keras.utils.vis_utils import plot_model
```

Traverse over the directories to see classes of facial expressions

```
main_folder = '../input/facial-emotion-expressions/images'

labels_training = os.listdir(main_folder + "/train")
labels_testing = os.listdir(main_folder + "/validation")
```

```
print(f'Total Train Classes - {len(labels_training)} - {labels_training}')
print(f'Total Validation Classes - {len(labels_testing)} - {labels_testing}')
```

Generate dataframe containing image path and the respective label

```
def generate_data_df(fol_path, dataset_name):
    print(f"Creating {dataset_name}")
    df = {"src_path":[],"label":[]}
    for label in os.listdir(fol_path):
            for src_path in glob.glob(f"{fol_path}/{label}/*"):
                df["src_path"].append(src_path)
                df["label"].append(label)
    df = pd.DataFrame(df)
    print("Done!")
    return df
```

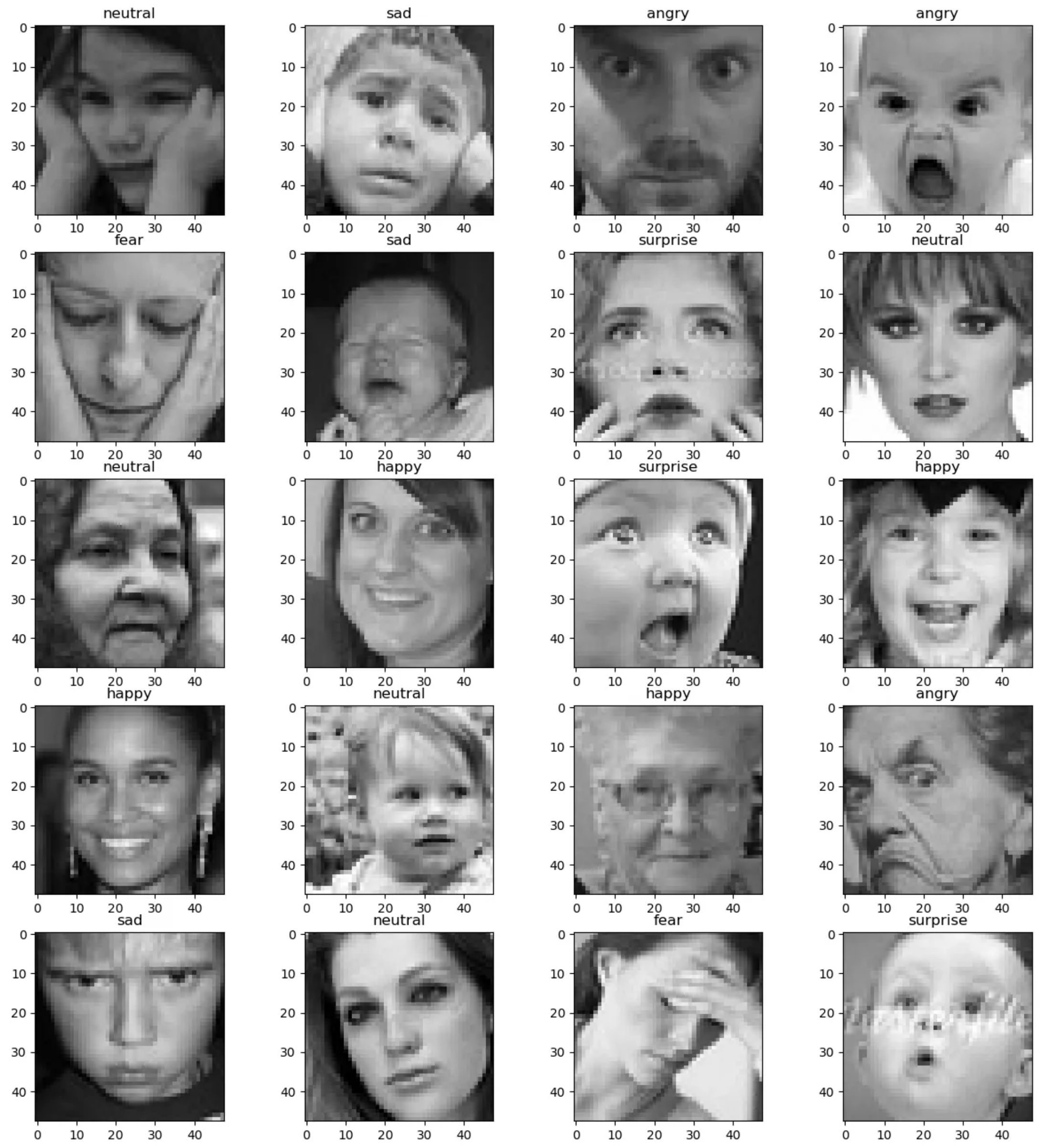
```
df_train = generate_data_df(main_folder + "/train", "Training dataset")
df_valid = generate_data_df(main_folder + "/validation", "Validation dataset")
```

Look at the number of items in each class to understand the data distribution

