USTH Deep learning research: Facial Expression recognition

LE Huy Duc

January 2021

1 Introduction

In this course, we work on the problem of Facial Expression Recognition. Given an image of a person, the program must classify it into one of the 7 emotions: Neutral, Anger, Disgust, Fear, Happiness, Sadness, Surprise.

To solve this, we apply deep learning, specifically Convolutional Neural Network (CNN). I will test and fine-tune 2 models: my simple CNN model (base model) and VGG16 with transfer learning. I measure the performance of each model with and without data augmentation.

2 Dataset

We use FER[1] 2013 dataset. It is provided in a .csv file by Kaggle.

The dataset contains 35887 images in total. We will use 28709 for training, 3589 for validation, and 3589 for testing. All images are grayscale, and have size 48×48.

Note that this dataset is extremely difficult to get a high accuracy. A survey from 2018 [2] shows that even the state-of-the-art models only reach about 70% to 75% test accuracy. [2]

Figure 1: Examples of images in the FER 2013 dataset

3 Project implementation

In this report, I focus on explaining the ideas of the methods that I use for the project as well as the general workflow of the program. To see the implementation details, please refer to the source code files that come with this report.

3.1 Reading the dataset

The FER 2013 dataset is given in a *.csv* file. As usual, each line represents an instance, and it has 3 columns: emotion (ground truth label), pixel values (input features), and usage (training/validation/test set). We do not use the 3rd column, instead we use our own functions to split the train/validation/test sets.

The file can be read using Python's pandas and opencv-python library. The function to read the dataset is already provided to us, so we do not go into detail here.

[1] (**F**acial **E**xpression **R**ecognition)

[2] My baseline model achieves over 93% accuracy in the reduced dataset (that we used in the lectures). However, its accuracy in FER2013 is much lower.

3.2 Normalizing the dataset

It is a well-known that the input of a neural network should be normalized. The normalized range should be $-1..1$ or $0..1$. In this project, I use $-1..1$ range, it can be done using the formula:

$$x_{norm} = ((x/255) - 0.5) * 2 \tag{1}$$

Almost every machine learning courses or tutorials will perform input normalization (for example, Andrew Ng's machine learning course on Coursera). In general, this is done to reduce the effects of different data ranges (for example: area of a house (m^2) has range 10..2000, but the number of rooms has range 1..10), which means the network might ignore features with small values. Also, normalization speeds up learning and leads to a faster convergence, since all features have the same small range from $-1..1$. [1].

3.3 Split the dataset into training, validation, and testing sets

This step should be done before any machine learning prototyping. The training set is straight-forwards. The test set is used for final testing and only testing alone; basically, we do are not allowed to look at or use the test set anywhere else except for the final model evaluation. This is done to provide an unbiased evaluation of the model. The validation set is used for parameter fine-tuning [1].

To do this task, we use sklearn.model_model_selection.train_test_split. Note that we must use the stratified sampling option to ensure each class has a balance representation in the 3 sets.

```
x_train, x_test, y_train, y_test = train_test_split(x_total, y_total, test_size=0.2, stratify=y_total)
x_valid, x_test, y_valid, y_test = train_test_split(x_test, y_test, test_size=0.5, stratify=y_test)
```

Figure 2:

3.4 CNN models for facial recognition

Generally, the CNN models we use can be separated into 2 parts: the convolutional base, and the output dense layers. In this project, the output dense layers are the same in all experiments: one dense layer with 128 units and ReLU activation, and the output layer has 7 units (7 emotions) with softmax activation.

Note that the Rectified Linear Unit (ReLU) is used as the default activation function for all tested models in this project. It is extremely fast, reduces the problem of vanishing/exploding gradient, and is also used in popular models such as VGG16, Resnet.

3.4.1 Base CNN model

This is a basic CNN model that contains multiple layers as well as max-pooling layers. The idea is based on VGG16 network, where there are multiple groups of 2-3 connected CNN layers followed by max-pooling.

```
def BaseModel():
    model = Sequential()
    model.add(Conv2D(input_shape=input_shape,filters=64,kernel_size=(3,3),padding="same", activation="relu")
    model.add(Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(num_classes, activation='softmax'))
    return model
```

Figure 3: My base CNN model

3.4.2 VGG16 with transfer learning

I also wish to compare my baseline model with another popular architecture. I choose VGG16 because it is not too heavy to train, and its architecture is similar to my model.

```python
def VggModel():
    cnn_base = VGG16(include_top=False, input_shape=(48, 48, 3))
    cnn_base.trainable = True

    flat = Flatten()(cnn_base.layers[-1].output) # connect last layer of cnn base to flatten
    dense = Dense(128, activation='relu')(flat)
    drop = Dropout(0.3)(dense)
    output = Dense(7, activation='softmax')(drop)
    model = Model(inputs=cnn_base.inputs, outputs=output)
    return model
```

Figure 4: Pre-trained VGG model from Keras

Luckily, VGG16 does not need to be implemented from scratch. Keras already provides us with a pre-trained VGG16 model. The only thing we need to do is replace its top layers with our top layers.

There's one thing to note: since VGG16 requires 3-channels input images, we must convert our grayscale images into RGB images, where $R = G = B$.

