Facial Emotion Recognition

March 27, 2023

The first step is to import the required libraries

[1]:

**import**

**numpy**

**as**

**np**

**import**

**pandas**

**as**

**pd**

**import**

**os**

**from**

**PIL**

**import**

Image

**import**

**tensorflow**

**as**

**tf**

**from**

**tensorflow**

**.**

**keras**

**import**

datasets, layers, models

The next step is to prepare the dataset for use. We first get the list of filenames for the train and validation datasets.

[2]: img\_dir = './images'

datasets

=

os

.

listdir(img\_dir)

emotions

=

os

.

listdir(

f

'

**{**

img\_dir

**}**

/train

'

)

The function get\_filenames creates a Pandas DataFrame containing the file path, label, and emotion index for each image in the given dataset. Two DataFrames are created for the training and validation sets.

[3]: **def** get\_filenames(dataset):

data = [] **for** i, e **in** enumerate(emotions):

**for** file **in** os.listdir(f'**{**img\_dir**}**/**{**dataset**}**/**{**e**}**'):

data.append({ "path": f'**{**img\_dir**}**/**{**dataset**}**/**{**e**}**/**{**file**}**', "label":␣

↪e, "emotion": i}) **return** pd.DataFrame(data)

df = get\_filenames('train').sample(frac=1) test\_df = get\_filenames('validation').sample(frac=1)

The training DataFrame is split into training and validation DataFrames.

[ ]:

validation\_df

=

df

.

sample(frac

=

0.2

)

train\_df

=

df

.

drop(validation\_df

.

index)

A sample of images from the training set is displayed for visual inspection.

[4]:

**import**

**matplotlib**

**.**

**pyplot**

**as**

**plt**

plt

.

figure(figsize

=

(

10

,

10

))

**for**

i

**in**

range

(

9

):

ax

=

plt

.

subplot(

3

,

3

,i

+

1

)

plt

.

imshow(plt

.

imread(train\_df

.

iloc[i][

'

path

'

])

, cmap

=

"

gray

"

)

plt

.

title(train\_df

.

iloc[i][

'

label

'

])

plt

.

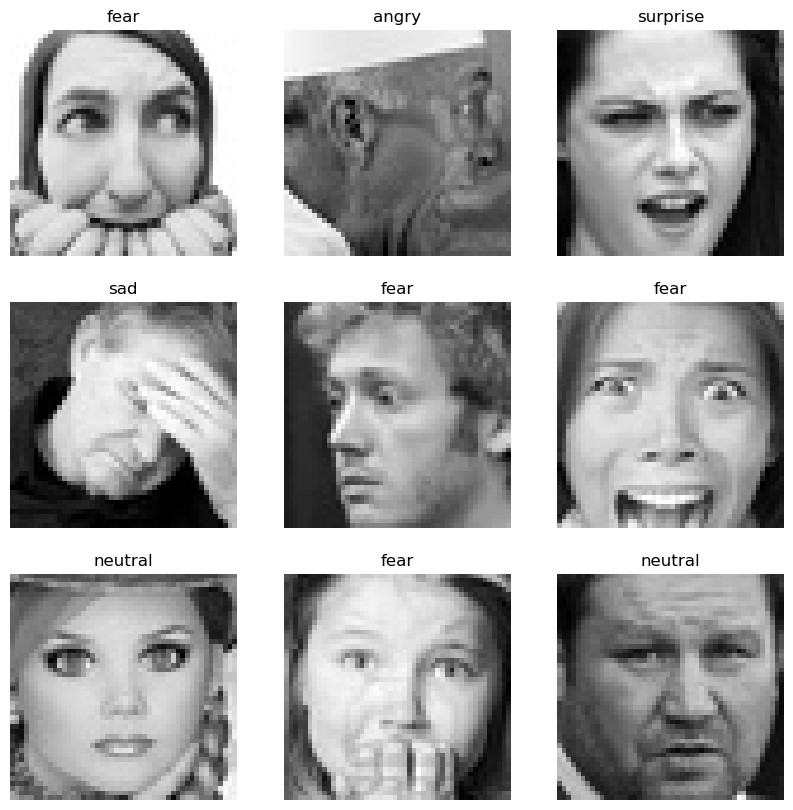
axis(

"

off

"

)



The load\_image function reads the image file at the given path, resizes it to 48x48 pixels, converts it to a NumPy array, and adds an additional dimension for the grayscale channel.