```
df_train.label.value_counts()
```

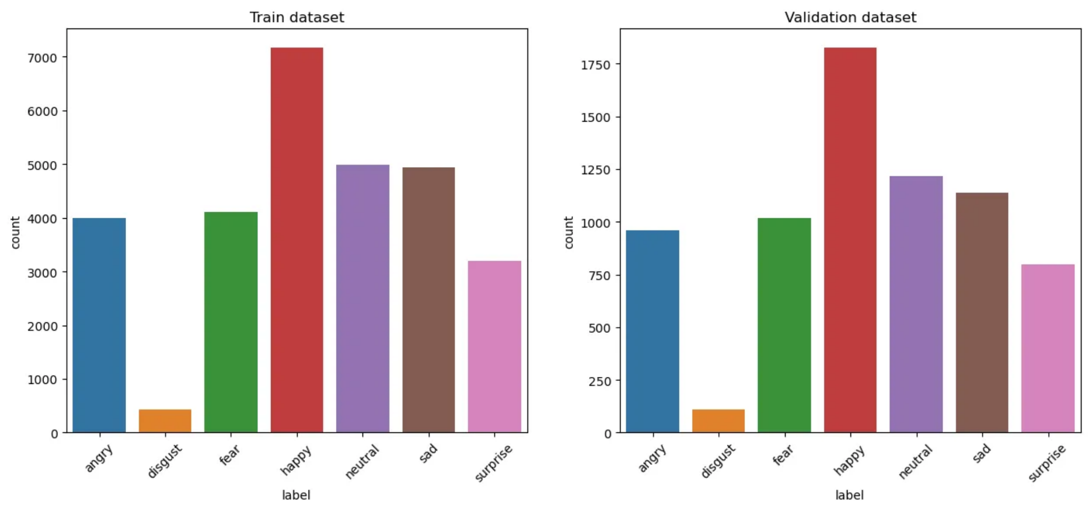
```
def visualize_data(df,n_rows,n_cols):
    plt.figure(figsize=(15,16))
    for i in range(n_rows*n_cols):
        index = np.random.randint(0, len(df))
        img = cv2.imread(df.src_path[index])
        class_nm = df.label[index]
        plt.subplot(n_rows, n_cols, i+1)
        plt.imshow(img)
        plt.title(class_nm)
    plt.show()
```

```
visualize_data(df_train, 5, 4)
```



Visualize data distribution using matplotlib charts

```
plt.figure(figsize=(15,6))
# training dataset
plt.subplot(1,2,1)
sns.countplot(data=df_train.sort_values("label"),x="label")
plt.title("Train dataset")
plt.xticks(rotation = 45)
# validation dataset
plt.subplot(1,2,2)
sns.countplot(data=df_valid.sort_values("label"),x="label")
plt.title("Validation dataset")
plt.xticks(rotation = 45)

plt.show()
```



We can see that the *disgust* class contains the least number of images. One hypothesis is that this will contribute to lower validation accuracy for this class. Roughly, the disgust class makes up about 1.5% of the entire dataset of 7 classes for training.

Label encoding

For the multiclass classification model, we need to convert the target labels to *numbers* instead of *strings.* Hence, we will use the label encoding technique for this.

```
from sklearn.preprocessing import LabelEncoder

Le = LabelEncoder()
df_train["label"] = Le.fit_transform(df_train["label"])

df_valid["label"] = Le.transform(df_valid["label"])
df_train["label"].value_counts()
```

One-Hot Encoding

After label encoding, we need to do one hot encoding as well. Here, for each row i.e. image, we will have a binary label representing which class it belongs to. Only the true class will be 1, rest labels will be 0.

