پروژه ترم داده کاوی

دانشجو: کیمیا اسماعیلی

610398193

استاد: دکتر هدیه ساجدی

لینک google colab:

https://colab.research.google.com/drive/1qK_heDBRuHMRhlOFGpzKNFPu8__eO42S?usp=sharing

فهرست مطالب:

تعریف پروژه:

این پروژه بر اساس مطالعه اخیر انجام شده توسط مورات کوکلو و همکاران است. آنها گونه های برگ انگور را به شرح زیر تعیین کردند: طبقه بندی (classification) مبتنی بر یادگیری عمیق با استفاده از تصاویر برگ انگور. برای این منظور تصاویری از 500 برگ درخت انگور متعلق به 5 گونه با سیستم خودنور مخصوص تهیه شد.

لینک مقاله مطالعه مذبور:

https://www.sciencedirect.com/science/article/abs/pii/S0263224121013142?via%3Dihub

بر اساس چکیده مطالعه آنها: محصول اصلی انگور به صورت تازه یا فرآوری شده مصرف می شود. علاوه بر این، برگ انگور سالی یک بار به عنوان محصول جانبی برداشت می شود. دانستن گونه های برگ انگور از نظر قیمت و طعم مهم است.

در اینجا می‌خواهیم از برخی مدل‌های تنظیم دقیق (fine-tuning) و یادگیری عمیق استفاده کنیم که از قبل برای طبقه‌بندی تصاویر برگ‌های انگور trained شده‌اند.

ایمپورت کردن لایبرری‌ها و توابع لازم:

```
import tensorflow as tf
from tensorflow.keras import layers
from keras.layers import Dense
from keras.models import Sequential

from keras.models import Model
from keras.layers import Input, Conv2D, MaxPool2D, UpSampling2D

from tensorflow.keras import losses
from tensorflow.keras import preprocessing
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator,load_img
from tensorflow.keras.layers.experimental.preprocessing import RandomFlip, RandomRotation, RandomZoom, RandomTranslation
from tensorflow.keras.preprocessing import image_dataset_from_directory
from sklearn.model_selection import train_test_split

from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
from PIL import Image
import tensorflow
```

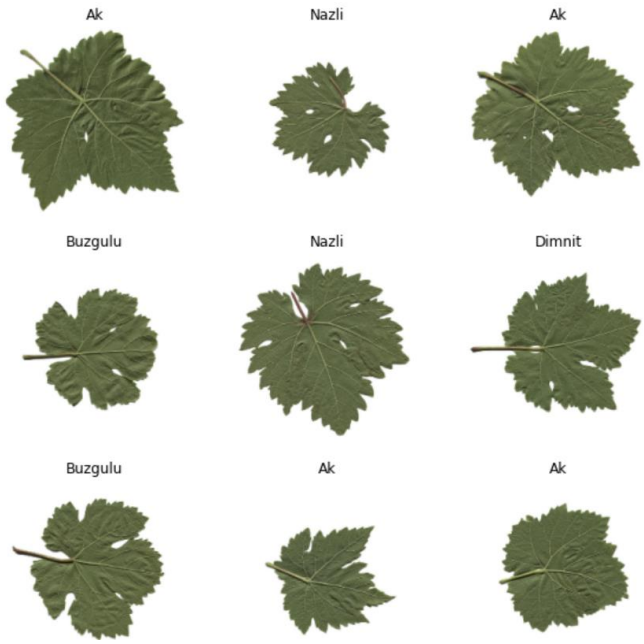
process کردن:

```
BATCH_SIZE = 16
IMG_SIZE = (224, 224)

directory = "dataset"
train_dataset = image_dataset_from_directory(directory, shuffle=True, batch_size=BATCH_SIZE, image_size=IMG_SIZE, validation_split=0.2,
                                             subset='training',
                                             seed=42)

test_dataset = image_dataset_from_directory(directory, shuffle=True, batch_size=BATCH_SIZE, image_size=IMG_SIZE, validation_split=0.2,
                                            subset='validation',
                                            seed=42)
```

```
Found 500 files belonging to 5 classes.
Using 400 files for training.
Found 500 files belonging to 5 classes.
Using 100 files for validation.
```

ما تصاویر را از دایرکتوری tensorflow.keras.preprocessing.image_dataset_from_directory() گرفتیم.
ما 500 تصویر با اندازه batch، 16 و اندازه تصاویر (3 * 224 * 224) داریم. اندازه دیتاست training، 320 و دیتاست validation، 80 و دیتاست test، 100 است.

```python
class_names = train_dataset.class_names

plt.figure(figsize=(10, 10))
for images, labels in train_dataset.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```



سپس برخی از تصاویر برگ ها را با برچسب های مربوط به آنها دیدیم.