[5]:

**def**

load\_image

(

img\_path

):

image

=

Image

.

open(img\_path)

image

=

np

.

asarray(image

.

resize((

48

,

48

)))

image

=

np

.

expand\_dims(image, axis

=

-

1

)

**return**

image

The batch\_generator function generates batches of images and labels from the given DataFrame. It shuffles the data and yields batches of the specified size until it has gone through the entire DataFrame. If loop is set to True, it resets the generator and allows the same generator to be used for multiple epochs.

[6]:

**def**

batch\_generator

(

data, batch\_size

=

32

, loop

=

**True**

):

**while**

**True**

:

df

=

data

.

sample(frac

=

1

)

start

=

0

**while**

start

<

len

(

df

):

df\_batch

=

df[start: start

+

batch\_size]

images

=

np

.

stack(df\_batch[

'

path

'

]

.

apply(load\_image)

.

values)

images

=

tf

.

convert\_to\_tensor(images, dtype

=

tf

.

float32)

labels

=

df\_batch[

'

emotion

'

]

.

values

**yield**

images, labels

start

+

=

batch\_size

**if**

**not**

loop:

**break**

This model is a Convolutional Neural Network (CNN) with three convolutional layers followed by two fully connected (Dense) layers. The input shape of the model is (48, 48, 1), which corresponds to grayscale images with a size of 48x48 pixels.

The first convolutional layer has 32 filters of size (3, 3) and uses the ReLU activation function. The output is then max-pooled with a pooling window of size (2, 2).

The second convolutional layer has 64 filters of size (3, 3) and uses the ReLU activation function. The output is again max-pooled with a pooling window of size (2, 2).

The third convolutional layer has 64 filters of size (3, 3) and uses the ReLU activation function.

The output from the convolutional layers is flattened into a 1-dimensional array and fed into a fully connected layer with 64 neurons, which uses the ReLU activation function.

Finally, there is a fully connected output layer with 7 neurons, which corresponds to the number of classes (emotions) being predicted. No activation function is specified for this output layer, which means that the output values will be logits (raw predictions) that can be passed through a softmax function to obtain class probabilities.

[7]: model = models.Sequential() model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(48, 48, 1))) model.add(layers.MaxPooling2D((2, 2))) model.add(layers.Conv2D(64, (3, 3), activation='relu')) model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu')) model.add(layers.Flatten()) model.add(layers.Dense(64, activation='relu')) model.add(layers.Dense(7))

Next the model is trained. We use early stopping, which is a technique used to prevent overfitting of the model by monitoring the validation loss and stopping the training process when the validation loss stops decreasing. It is specified with a patience parameter which is the number of epochs with no improvement after which training will be stopped. The restore\_best\_weights parameter is set to True to restore the weights of the model corresponding to the epoch with the lowest validation loss.

The code then compiles the Keras model with the specified optimizer, loss function, and metrics. The optimizer used here is Adam and the loss function used is sparse categorical cross-entropy.

The model is then trained on the training data using the fit() method of the Keras model. The training is done for 25 epochs with a batch size of 64. The validation data is also used during training to monitor the model’s performance on data it has not seen during training. The early\_stop callback function is passed to the model’s fit method to prevent overfitting.

After the model is trained, it is evaluated on the test data using the evaluate() method of the Keras model. The test loss and accuracy are then printed to the console.

[8]:

**from**

**tensorflow**

**.**

**keras**

**.**

**callbacks**

**import**

EarlyStopping

early\_stop

=

EarlyStopping(monitor

=

'

val\_loss

'

,

patience

=

10

,

restore\_best\_weights

=

**True**

,

mode

=

'

min

'

)

*# Compile the model*

model

.

compile(optimizer

=

'

adam

'

,

loss

=

tf

.

keras

.

losses

.