```
labels_train = tf.keras.utils.to_categorical(df_train["label"])
```

```
labels_valid = tf.keras.utils.to_categorical(df_valid["label"])
```

The label for each row looks something like this -

```
[0., 0., 0., 0., 0., 0., 1.]
```

Load Images to Numpy Array format

We will now load the dataset as a numpy array. We'll be reading the images one by one and store them in an array format to feed to the neural network.

```
def get_features(items):
    feats = []
    for item in items:
        image = load_img(item, color_mode="grayscale")
        image = np.array(image)
        feats.append(image)
    feats = np.array(feats)
    feats = feats.reshape(len(feats), 48, 48, 1)
    return feats
```

```
features_train = get_features(df_train['src_path'])
features_valid = get_features(df_valid['src_path'])
```

To analyze the amount of time taken to load the dataset, we can use the time library from python, like this —

```
start = time.time()
features_train = get_features(df_train['src_path'])
stop = time.time()
print(f"Feature extraction time for training: {round(stop - start,2)} seconds")
```

```
Feature extraction time for training: 242.47 seconds
```

Data Normalization

We need to normalize the data on a scale of 0 to 1 so that data processing and learning are easy for the neural network.

An image is basically an array of numbers. In our case, it is a 48*48 numbers array. Each value ranges between 0–255. So we'll normalize each image by dividing each number by the maximum.

