# Facial expression – Building image classifier

Classifying an image into a particular class or category based on its content is known as image classification. In order to accomplish this, a machine learning model must be trained to recognize specific patterns or characteristics in an image related to a particular class or category.

We would want a dataset of labeled images representing each emotion to train an image classifier that can identify several emotions, including happy, angry, sad, fear, disgust, neutral, and surprise.

The Facial Emotion Expressions dataset available at Kaggle, which consists of 48 x 48-pixel grayscale tagged images of human faces exhibiting various emotions, is one such dataset.

**My objective is to develop a neural network by utilizing this dataset and subsequently train it to determine an image's emotional category.**

I have taken the reference of the Kaggle notebook (Link) attached to the dataset which has a **validation_accuracy of roughly 25.7%** as can be seen from the output of the notebook:

```
Epoch 1/8
450/450 [==============================] - ETA: 0s - loss: 1.4179 - accuracy: 0.7967 - precision: 0.8458 - recall: 0.7467
Epoch 1: val_loss improved from inf to 9.55412, saving model to best_weights.h5
450/450 [==============================] - 78s 172ms/step - loss: 1.4179 - accuracy: 0.7967 - precision: 0.8458 - recall: 0.7467
- val_loss: 9.5541 - val_accuracy: 0.2562 - val_precision: 0.2572 - val_recall: 0.2560
Epoch 2/8
450/450 [==============================] - ETA: 0s - loss: 1.4314 - accuracy: 0.7947 - precision: 0.8604 - recall: 0.7414
Epoch 2: val_loss did not improve from 9.55412
450/450 [==============================] - 77s 171ms/step - loss: 1.4314 - accuracy: 0.7947 - precision: 0.8604 - recall: 0.7414
- val_loss: 30.8738 - val_accuracy: 0.2562 - val_precision: 0.2563 - val_recall: 0.2562
Epoch 3/8
450/450 [==============================] - ETA: 0s - loss: 1.3361 - accuracy: 0.8006 - precision: 0.8605 - recall: 0.7461
Epoch 3: val_loss did not improve from 9.55412
450/450 [==============================] - 77s 171ms/step - loss: 1.3361 - accuracy: 0.8006 - precision: 0.8605 - recall: 0.7461
- val_loss: 13.9075 - val_accuracy: 0.2570 - val_precision: 0.2602 - val_recall: 0.2558
Epoch 4/8
450/450 [==============================] - ETA: 0s - loss: 1.4208 - accuracy: 0.8221 - precision: 0.8661 - recall: 0.7742
Epoch 4: val_loss did not improve from 9.55412
450/450 [==============================] - 77s 170ms/step - loss: 1.4208 - accuracy: 0.8221 - precision: 0.8661 - recall: 0.7742
- val_loss: 11.5059 - val_accuracy: 0.2558 - val_precision: 0.2559 - val_recall: 0.2558
Epoch 5/8
450/450 [==============================] - ETA: 0s - loss: 1.3284 - accuracy: 0.8200 - precision: 0.8617 - recall: 0.7677
Epoch 5: val_loss did not improve from 9.55412
450/450 [==============================] - 77s 171ms/step - loss: 1.3284 - accuracy: 0.8200 - precision: 0.8617 - recall: 0.7677
- val_loss: 11.3843 - val_accuracy: 0.2562 - val_precision: 0.2562 - val_recall: 0.2562
Epoch 6/8
450/450 [==============================] - ETA: 0s - loss: 1.5165 - accuracy: 0.7865 - precision: 0.8434 - recall: 0.7323
Epoch 6: val_loss did not improve from 9.55412
450/450 [==============================] - 77s 171ms/step - loss: 1.5165 - accuracy: 0.7865 - precision: 0.8434 - recall: 0.7323
- val_loss: 10.9685 - val_accuracy: 0.2565 - val_precision: 0.2565 - val_recall: 0.2565
```

The training accuracy of this model on the other hand is roughly 78.7%.
This model suffers from extreme overfitting as there is a huge difference of almost 50% between validation accuracy and training accuracy.

**I am going to attempt to improve the results.**

The steps we can take to train an image classifier to recognize emotions are as follows:

## Preparing the data:

First, we must download the Facial Emotion Expressions dataset from Kaggle and extract the files. (on your local machine)

However, when using google collab, you should download the files from Kaggle and zip images into an images.zip file, and upload it to the root of your google drive.

Once that is done, paste the code block given below at the top of the notebook

```
from google.colab import drive

drive.mount('/content/drive')



import zipfile

zip_ref = zipfile.ZipFile("/content/drive/MyDrive/images.zip", 'r')

zip_ref.extractall("/content/images")

zip_ref.close()

!pip install livelossplot
```

The images will be extracted in the colab runtime.