↪

SparseCategoricalCrossentropy(from\_logits

=

**True**

)

,

metrics

=

[

'

accuracy

'

])

history

=

model

.

fit(

batch\_generator(train\_df,

64

)

,

steps\_per\_epoch

=

len

(

train\_df

)

/

/

64

,

epochs

=

25

,

validation\_data

=

batch\_generator(validation\_df,

64

)

,

validation\_steps

=

len

(

validation\_df

)

/

/

64

,

callbacks

=

[

early\_stop],

)

test\_loss, test\_acc

=

model

.

evaluate(batch\_generator(test\_df,

1024

,

**False**

)

,

␣

↪

verbose

=

2

)

print

(

f

"

Test accuracy:

**{**

test\_acc

**}**

"

)

Epoch 1/25

360/360 [==============================] - 36s 90ms/step - loss: 2.0807 -

accuracy: 0.2929 - val\_loss: 1.6312 - val\_accuracy: 0.3714

Epoch 2/25

360/360 [==============================] - 24s 67ms/step - loss: 1.5651 -

accuracy: 0.3966 - val\_loss: 1.5675 - val\_accuracy: 0.3970

Epoch 3/25

360/360 [==============================] - 30s 84ms/step - loss: 1.4607 -

accuracy: 0.4390 - val\_loss: 1.4654 - val\_accuracy: 0.4366

Epoch 4/25

360/360 [==============================] - 26s 72ms/step - loss: 1.3787 -

accuracy: 0.4732 - val\_loss: 1.4324 - val\_accuracy: 0.4528

Epoch 5/25

360/360 [==============================] - 26s 71ms/step - loss: 1.3074 -

accuracy: 0.5052 - val\_loss: 1.3743 - val\_accuracy: 0.4741

Epoch 6/25

360/360 [==============================] - 25s 70ms/step - loss: 1.2326 -

accuracy: 0.5342 - val\_loss: 1.3813 - val\_accuracy: 0.4814

Epoch 7/25

360/360 [==============================] - 24s 68ms/step - loss: 1.1439 -

accuracy: 0.5684 - val\_loss: 1.3892 - val\_accuracy: 0.4908

Epoch 8/25

360/360 [==============================] - 23s 63ms/step - loss: 1.0651 -

accuracy: 0.6001 - val\_loss: 1.4366 - val\_accuracy: 0.4839

Epoch 9/25

360/360 [==============================] - 23s 64ms/step - loss: 0.9864 -

accuracy: 0.6319 - val\_loss: 1.4948 - val\_accuracy: 0.4747

Epoch 10/25

360/360 [==============================] - 23s 65ms/step - loss: 0.9015 -

accuracy: 0.6638 - val\_loss: 1.5591 - val\_accuracy: 0.4828

Epoch 11/25

360/360 [==============================] - 30s 83ms/step - loss: 0.8041 accuracy: 0.7026 - val\_loss: 1.6586 - val\_accuracy: 0.4946

Epoch 12/25

360/360 [==============================] - 29s 80ms/step - loss: 0.7274 accuracy: 0.7344 - val\_loss: 1.8447 - val\_accuracy: 0.4769

Epoch 13/25

360/360 [==============================] - 25s 69ms/step - loss: 0.6595 accuracy: 0.7585 - val\_loss: 1.9819 - val\_accuracy: 0.4865

Epoch 14/25

360/360 [==============================] - 26s 73ms/step - loss: 0.5831 accuracy: 0.7890 - val\_loss: 2.1120 - val\_accuracy: 0.4847

Epoch 15/25

360/360 [==============================] - 22s 62ms/step - loss: 0.5244 accuracy: 0.8075 - val\_loss: 2.2483 - val\_accuracy: 0.4842 7/7 - 9s - loss: 1.3664 - accuracy: 0.4762 - 9s/epoch - 1s/step

Test accuracy: 0.4762241840362549

We then plot the training accuracy (accuracy) and validation accuracy (val\_accuracy) on the y-axis, and the epoch number on the x-axis to visualize the progress of the training.

[9]:

plt

.

plot(history

.

history[

'

accuracy

'

])

plt

.

plot(history

.

history[

'

val\_accuracy

'

])

plt

.

title(

'

model accuracy

'

)

plt

.

ylabel(

'

accuracy

'

)

plt

.

xlabel(

'

epoch

'

)

plt

.

legend([

'

train

'

,

'

validate

'

]