```
x_train = features_train/255.0
x_valid = features_valid/255.0
```

Model Training

I will be tweaking some hyperparameters of neural network architecture like — Number of layers, Batch size, Epochs, and Number of neurons in each layer.

To compare the best output, I will save these model instances in a dictionary to compare later.

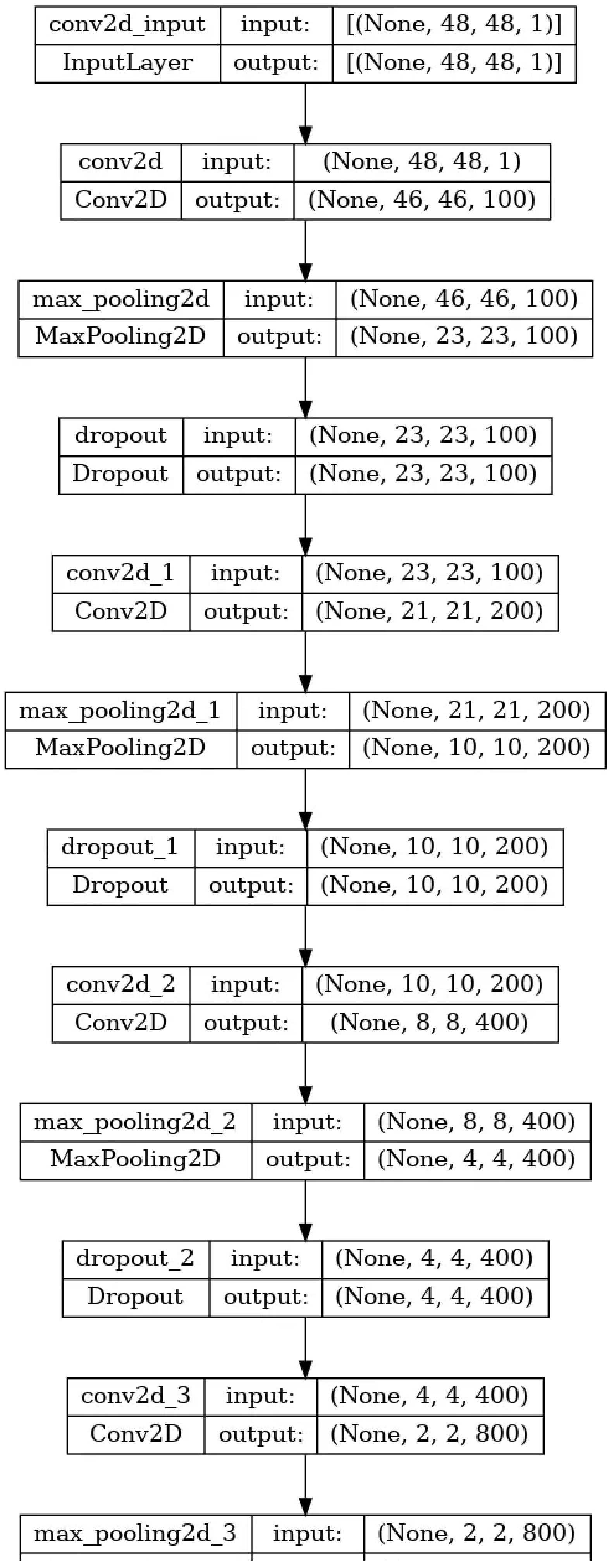
```
perf_stats = {'model':[],'history':[],'batch_size':[],'epochs':[]}
```

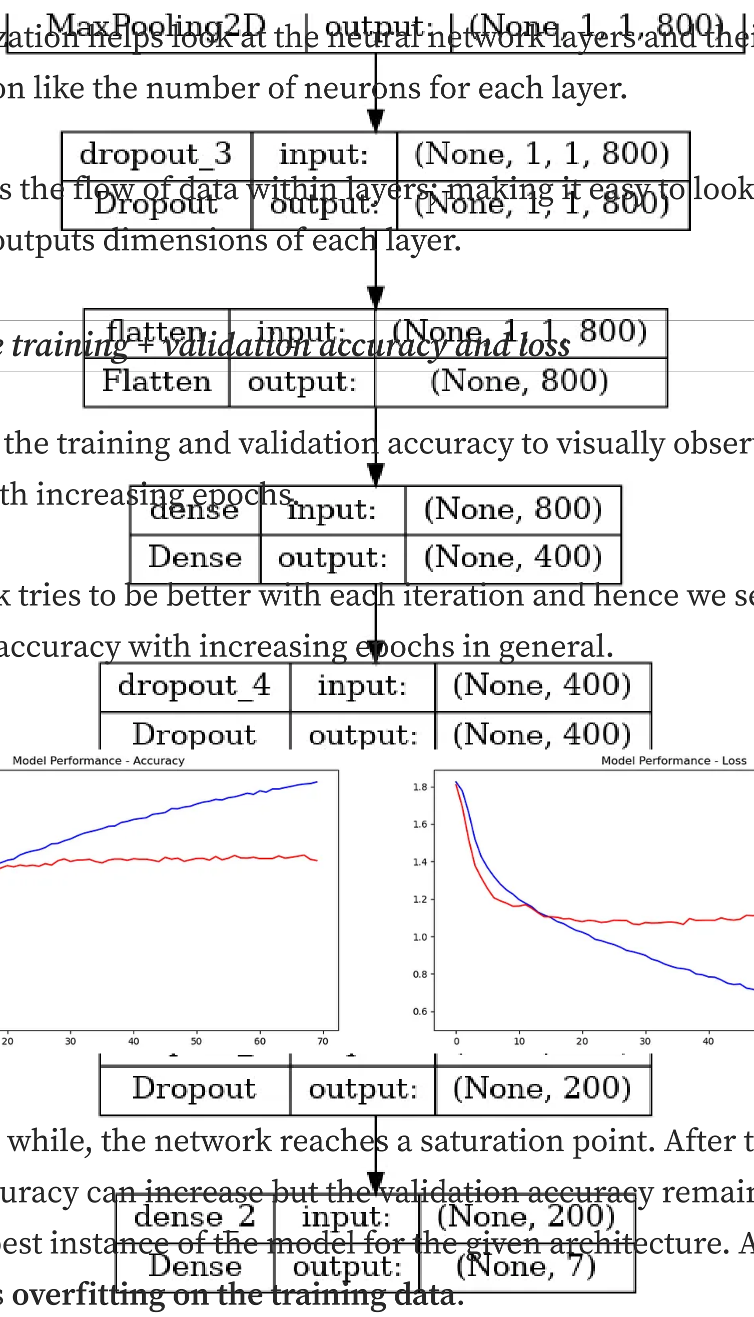
Model 1 — Architure