Now we'll proceed to understand the folder structure and contents.

```
# understanding the dataset structure

data_dir = 'images/'

training_path = data_dir+'train/'

validation_path = data_dir+'validation/'




# Training classes

print("--------- Training set description --------------")

print("S.No.\tSample Count\tClass\t")

# Printing distribution of each class in training set

for (i,expression) in enumerate(os.listdir(training_path)):
```

```
    print(str(i+1)+ "\t"+ str(len(os.listdir(training_path + expression))) + "\t\t" +
expression )



# Validation classes
print("\n--------- Validation set description --------------")
print("S.No.\tSample Count\tClass\t")
# Printing distribution of each class in training set
for (i,expression) in enumerate(os.listdir(validation_path)):
    print(str(i+1)+ "\t"+ str(len(os.listdir(validation_path + expression))) + "\t\t"
+ expression )
```

```
--------- Training set description ----------
S.No.   Sample Count    Class
1       7164            happy
2       4938            sad
3       436             disgust
4       4982            neutral
5       4103            fear
6       3993            angry
7       3205            surprise

--------- Validation set description ----------
S.No.   Sample Count    Class
1       1825            happy
2       1139            sad
3       111             disgust
4       1216            neutral
5       1018            fear
6       960             angry
7       797             surprise
```

## Preparing training and validation data:

```
def create_dataset(main_path):

    label_encoder = LabelEncoder()


    df = {"img":[],"img_class":[]}

    for class_names in os.listdir(main_path):

        for img_path in glob.glob(f"{main_path}/{class_names}/*"):

            img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
```

```
            img = cv2.resize(img, (48, 48))

            img = img[..., np.newaxis] / 255.0

            df["img"].append(img)

            df["img_class"].append(class_names)

    df["img"] = np.array(df["img"], dtype='float32')

    df["img_class"] = label_encoder.fit_transform(df["img_class"])

    df["img_class"] = tf.keras.utils.to_categorical(df["img_class"])

    return df["img"],df["img_class"]



training_dataset, training_labels = create_dataset(training_path[:len(training_path)-

1])

validation_dataset, validation_labels  =
create_dataset(validation_path[:len(validation_path)-1])
```

**Instead of creating tensors out of the dataset (as in the referenced code), I have opted to build numpy array for storing image data and used one hot encoding as in reference to encode labels.**

The function `create_dataset` takes in a `main_path` parameter. The function then uses `os.listdir` to get a list of all the subdirectories (which represent the different classes of images) within the `main_path` directory. It then uses a nested loop to iterate over each subdirectory and all the images within each subdirectory.

For each image, the function reads the image using `cv2.imread`, converts it to grayscale, resizes it to 48×48 pixels, and normalizes the pixel values between 0 and 1. It then appends the preprocessed image and its corresponding class label to a dictionary called `df`.

Finally, the function converts the class labels into numeric values using `LabelEncoder`, and then converts these numeric labels into one-hot encoded vectors using `tf.keras.utils.to_categorical`. The function returns the preprocessed images and their corresponding one-hot encoded labels.

The code then calls `create_dataset` twice, passing in the `training_path` and `validation_path` directories as arguments. The resulting preprocessed images and their one-hot encoded labels are assigned to `training_dataset`, `training_labels`, `validation_dataset`, and `validation_labels`.

**Problem with Label Encoding:**

Label encoding assumes that the higher the categorical value, the better the category.

Supposing our model internally calculates average, then accordingly we get, 0+4 = 4/2 =2. This implies that: Average of Angry (0) and Neutral (4) is Fear (2). This is definitely a recipe for disaster. This model's prediction would have a lot of errors.

**Solution:**

We use one hot encoder to perform "binarization" of the category and include it as a feature to train the model.

For example, suppose you have a categorical variable "Color" with three categories: "Red," "Green," and "Blue." In one hot encoding, you would create three binary vectors, one for each category. The "Red" vector would be [1, 0, 0], the "Green" vector would be [0, 1, 0], and the "Blue" vector would be [0, 0, 1].