```
X_train = np.empty((0,224, 224, 3))
y_train = np.array([])
for x, y in train_dataset:
    X_train = np.concatenate([X_train, x.numpy()/255.])
    y_train = np.concatenate([y_train, y.numpy()])
```

```
X_train_target, X_val, y_train_target, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
print(X_train_target.shape, X_val.shape)
```

```
(320, 224, 224, 3) (80, 224, 224, 3)
```

```
X_test = np.empty((0,224, 224, 3))
y_test = np.array([])
for x, y in test_dataset:
    X_test = np.concatenate([X_test, x.numpy()/255.])
    y_test = np.concatenate([y_test, y.numpy()])
```

سپس به صورت بالا، دیتاست های ولیدیشن و تست و train را ساختیم.

```
def data_augmenter():
    data_augmentation = tf.keras.Sequential()
    data_augmentation.add(RandomFlip('horizontal'))
    data_augmentation.add(RandomRotation(0.3))
    data_augmentation.add(RandomZoom(0.5,0.2))
    return data_augmentation
```

ما یک تابع تقویت کننده داده را تعریف کردیم که در مجموعه داده train/ اعتبارسنجی اعمال خواهد شد. تابع مذکور زوم، flip و چرخش تصادفی را انجام می دهد.

```
data_augmentation = data_augmenter()

for image, _ in train_dataset.take(1):
    plt.figure(figsize=(10, 10))
    first_image = image[10]
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        augmented_image = data_augmentation(tf.expand_dims(first_image, 0))
        plt.imshow(augmented_image[0] / 255)
        plt.axis('off')
```

تصاویر تقویت شده (augmented) را میبینیم.

اتواینکدر:

Autoencoder یک شبکه عصبی مصنوعی بدون نظارت است که برای کپی کردن ورودی خود در خروجی آموزش دیده است. در مورد داده های تصویر، رمزگذار خودکار ابتدا تصویر را به یک نمایش با ابعاد پایین تر رمزگذاری می کند، سپس آن نمایش را به تصویر رمزگشایی می کند. در اینجا ما میخواهیم یک مدل برای رمزگذار خودکار ایجاد می کنیم.

```python
inputs = layers.Input(shape=(224, 224, 3))

#Encoder
x = layers.Conv2D(32, (3, 3), activation="relu", padding="same")(inputs)
x = layers.MaxPooling2D((2, 2), padding="same")(x)
x = layers.Conv2D(32, (3, 3), activation="relu", padding="same")(x)
x = layers.MaxPooling2D((2, 2), padding="same")(x)

#Decoder
x = layers.Conv2DTranspose(32, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2DTranspose(32, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(3, (3, 3), activation="sigmoid", padding="same")(x)

#Autoencoder
autoencoder = Model( inputs, x)
autoencoder.compile( optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3), loss="binary_crossentropy")
autoencoder.summary()
```

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_6 (InputLayer)        [(None, 224, 224, 3)]     0

 conv2d_13 (Conv2D)          (None, 224, 224, 32)      896

 max_pooling2d_6 (MaxPooling  (None, 112, 112, 32)     0
 2D)

 conv2d_14 (Conv2D)          (None, 112, 112, 32)      9248

 max_pooling2d_7 (MaxPooling  (None, 56, 56, 32)       0
 2D)

 conv2d_transpose_6 (Conv2DT  (None, 112, 112, 32)     9248
 ranspose)

 conv2d_transpose_7 (Conv2DT  (None, 224, 224, 32)     9248
 ranspose)

 conv2d_15 (Conv2D)          (None, 224, 224, 3)       867

=================================================================
Total params: 29,507
Trainable params: 29,507
Non-trainable params: 0
_________________________________________________________________
```

Encoder-Decoder به طور خودکار از دو ساختار زیر تشکیل شده است:

1: رمزگذار: این شبکه داده ها را به ابعاد پایین تر تقلیل میدهد و downsample میکند.

2: رمزگشا: این شبکه داده های اصلی را از نمایش ابعاد پایین تر بازسازی می کند.