```
model = Sequential()
# convolutional layers
model.add(Conv2D(100, kernel_size=(3,3), activation='relu', input_shape=(48,48,1
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.3))

model.add(Conv2D(200, kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.3))
model.add(Conv2D(400, kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.3))

model.add(Conv2D(800, kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.3))

model.add(Flatten())

# fully connected layers
model.add(Dense(400, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(200, activation='relu'))
model.add(Dropout(0.3))
# output layer
model.add(Dense(len(labels_train[0]), activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics='accura
```

```
batch_size = 100
epochs = 70

# train the model
start = time.time()
history = model.fit(x=x_train, y=labels_train, batch_size=batch_size, epochs=epo
stop = time.time()
print(f"Model training time: {round(stop - start,2)} seconds")

perf_stats['model'].append(model)
perf_stats['history'].append(history)
perf_stats['batch_size'].append(batch_size)
perf_stats['epochs'].append(epochs)
```

We can also plot the model architecture in an image using tensorflow's built-in package


conv2d_input | input: | [(None, 48, 48, 1)]
InputLayer | output: | [(None, 48, 48, 1)]
conv2d | input: | (None, 48, 48, 1)
Conv2D | output: | (None, 46, 46, 100)
max_pooling2d | input: | (None, 46, 46, 100)
MaxPooling2D | output: | (None, 23, 23, 100)
dropout | input: | (None, 23, 23, 100)
Dropout | output: | (None, 23, 23, 100)
conv2d_1 | input: | (None, 23, 23, 100)
Conv2D | output: | (None, 21, 21, 200)
max_pooling2d_1 | input: | (None, 21, 21, 200)
MaxPooling2D | output: | (None, 10, 10, 200)
dropout_1 | input: | (None, 10, 10, 200)
Dropout | output: | (None, 10, 10, 200)
conv2d_2 | input: | (None, 10, 10, 200)
Conv2D | output: | (None, 8, 8, 400)
max_pooling2d_2 | input: | (None, 8, 8, 400)
MaxPooling2D | output: | (None, 4, 4, 400)
dropout_2 | input: | (None, 4, 4, 400)
Dropout | output: | (None, 4, 4, 400)
conv2d_3 | input: | (None, 4, 4, 400)
Conv2D | output: | (None, 2, 2, 800)
max_pooling2d_3 | input: | (None, 2, 2, 800)



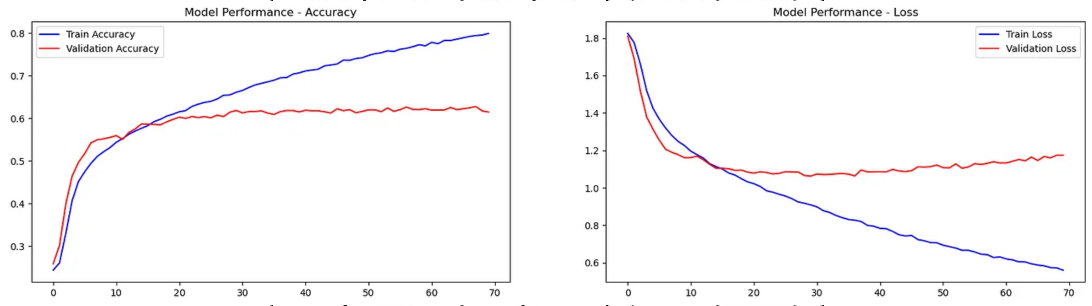
This visualization helps look at the neural network layers and their configuration like the number of neurons for each layer.

It also shows the flow of data within layers; making it easy to look at the inputs and outputs dimensions of each layer.

Plotting the training + validation accuracy and loss

We can plot the training and validation accuracy to visually observe the accuracy with increasing epochs.

The network tries to be better with each iteration and hence we see the increase in accuracy with increasing epochs in general.



After a little while, the network reaches a saturation point. After this point training accuracy can increase but the validation accuracy remains same. This is the best instance of the model for the given architecture. After this, the model is **overfitting on the training data**.

In the image above, the best model instance is where the blue and red lines meet. We can observe that after this point, the blue line i.e. training accuracy keeps on increasing but the red line i.e. validation accuracy remains consistent. This is **one of the ways to detect overfitting** in the network.

Here, we can say that the **model is overfitting after 20 epochs**. So training after that is not beneficial.

Tweak model architecuture

Once we loosely define an architecture to train, we can experiment with the hyperparameters.

In this case, I have experimented with —

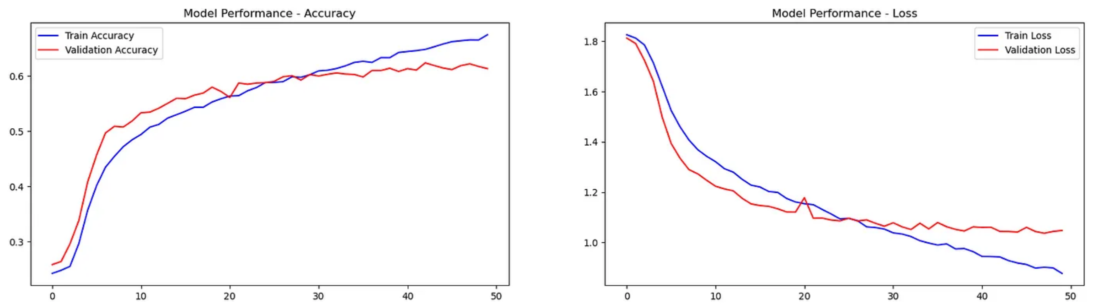
- Number of layers
- Number of neurons in each layer
- Training batch size
- Number of epochs for training