, loc

=

'

upper left

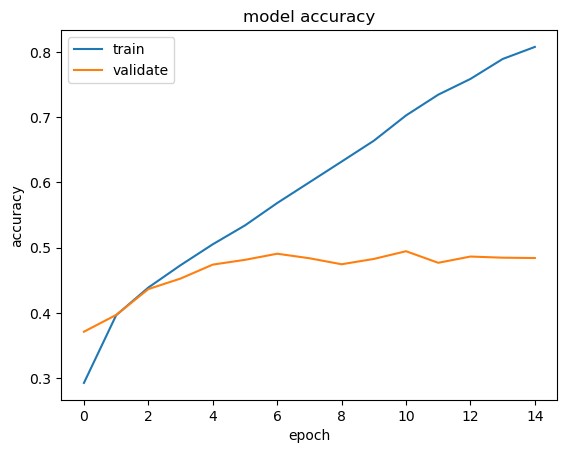
'

)

plt

.

show()



Both the training and validation accuracy increase for the first few epochs, but then the validation accuracy trails off. This observation suggests that the model is overfitting the training data. Overfitting occurs when the model becomes too complex and starts to memorize the training data instead of learning to generalize well to new, unseen data. As a result, the model may perform well on the training data but poorly on the validation and test data. To address this issue, we apply two techniques: 1. Data augmentation - a technique used to artificially expand a dataset by creating modified versions of existing images. The modified images retain the same label as the original, but have been modified in some way, such as by rotation, scaling, or flipping. Data augmentation can be used to improve the performance of machine learning models by exposing them to a greater variety of training data, which can help to prevent overfitting and improve generalization. 2. Dropout - a regularization technique commonly used in neural networks to prevent overfitting. It works by randomly setting some of the output values of a layer to zero during training, which causes the network to learn more robust features that are not overly dependent on the input values. By randomly dropping out some neurons during training, the network is forced to learn multiple independent representations of the same data, which helps to reduce the network’s sensitivity to specific inputs and increases its generalization ability.

Here, we use ImageDataGenerator to create a generator for the images. The rescale parameter is used to scale the pixel values between 0 and 1. The flow\_from\_directory method is used to generate batches of data from the directory data. The target\_size parameter is used to resize the images to 224x224 pixels. The batch\_size parameter specifies the size of the batches. The class\_mode parameter is set to categorical to indicate that the labels are categorical. When using the ImageDataGenerator class, each image is randomly transformed before being fed to the model during training, which creates new examples to train on. The flow\_from\_directory method is used to generate batches of images from a directory and automatically infer the labels from the directory structure.

[10]:

**from**

**tensorflow**

**.**

**keras**

**.**

**preprocessing**

**.**

**image**

**import**

ImageDataGenerator

datagen

=

ImageDataGenerator(

rescale

=

1.0

/

255

,

rotation\_range

=

30

,

width\_shift\_range

=

0.1

,

height\_shift\_range

=

0.1

,

shear\_range

=

0.2

,

zoom\_range

=

0.2

,

horizontal\_flip

=

**True**

,

fill\_mode

=

"

nearest

"

,

validation\_split

=

0.2

)

train\_generator

=

datagen

.

flow\_from\_directory(

f

"

**{**

img\_dir

**}**

/train

"

,

target\_size

=

(

48

,

48

)

,

batch\_size

=

64

,

class\_mode

=

"

categorical

"

,

subset

=

"

training

"

,

)

validation\_generator

=

datagen

.

flow\_from\_directory(

f

"

**{**

img\_dir

**}**

/validation

"

,

target\_size

=

(

48

,

48

)

,

batch\_size

=

64

,

class\_mode

=

"

categorical

"

,

subset

=

"

validation

"

,

)

test\_generator

=

ImageDataGenerator(rescale

=

1.0

/

255

)

.

flow\_from\_directory(

f

"

**{**

img\_dir

**}**

/validation

"

,

target\_size

=

(

48

,

48

)

,

batch\_size

=

1024

,

class\_mode

=

"

categorical

"

,

)

Found 23060 images belonging to 7 classes.

Found 1411 images belonging to 7 classes.

Found 7066 images belonging to 7 classes.

The model is redefined to include Dropout layers

[15]:

model

=

models

.

Sequential()

model

.

add(layers

.

Conv2D(

32

,(

3

,

3

, activation

)

=

'

relu

'

, input\_shape

=

(

48

,

48

,

3

)))

model

.

add(layers

.

Dropout(

0.25

))

model

.

add(layers

.

MaxPooling2D((

2

,

2

)))

model

.

add(layers

.