Reference: https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f

## Defining Constants:

```
# Defining constants

BATCH_SIZE = 64

LEARNING_RATE = 0.0001
```

## Using Augmentation Techniques to produce a variety of training examples

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator



# Data Augmentation

data_augmentation = ImageDataGenerator(

    rotation_range=10,

    width_shift_range=0.1,

    height_shift_range=0.1,

    shear_range=0.1,

    zoom_range=0.1,

    horizontal_flip=True,

    vertical_flip=True,

    fill_mode='nearest'
```

```
)
```

**I have been inspired to use data augmentation techniques from the reference Kaggle notebook.**

By using this `ImageDataGenerator` object with the `fit` method of a Keras model, the training data can be augmented on-the-fly during training, which can help to improve the robustness and generalization of the trained model.

By preparing the dataset in this way, the model is more likely to generalize well to new, unseen data, as it has been trained on a diverse set of augmented images.

## Building Model

**Instead of using EfficientNetB2 as a network layer like in the referred code, I elected to bypass it and create my own network layers** as was done by the author in this post.

```
from tensorflow.keras.layers import Dense,  Dropout,Flatten, Conv2D

from tensorflow.keras.layers import BatchNormalization, Activation, MaxPooling2D

from tensorflow.keras.models import Sequential

from tensorflow.keras.optimizers import Adam


from keras.models import Sequential

from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout


from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense,

BatchNormalization, Activation

from tensorflow.keras.optimizers import Adam




model = Sequential()


# 1 - Convolution

model.add(Conv2D(128, (3, 3),  input_shape=(48, 48, 1)))
```

```
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))

# 2nd Convolution layer
model.add(Conv2D(256, (3, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))

# 3rd Convolution layer
model.add(Conv2D(128, (3, 3) ))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))

# Flattening
model.add(Flatten())

# Fully connected layer 1st layer
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))

# Output layer
model.add(Dense(7, activation='softmax'))

opt = Adam(learning_rate=LEARNING_RATE)
```

```
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])

model.summary()
```

This is a Convolutional Neural Network (CNN) image classification model. It is built with TensorFlow's Keras API. The model is made up of the following layers:

1. **Input layer**: The input images are grayscale and 48×48 pixels in size. (1 channel).
2. The first convolutional layer contains 128 3×3 filters. The **'relu'** activation function is used to introduce nonlinearity into the model. To speed up training, batch normalization is employed, and max pooling is used to reduce the size of the feature maps.
3. The second convolutional layer contains 256 3×3 filters. It, like the first convolutional layer, employs the 'relu' activation function, batch normalization, and max pooling.
4. 3rd Convolutional Layer: This layer contains 128 3×3 filters. Like the previous convolutional layers, it employs the'relu' activation function, batch normalization, and max pooling.
5. Flattening layer: This layer converts the previous layer's output to a 1D array.
6. The first fully linked layer has 128 neurons and employs the 'relu' activation function. To prevent overfitting, a dropout rate of 0.5 is utilized.
7. Output layer: This layer has 7 neurons (equivalent to 7 classes) and employs the 'softmax' activation function, which is often employed for multiclass classification applications.

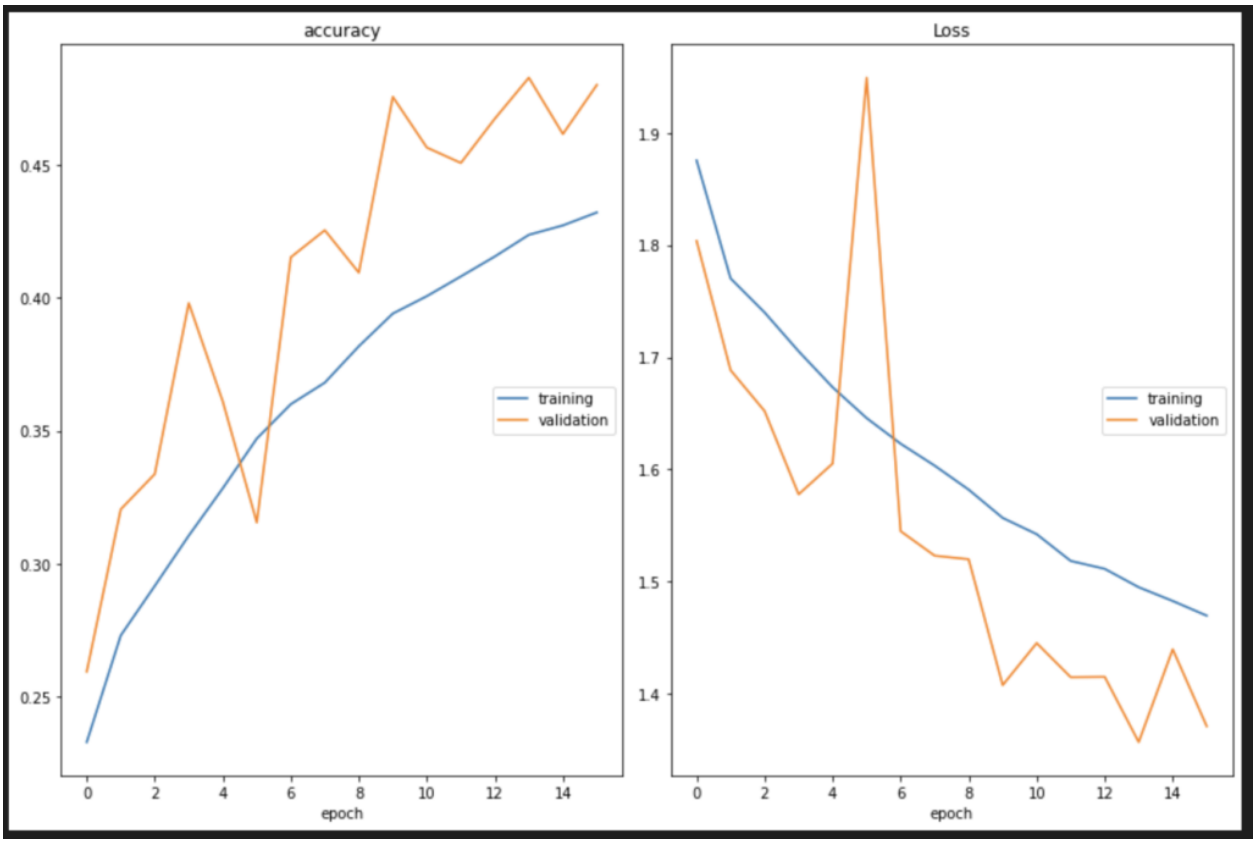
**Defining callbacks to save weights of neurons with best accuracy**