```
model = Sequential()
# convolutional layers
model.add(Conv2D(200, kernel_size=(3,3), activation='relu', input_shape=(48,48,1
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.4))

model.add(Conv2D(400, kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.4))

model.add(Conv2D(800, kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.4))

model.add(Conv2D(1600, kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.4))

model.add(Flatten())
# fully connected layers
model.add(Dense(800, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(400, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(200, activation='relu'))
model.add(Dropout(0.2))
# output layer
model.add(Dense(len(labels_train[0]), activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics='accura

batch_size = 200
epochs = 50

# train the model
start = time.time()

history = model.fit(x=x_train, y=labels_train, batch_size=batch_size, epochs=epo

stop = time.time()
print(f"Model training time: {round(stop - start,2)} seconds")

perf_stats['model'].append(model)
perf_stats['history'].append(history)
perf_stats['batch_size'].append(batch_size)
perf_stats['epochs'].append(epochs)
```

This model architecture also provides a similar kind of accuracy i.e. around 60% for validation.

But the overfitting is very less in this case.

Here is the accuracy and loss plot for this model —



As we can see in the image, the model is not highly overfitting. Unlike the previous model, this one is **not overfitting till 30 epochs** clearly. Even after that, the training and validation accuracies are pretty close.

This indicates that the model is learning well with each iteration.

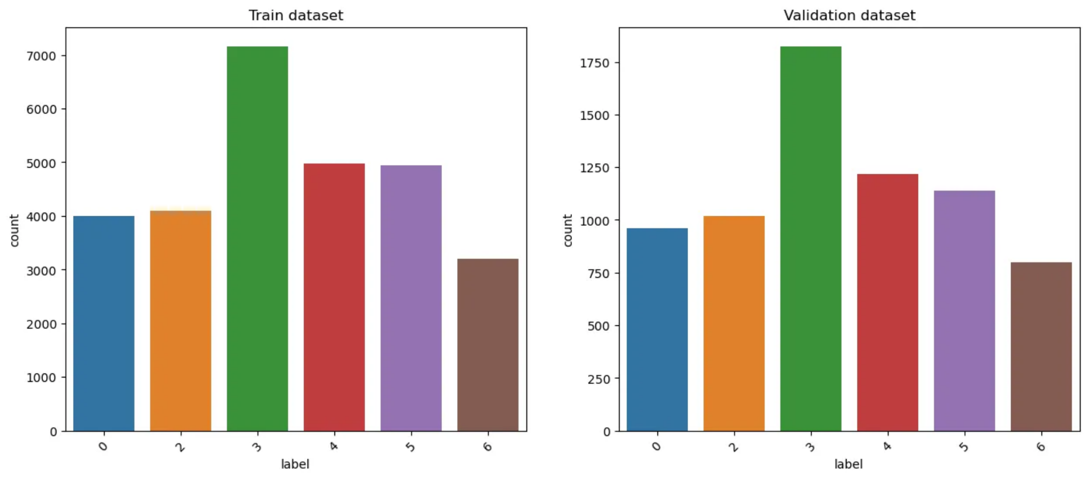
> *Experiment — Try training without disgust class*

As mentioned above, the *disgust* class contains the least number of images to train. The entire class only constitutes 1.5% of the entire dataset.

Let's find out if excluding this class from training helps in the validation accuracy with a similar model architecture.