Conv2D(

64

,(

3

,

3

)

, activation

=

'

relu

'

))

model

.

add(layers

.

Dropout(

0.25

))

model

.

add(layers

.

MaxPooling2D((

2

,

2

)))

model

.

add(layers

.

Conv2D(

64

,(

3

,

3

)

, activation

=

'

relu

'

))

model

.

add(layers

.

Dropout(

0.5

))

model

.

add(layers

.

Flatten())

model

.

add(layers

.

Dense(

64

, activation

=

'

relu

'

))

model

.

add(layers

.

Dense(

7

))

[16]:

*# Compile the model*

model

.

compile(optimizer

=

'

adam

'

,

loss

=

tf

.

keras

.

losses

.

CategoricalCrossentropy(from\_logits

=

**True**

)

,

metrics

=

[

'

accuracy

'

])

history

=

model

.

fit(

train\_generator,

epochs

=

25

,

validation\_data

=

validation\_generator,

callbacks

=

[

early\_stop],

)

test\_loss, test\_acc

=

model

.

evaluate(test\_generator, verbose

=

2

)

print

(

f

"

Test accuracy:

**{**

test\_acc

**}**

"

)

Epoch 1/25

2023-03-27 17:14:27.005565: E tensorflow/core/grappler/optimizers/meta\_optimizer.cc:954] layout failed:

INVALID\_ARGUMENT: Size of values 0 does not match size of permutation 4 @ fanin shape insequential\_3/dropout\_6/dropout/SelectV2-2-TransposeNHWCToNCHW-

LayoutOptimizer

361/361 [==============================] - 46s 122ms/step - loss: 1.7970 -

accuracy: 0.2546 - val\_loss: 1.7613 - val\_accuracy: 0.2934

Epoch 2/25

361/361 [==============================] - 44s 122ms/step - loss: 1.7432 -

accuracy: 0.2883 - val\_loss: 1.7328 - val\_accuracy: 0.3040

Epoch 3/25

361/361 [==============================] - 44s 121ms/step - loss: 1.7034 -

accuracy: 0.3101 - val\_loss: 1.7013 - val\_accuracy: 0.3189

Epoch 4/25

361/361 [==============================] - 43s 120ms/step - loss: 1.6537 -

accuracy: 0.3411 - val\_loss: 1.6368 - val\_accuracy: 0.3629

Epoch 5/25

361/361 [==============================] - 44s 121ms/step - loss: 1.6122 -

accuracy: 0.3636 - val\_loss: 1.6053 - val\_accuracy: 0.3749

Epoch 6/25

361/361 [==============================] - 43s 120ms/step - loss: 1.5756 accuracy: 0.3809 - val\_loss: 1.5470 - val\_accuracy: 0.4132

Epoch 7/25

361/361 [==============================] - 43s 119ms/step - loss: 1.5477 -

accuracy: 0.3947 - val\_loss: 1.5130 - val\_accuracy: 0.4174

Epoch 8/25

361/361 [==============================] - 43s 120ms/step - loss: 1.5208 -

accuracy: 0.4087 - val\_loss: 1.4918 - val\_accuracy: 0.4174

Epoch 9/25

361/361 [==============================] - 44s 121ms/step - loss: 1.4949 -

accuracy: 0.4219 - val\_loss: 1.4606 - val\_accuracy: 0.4465

Epoch 10/25

361/361 [==============================] - 44s 121ms/step - loss: 1.4819 -

accuracy: 0.4271 - val\_loss: 1.4789 - val\_accuracy: 0.4309

Epoch 11/25

361/361 [==============================] - 44s 121ms/step - loss: 1.4622 -

accuracy: 0.4316 - val\_loss: 1.4409 - val\_accuracy: 0.4543

Epoch 12/25

361/361 [==============================] - 43s 119ms/step - loss: 1.4559 accuracy: 0.4373 - val\_loss: 1.4555 - val\_accuracy: 0.4465

Epoch 13/25

361/361 [==============================] - 43s 119ms/step - loss: 1.4433 accuracy: 0.4436 - val\_loss: 1.4277 - val\_accuracy: 0.4571

Epoch 14/25

361/361 [==============================] - 43s 120ms/step - loss: 1.4231 accuracy: 0.4542 - val\_loss: 1.4318 - val\_accuracy: 0.4635