```
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 46, 46, 128)       1280

 batch_normalization_3 (Batc  (None, 46, 46, 128)      512
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 23, 23, 128)      0
 2D)

 activation_4 (Activation)   (None, 23, 23, 128)       0

 conv2d_4 (Conv2D)           (None, 21, 21, 256)       295168

 batch_normalization_4 (Batc  (None, 21, 21, 256)      1024
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 10, 10, 256)      0
 2D)

 activation_5 (Activation)   (None, 10, 10, 256)       0

 conv2d_5 (Conv2D)           (None, 8, 8, 128)         295040
...
Total params: 856,711
Trainable params: 855,687
Non-trainable params: 1,024
```

```
early_stopping=tf.keras.callbacks.EarlyStopping(monitor="accuracy",patience=3,mode="a

uto")

checkpoint =
tf.keras.callbacks.ModelCheckpoint("best_weights.h5",verbose=1,save_best_only=True,sa
ve_weights_only = True)
```

## Training Model

```
history = model.fit(
    data_augmentation.flow(training_dataset, training_labels, batch_size=BATCH_SIZE),
    epochs=100,
    callbacks=[PlotLossesCallback(),checkpoint, early_stopping],
    validation_data=(validation_dataset,validation_labels)
)
```




**Statistics**

```
accuracy
        training                (min:    0.233, max:    0.432, cur:    0.432)
        validation              (min:    0.259, max:    0.483, cur:    0.480)
Loss
        training                (min:    1.470, max:    1.875, cur:    1.470)
        validation              (min:    1.357, max:    1.949, cur:    1.371)

Epoch 16: val_loss did not improve from 1.35694
451/451 [==============================] - 623s 1s/step - loss: 1.4696 - accuracy: 0.4323 - val_loss: 1.3710
```

**As a result, at the end of the exercise, I was able to raise the validation accuracy to up to 48.3%, which is nearly double the referred code.**

**Saving Model**