3.5 Handling imbalance classes

Since the ratio of images of different classes are not equal, we use class-weighting to ensure the model does not underfit (gives too much attention to a class).

```python
y_train_label = np.argmax(y_train, axis=1)
balance = np.array([1 / (np.sum(y_train_label==i) / len(y_train)) for i in range(num_classes)])
balance = np.array([balance[i] / np.sum(balance) for i in range(num_classes)])
class_weights = {i: balance[i] for i in range(num_classes)}
```

Figure 5: Assigning class weights

3.6 Data augmentation

We use Keras's ImageDataGenerator for data augmentation. The operation we use are: rotation, horizontal/vertical shifting, and zooming.

The default batch size is 4. Larger batch size can be used to improve speed, however, I choose 4 to ensure that the training quality is good. Larger batch sizes might result in worse test error, as shown in [3].

Note that the rotation range and the shift ranges must be very small, since the size of the images are very small. In our testing, if $rotation\_range \geq 6$, it will cause underfitting, since corner pixels of the image are lost.

3.7 Compile and train the models

We use Adam optimizer, since this is a great default optimizer. The learning rate is 10^{-5}, and the batch size is 4.

We also use use the following callbacks: *EarlyStopping* and *ReduceLROnPlateau*. The first one stops the training if validation loss does not improve after several iterations. The latter reduces the learning rate when validation loss does not change

4 Result

4.1 General result

The table below shows the result of the baseline CNN model and the pre-trained VGG16 model when using default learning rate and batch size (10^{-5} and 4).

Model / DataAug	No	Yes
My CNN	55.75%	64.5%
VGG16	66.54%	68.32%

We can see that without data augmentation, the base model's accuracy is quite low, only 55.75%. However, with data augmentation, it reaches 64.5%, which is nearly as good as VGG16.

With data augmentation, VGG16 performs very well. It achieves 68.32% accuracy, which is only 5-7% lower than state-of-the-art methods [2].

You can see the confusion matrix of the my CNN model and of VGG16 below. We see that both models can classify correctly the facial expression, and they are not overfit.

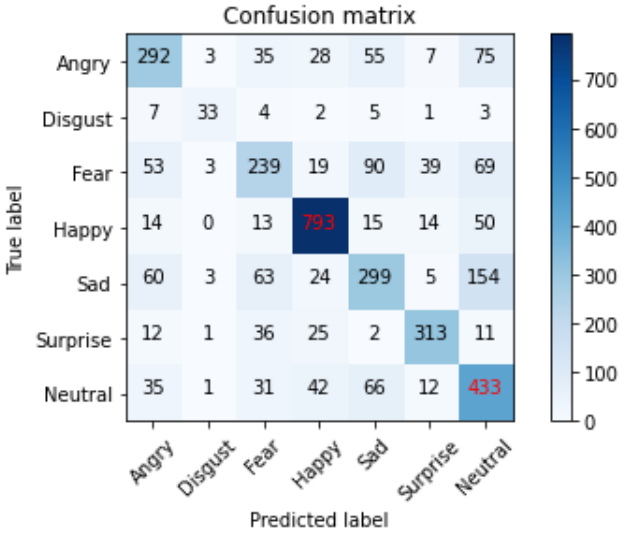

Figure 6: Confusion matrix of baseline model

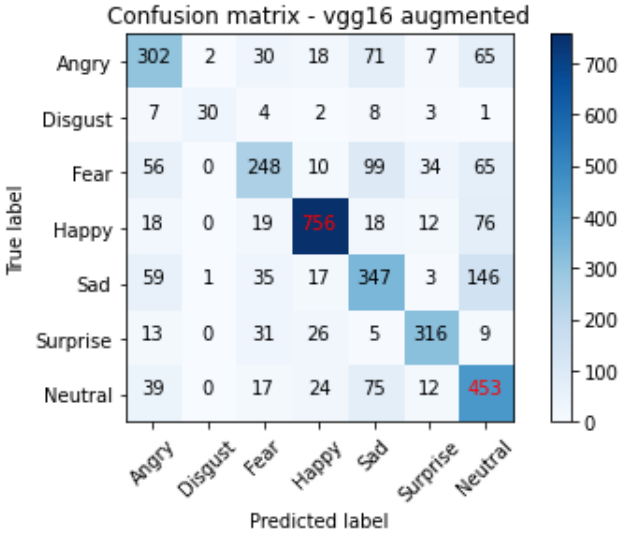

Figure 7: Confusion matrix of VGG16

4.2 Effect of batch size

I have also tested the models with $batch\_size = 32$. The result is as follow:

Model / DataAug	No	Yes
My CNN	44.89%	65.12%
VGG16	63.92%	65.78%

We see that the accuracy of both models are lower. In fact, if we train these two models on the small dataset (that we use during the lecture) with batch size 32, both will underfit. They will predict all classes with equal probability.

5 Conclusion

In this project, we have benchmarked and compared 2 CNN models: a simple CNN (baseline) and pre-trained VGG16 - using FER 2013 dataset. We also compared the performance uplift of each model when using data augmentation and showed the importance of batch size.

Overall, VGG16 with data augmentation performs quite well, achieving 68.3%. For future work, we can try combining facial landmarks, SVM, and CNN together to get a higher accuracy.

References

[1] Aurelien Geron. *HANDS-ON MACHINE LEARNING WITH SCIKIT-LEARN, KERAS, AND TENSORFLOW: CONCEPTS, TOOLS, AND TECHNIQUES*. O'REILLY MEDIA, 2019.

[2] Shan Li and Weihong Deng. Deep facial expression recognition: A survey. 2018.

[3] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2021.