Epoch 15/25

361/361 [==============================] - 43s 120ms/step - loss: 1.4226 accuracy: 0.4554 - val\_loss: 1.4090 - val\_accuracy: 0.4500

Epoch 16/25

361/361 [==============================] - 44s 121ms/step - loss: 1.4148 -

accuracy: 0.4548 - val\_loss: 1.4014 - val\_accuracy: 0.4678 Epoch 17/25

361/361 [==============================] - 45s 125ms/step - loss: 1.4001 -

accuracy: 0.4608 - val\_loss: 1.3916 - val\_accuracy: 0.4564

Epoch 18/25

361/361 [==============================] - 44s 122ms/step - loss: 1.3956 -

accuracy: 0.4662 - val\_loss: 1.3823 - val\_accuracy: 0.4706

Epoch 19/25

361/361 [==============================] - 46s 126ms/step - loss: 1.3916 -

accuracy: 0.4630 - val\_loss: 1.3738 - val\_accuracy: 0.4670

Epoch 20/25

361/361 [==============================] - 44s 123ms/step - loss: 1.3795 -

accuracy: 0.4675 - val\_loss: 1.3916 - val\_accuracy: 0.4663

Epoch 21/25

361/361 [==============================] - 44s 122ms/step - loss: 1.3755 -

accuracy: 0.4690 - val\_loss: 1.3586 - val\_accuracy: 0.4819

Epoch 22/25

361/361 [==============================] - 44s 122ms/step - loss: 1.3683 accuracy: 0.4726 - val\_loss: 1.3422 - val\_accuracy: 0.4819

Epoch 23/25

361/361 [==============================] - 44s 122ms/step - loss: 1.3633 -

accuracy: 0.4772 - val\_loss: 1.3745 - val\_accuracy: 0.4720

Epoch 24/25

361/361 [==============================] - 43s 120ms/step - loss: 1.3645 -

accuracy: 0.4808 - val\_loss: 1.3247 - val\_accuracy: 0.5004

Epoch 25/25

361/361 [==============================] - 43s 120ms/step - loss: 1.3593 accuracy: 0.4758 - val\_loss: 1.3616 - val\_accuracy: 0.4741 7/7 - 10s - loss: 1.2246 - accuracy: 0.5467 - 10s/epoch - 1s/step

Test accuracy: 0.54670250415802

[17]:

plt

.

plot(history

.

history[

'

accuracy

'

])

plt

.

plot(history

.

history[

'

val\_accuracy

'

])

plt

.

title(

'

model accuracy

'

)

plt

.

ylabel(

'

accuracy

'

)

plt

.

xlabel(

'

epoch

'

)

plt

.

legend([

'

train

'

,

'

validate

'

]

, loc

=

'

upper left

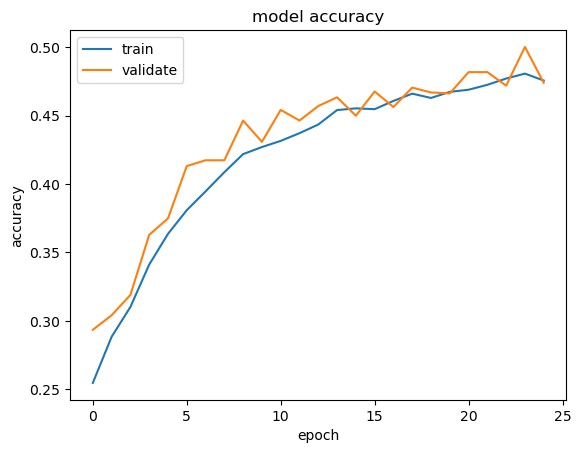
'

)

plt

.

show()



After the model improvements, the train accuracy does not improve significantly while the validation accuracy remains close to the train accuracy. This is an indication of good generalization of the model. This means that the model is able to learn from the training data and generalize well to unseen data. Higher accuracy may be achieved by training for more epochs.

Another approach at improving the model is to use a pretrained model. We use the VGG16 pretrained model. VGG-16 is a popular convolutional neural network architecture that has been pretrained on large-scale image classification tasks such as the ImageNet dataset. Using a pretrained VGG-16 model can provide several benefits, including: 1. Transfer learning: Pretrained models can be fine-tuned on a new dataset with fewer training examples, which can lead to better performance than training a new model from scratch. 2. Feature extraction: The convolutional layers of a pretrained VGG-16 model can be used to extract features from images, which can then be used as inputs to a new model for a specific task.