```
# Save Model

model.save("Model.h5")
```

# Variation-2

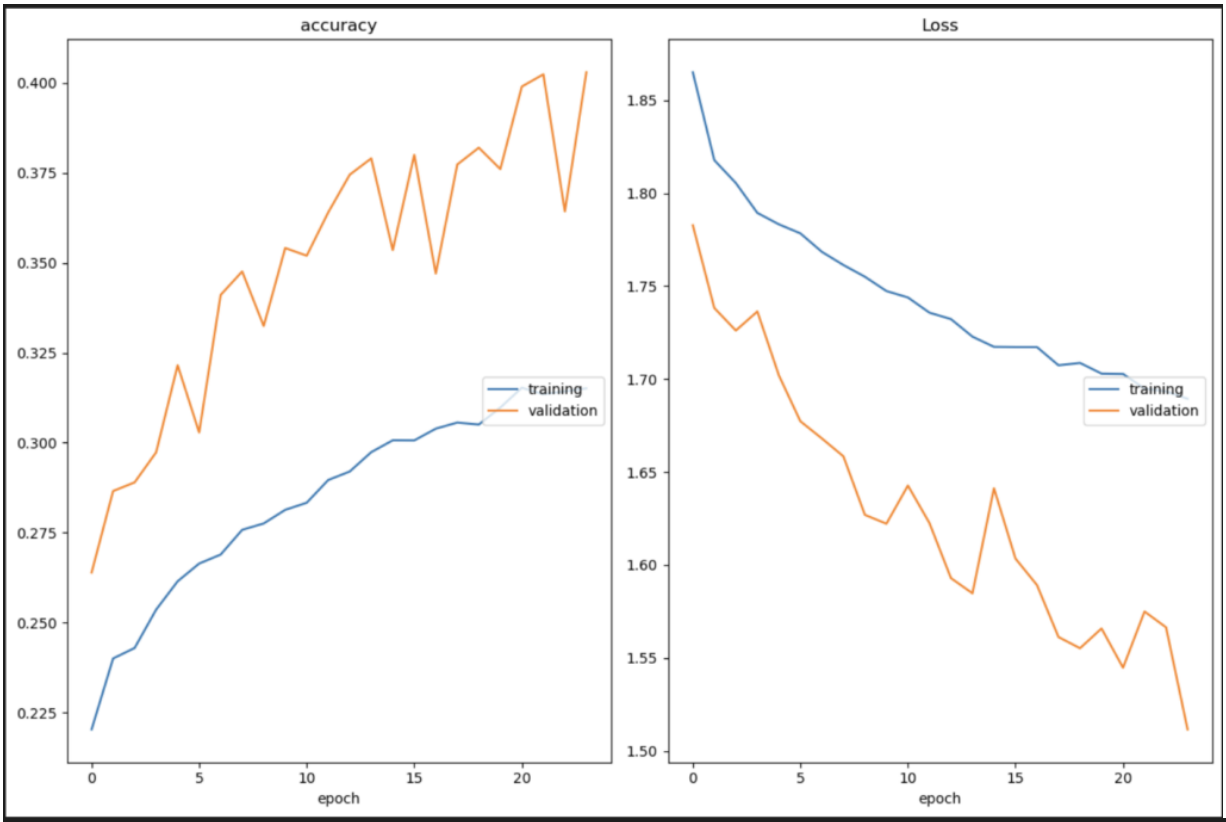
Having made changes to network topology:

1. Removed one convolutional layer
2. Reduced filter size of other conv2D layers:

```
model = Sequential()



# 1 - Convolution

model.add(Conv2D(32, (3, 3),  input_shape=(48, 48, 1)))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Activation('relu'))




# 2nd Convolution layer

model.add(Conv2D(64, (3, 3)))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model.add(Activation('relu'))



# Flattening
model.add(Flatten())

# Fully connected layer 1st layer
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))

# Output layer
model.add(Dense(7, activation='softmax'))

opt = Adam(learning_rate=LEARNING_RATE)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

The model converged earlier and could not go beyond roughly 40% validation accuracy:




```
accuracy
        training                (min:    0.220, max:    0.315, cur:    0.315)
        validation              (min:    0.264, max:    0.403, cur:    0.403)
Loss
        training                (min:    1.689, max:    1.865, cur:    1.689)
        validation              (min:    1.512, max:    1.783, cur:    1.512)

Epoch 24: val_loss improved from 1.54474 to 1.51155, saving model to best_weights.h5
451/451 [==============================] - 35s 78ms/step - loss: 1.6894 - accuracy: 0.3151 - val_loss: 1.5116
```

## Challenges faced

1. **Dataset preparation**: For the dataset preparation, the Kaggle notebook referred to utilized TensorFlow libraries to decode image data with RGB channel. However, for our particular use case, I found it unnecessary and opted for GRAYSCALE images instead. Despite spending hours using image data inside tensors and experimenting with various models, I had no success. Eventually, I switched to using cv2 to decode images as numpy float arrays, which immediately produced favorable results.
2. **Hyperparameters tuning**: My initial attempts at training the network resulted in poor performance, and I quickly learned that the hyperparameters played a crucial role in determining the network's accuracy. I spent several days adjusting the learning rate, batch size, and the number of epochs to finally be able to find the optimal hyperparameters and achieve the desired accuracy.

**References:**

1. https://www.kaggle.com/code/anand1994sp/facial-expression/notebook
2. https://towardsdatascience.com/image-classification-in-data-science-422855878d2a
3. https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f