```
## removing disgust class, because it only contains ~1.5% of the data, which is
df_train_new = df_train[df_train.label!=1].reset_index(drop=True)
df_valid_new = df_valid[df_valid.label!=1].reset_index(drop=True)

x_train_new = x_train[df_train.label!=1]
x_valid_new = x_valid[df_valid.label!=1]
```

After removing the disgust class, the data distribution looks something like the following —



Just like earlier, we need to convert the labels into one-hot encoded labels. This time, since we removed one of the classes, we should have an array of size 6 for each row, where only the true label is 1, rest are 0.

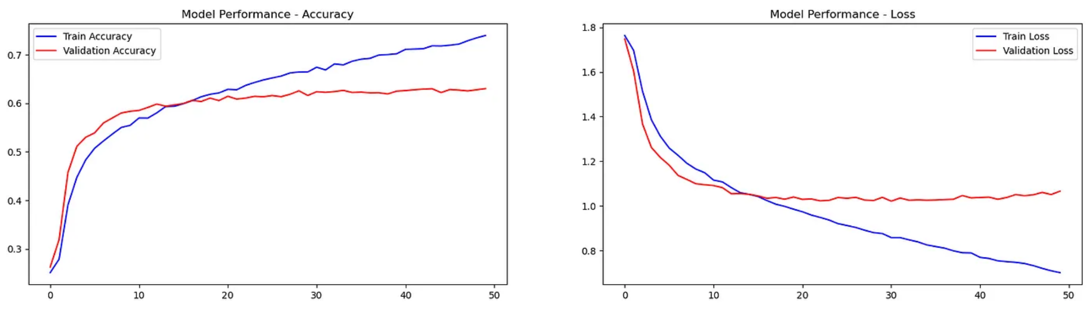
After this, the shape of our dataset is —