First we load the pre-trained VGG16 model with the imagenet weights. We do not include the fully connected layers at the output.

Next we freeze weights for the convolutional base

Next, we create a model with the pretrained VGG16 model convolutional base and additional fully connected layers

[18]:

**from**

**tensorflow**

**.**

**keras**

**.**

**applications**

**import**

VGG16

*# Load the pre-trained VGG16 model*

conv\_base

=

VGG16(weights

=

'

imagenet

'

,

include\_top

=

**False**

,

input\_shape

=

(

48

,

48

,

3

))

*# Freeze the weights of the convolutional base*

conv\_base

.

trainable

=

**False**

*# Create a new model that includes the VGG16 convolutional base*

model

=

models

.

Sequential()

model

.

add(conv\_base)

model

.

add(layers

.

Flatten())

model

.

add(layers

.

Dense(

4096

, activation

=

'

relu

'

))

model

.

add(layers

.

Dense(

1024

, activation

=

'

relu

'

))

model

.

add(layers

.

Dropout(

0.2

))

model

.

add(layers

.

Dense(

7

, activation

=

'

softmax

'

))

*# Compile the model*

model

.

compile(optimizer

=

'

adam

'

,

loss

=

'

categorical\_crossentropy

'

,

metrics

=

[

'

accuracy

'

])

*# Print the model summary*

model

.

summary()

Model: "sequential\_4"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

|  |  |  |
| --- | --- | --- |
| vgg16 (Functional) | (None, 1, 1, 512) | 14714688 |
| flatten\_4 (Flatten) | (None, 512) | 0 |
| dense\_8 (Dense) | (None, 4096) | 2101248 |
| dense\_9 (Dense) | (None, 1024) | 4195328 |
| dropout\_9 (Dropout) | (None, 1024) | 0 |
| dense\_10 (Dense) | (None, 7) | 7175 |

=================================================================

Total params: 21,018,439

Trainable params: 6,303,751

Non-trainable params: 14,714,688

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[19]:

history

=

model

.

fit(

train\_generator,

epochs

=

25

,

validation\_data

=

validation\_generator,

callbacks

=

[

early\_stop],

)

test\_loss, test\_acc

=

model

.

evaluate(test\_generator, verbose

=

2

)

print

(

f

"

Test accuracy:

**{**

test\_acc

**}**

"

)

Epoch 1/25

361/361 [==============================] - 47s 124ms/step - loss: 1.7104 accuracy: 0.3161 - val\_loss: 1.6659 - val\_accuracy: 0.3359

Epoch 2/25

361/361 [==============================] - 44s 122ms/step - loss: 1.6472 -

accuracy: 0.3514 - val\_loss: 1.6344 - val\_accuracy: 0.3664

Epoch 3/25

361/361 [==============================] - 45s 126ms/step - loss: 1.6290 -

accuracy: 0.3523 - val\_loss: 1.6539 - val\_accuracy: 0.3466

Epoch 4/25

361/361 [==============================] - 45s 125ms/step - loss: 1.6152 -

accuracy: 0.3645 - val\_loss: 1.6188 - val\_accuracy: 0.3692

Epoch 5/25

361/361 [==============================] - 45s 124ms/step - loss: 1.6041 -

accuracy: 0.3692 - val\_loss: 1.6165 - val\_accuracy: 0.3728

Epoch 6/25

361/361 [==============================] - 45s 125ms/step - loss: 1.5898 -

accuracy: 0.3762 - val\_loss: 1.6279 - val\_accuracy: 0.3707

Epoch 7/25

361/361 [==============================] - 45s 126ms/step - loss: 1.5850 accuracy: 0.3775 - val\_loss: 1.6111 - val\_accuracy: 0.3777

Epoch 8/25

361/361 [==============================] - 46s 126ms/step - loss: 1.5781 accuracy: 0.3791 - val\_loss: 1.5900 - val\_accuracy: 0.3785

Epoch 9/25

361/361 [==============================] - 45s 126ms/step - loss: 1.5700 accuracy: 0.3863 - val\_loss: 1.5888 - val\_accuracy: 0.3756

Epoch 10/25

361/361 [==============================] - 46s 128ms/step - loss: 1.5657 accuracy: 0.3818 - val\_loss: 1.5888 - val\_accuracy: 0.3770

Epoch 11/25

361/361 [==============================] - 46s 128ms/step - loss: 1.5475 -

accuracy: 0.3964 - val\_loss: 1.5850 - val\_accuracy: 0.3763 Epoch 12/25

361/361 [==============================] - 46s 126ms/step - loss: 1.5413 -

accuracy: 0.3973 - val\_loss: 1.5620 - val\_accuracy: 0.3841

Epoch 13/25

361/361 [==============================] - 46s 126ms/step - loss: 1.5290 -

accuracy: 0.4016 - val\_loss: 1.5633 - val\_accuracy: 0.3926

Epoch 16/25

361/361 [==============================] - 45s 125ms/step - loss: 1.5202 -

accuracy: 0.4072 - val\_loss: 1.5650 - val\_accuracy: 0.3877

Epoch 17/25

361/361 [==============================] - 45s 124ms/step - loss: 1.5174 -

accuracy: 0.4073 - val\_loss: 1.5871 - val\_accuracy: 0.3891

Epoch 18/25

361/361 [==============================] - 44s 122ms/step - loss: 1.5077 -

accuracy: 0.4127 - val\_loss: 1.5766 - val\_accuracy: 0.3855

Epoch 19/25

361/361 [==============================] - 44s 122ms/step - loss: 1.5082 accuracy: 0.4112 - val\_loss: 1.5776 - val\_accuracy: 0.3990

Epoch 20/25

361/361 [==============================] - 44s 123ms/step - loss: 1.5025 -

accuracy: 0.4105 - val\_loss: 1.5866 - val\_accuracy: 0.3785

Epoch 21/25

361/361 [==============================] - 47s 131ms/step - loss: 1.4897 -

accuracy: 0.4188 - val\_loss: 1.5638 - val\_accuracy: 0.4018

Epoch 22/25

361/361 [==============================] - 45s 123ms/step - loss: 1.4916 accuracy: 0.4200 - val\_loss: 1.5827 - val\_accuracy: 0.3990 7/7 - 16s - loss: 1.5402 - accuracy: 0.4137 - 16s/epoch - 2s/step

Test accuracy: 0.4136711061000824

[20]:

plt

.

plot(history

.

history[

'

accuracy

'

])

plt

.

plot(history

.

history[

'

val\_accuracy

'

])

plt

.

title(

'

model accuracy

'

)

plt

.

ylabel(

'

accuracy

'

)

plt

.

xlabel(

'

epoch

'

)

plt

.

legend([

'

train

'

,

'

validate

'

]

, loc

=

'

upper left

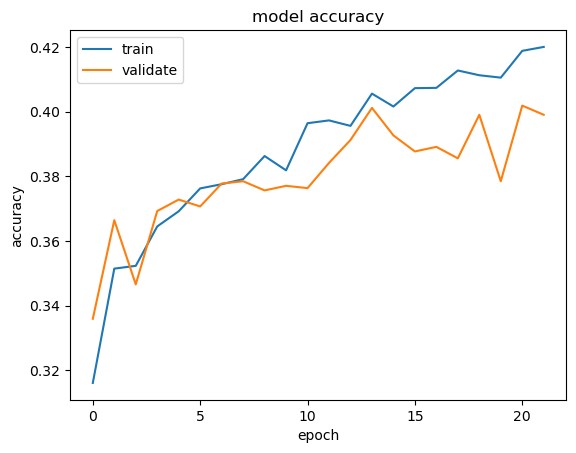
'

)

plt

.

show()



The training and validation accuracies go hand in hand for this model as well.

The accuracy on the training set is 41.37%.

Increasing the number of epochs and also increasing the amount of training data would result in a higher accuracy.

[ ]

**References**

Kortli, Y., Jridi, M., Al Falou, A., & Atri, M. (2020). Face Recognition Systems: A Survey. *Sensors*, *20*(2), 342. https://doi.org/10.3390/s20020342

Salama AbdELminaam, D., Almansori, A. M., Taha, M., & Badr, E. (2020). A deep facial recognition system using computational intelligent algorithms. *PLOS ONE*, *15*(12), e0242269. https://doi.org/10.1371/journal.pone.0242269