```
df_train_new.shape, x_train_new.shape
```

```
((28385, 2), (28385, 48, 48, 1))
```

The model architecture is the same as we used earlier as Model 1.

The training accuracy graph looks like the following —

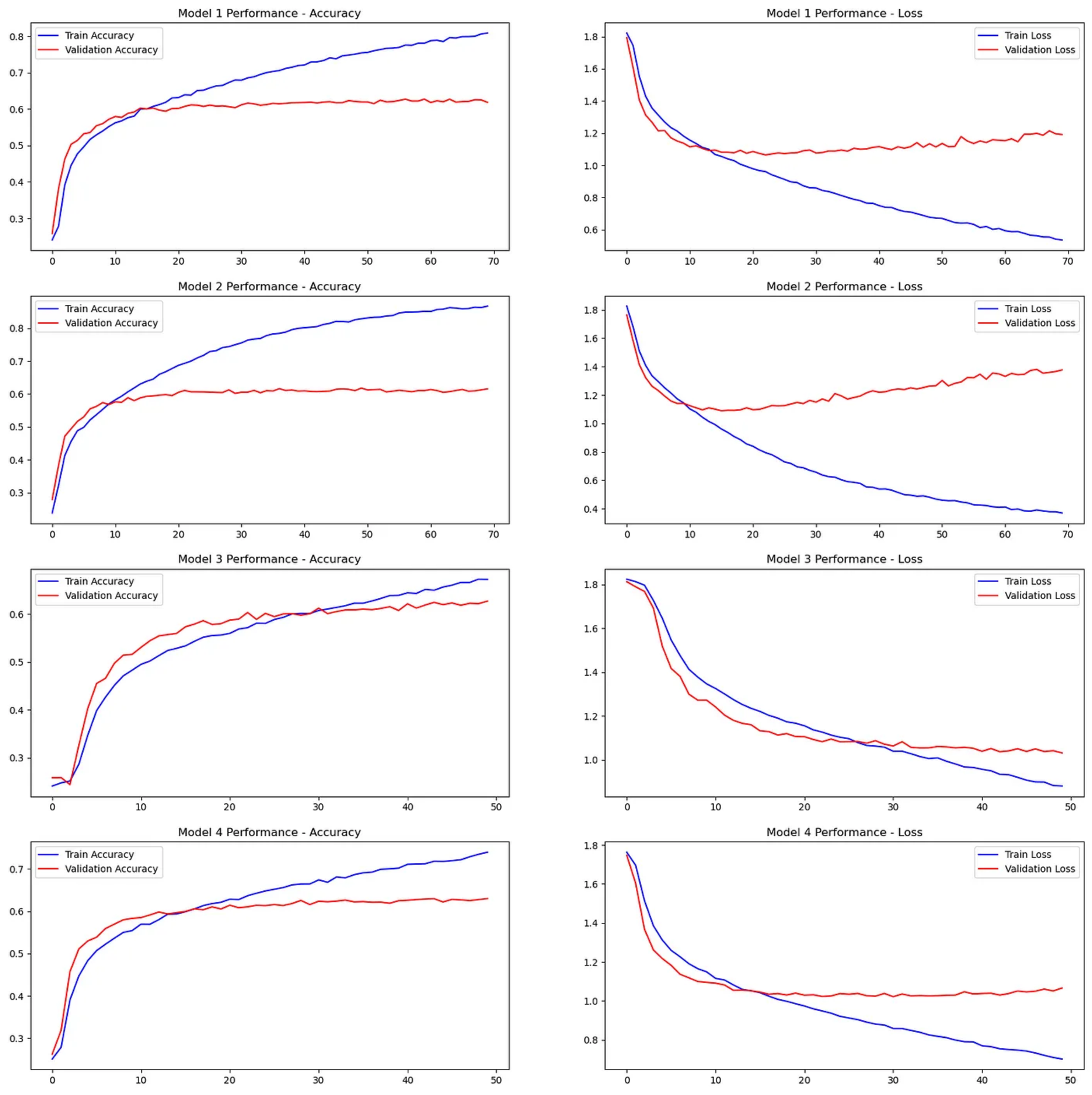


My observation here is — there was no effect of removing disgust class. The model architecture behaves the same way. The validation accuracy is also similar to previous attempts i.e. ~63% at best.

Comparison of all models

As discussed earlier, we have been saving each model configuration with its performance in a dictionary. We can analyze and compare the performance of each model at this point.

Visual comparison of accuracy and loss chart is —



Looking at this, we can decide which model is the least overfitting and has maximum accuracy.

According to my observation — It turns out that Model 3 is the best model.

The model configuration is saved into the respective dictionary item with its history, training batch size, and epochs.

We can also plot the model configuration as we did earlier, using the tensorflow's built-in *plot_model* package.

Save model

Finally, we can save the model instance to deploy it anywhere like AWS or a custom API endpoint.

We use the python library *pickle* for this.

```
# Save Model
model.save("FacialExpressionModel.h5")
```

```
# Save Label Encoder
import pickle

def save_object(obj , name):
    pickle_obj = open(f"{name}.pck","wb")
    pickle.dump(obj, pickle_obj)
    pickle_obj.close()
```

```
save_object(Le, "LabelEncoder")
```

We should also be saving the Label encoder instance, as it tells us which encoded label belongs to which class. This will help if we had any more data as we will need to encode the labels in the same way to test this algorithm.

Contribution

In this project, I have referred to the Kaggle challenge notebook available at this link — https://www.kaggle.com/datasets/samaneheslamifar/facial-emotion-expressions

My major contribution is regarding —

- Number of network layers
- Number of neurons in each layer
- Size of input data i.e. (48*48*1)
- Training Batch Size
- Number of iterations — Epochs
- Performance Comparison

Technical Challenges

While developing this — I faced some technical issues while designing the neural network layers.

- I got many errors regarding the input and output dimensions
- Used StackOverflow to solve issues
- While designing the number of layers, it is difficult to decide how many are sufficient

Conclusion

In this project, I explored the mechanism to design a neural network architecture and how to evaluate it. I understood the factors which affect the model performance as well. And I also realized the importance of hyperparameters in a typical machine-learning problem

Facial Expressions Neural Networks