نمایش ابعاد پایین تر (خروجی شبکه رمزگذار) معمولاً به عنوان نمایش فضای نهفته شناخته می شود. لایه ورودی برای این مدل شکل (3*224*224) دارد، چند لایه کانولوشن با 'activation='relu وpadding='same ' برای قسمت رمزگذار اضافه می کنیم. پس از آن، چند لایه کانولوشنال transpose (Conv2DTranspose) با 'activation='relu و 'padding='same برای قسمت رمزگشا اضافه می کنیم. این لایه‌ها داده‌ها را نمونه‌گیری می‌کنند، بنابراین در پایان با یک خروجی (3*224*224) مواجه می‌شویم که اندازه‌ای مشابه لایه اول دارد. بر اساس خلاصه مدل، تعداد پارامترهای آموزش 449795 است.

```
autoencoder.fit(X_train,X_train, epochs=20,shuffle = True, batch_size = 16)
```

```
Epoch 1/20
25/25 [==============================] - 10s 381ms/step - loss: 0.5053
Epoch 2/20
25/25 [==============================] - 9s 376ms/step - loss: 0.3685
Epoch 3/20
25/25 [==============================] - 10s 382ms/step - loss: 0.2844
Epoch 4/20
25/25 [==============================] - 9s 368ms/step - loss: 0.2382
Epoch 5/20
25/25 [==============================] - 9s 365ms/step - loss: 0.2310
Epoch 6/20
25/25 [==============================] - 9s 366ms/step - loss: 0.2290
Epoch 7/20
25/25 [==============================] - 9s 366ms/step - loss: 0.2277
Epoch 8/20
25/25 [==============================] - 9s 369ms/step - loss: 0.2266
Epoch 9/20
25/25 [==============================] - 9s 370ms/step - loss: 0.2256
Epoch 10/20
25/25 [==============================] - 9s 370ms/step - loss: 0.2247
Epoch 11/20
25/25 [==============================] - 9s 368ms/step - loss: 0.2238
Epoch 12/20
25/25 [==============================] - 9s 368ms/step - loss: 0.2230
Epoch 13/20
25/25 [==============================] - 9s 370ms/step - loss: 0.2222
Epoch 14/20
25/25 [==============================] - 9s 369ms/step - loss: 0.2214
Epoch 15/20
25/25 [==============================] - 9s 366ms/step - loss: 0.2206
Epoch 16/20
25/25 [==============================] - 9s 368ms/step - loss: 0.2180
Epoch 17/20
25/25 [==============================] - 9s 372ms/step - loss: 0.2120
Epoch 18/20
25/25 [==============================] - 983s 41s/step - loss: 0.2110
Epoch 19/20
25/25 [==============================] - 1986s 83s/step - loss: 0.2108
Epoch 20/20
25/25 [==============================] - 480s 20s/step - loss: 0.2106
<keras.callbacks.History at 0x15a9e8310>
```

همانطور که می بینید ما رمزگذار را با مجموعه تست/ اعتبارسنجی برای batch_size 16 و epoch 20 مطابقت دادیم. تابع ضرر بر روی binary_crossentropy تنظیم است و بهینه ساز برای این fitting، آدام با نرخ یادگیری 0.001 است.

```
predicts = autoencoder.predict(X_train)
```

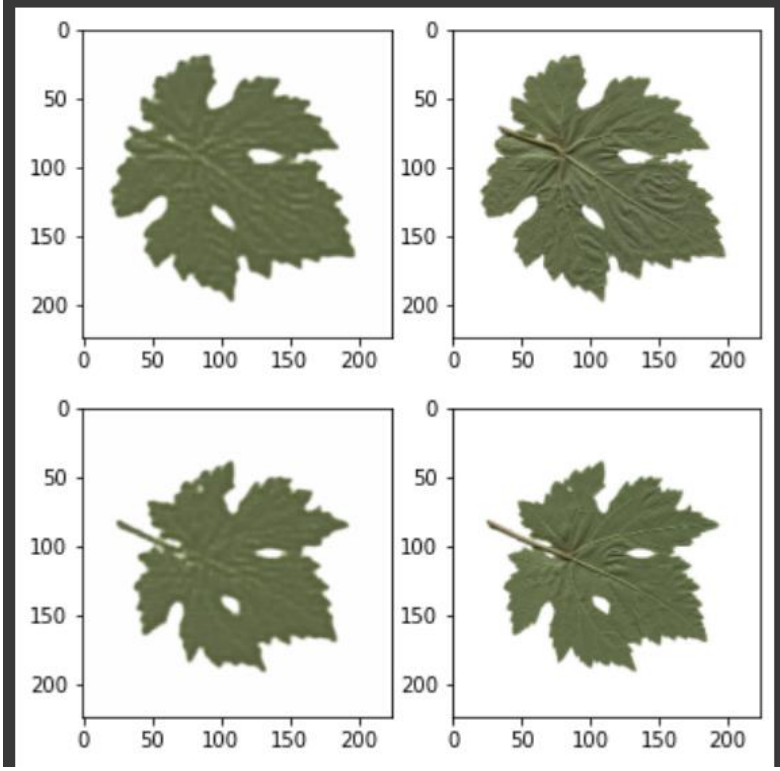
ما تصاویر جدید دیتاست آموزش را با استفاده از تابع autoencoder.predict() به صورت بالا ساختیم.

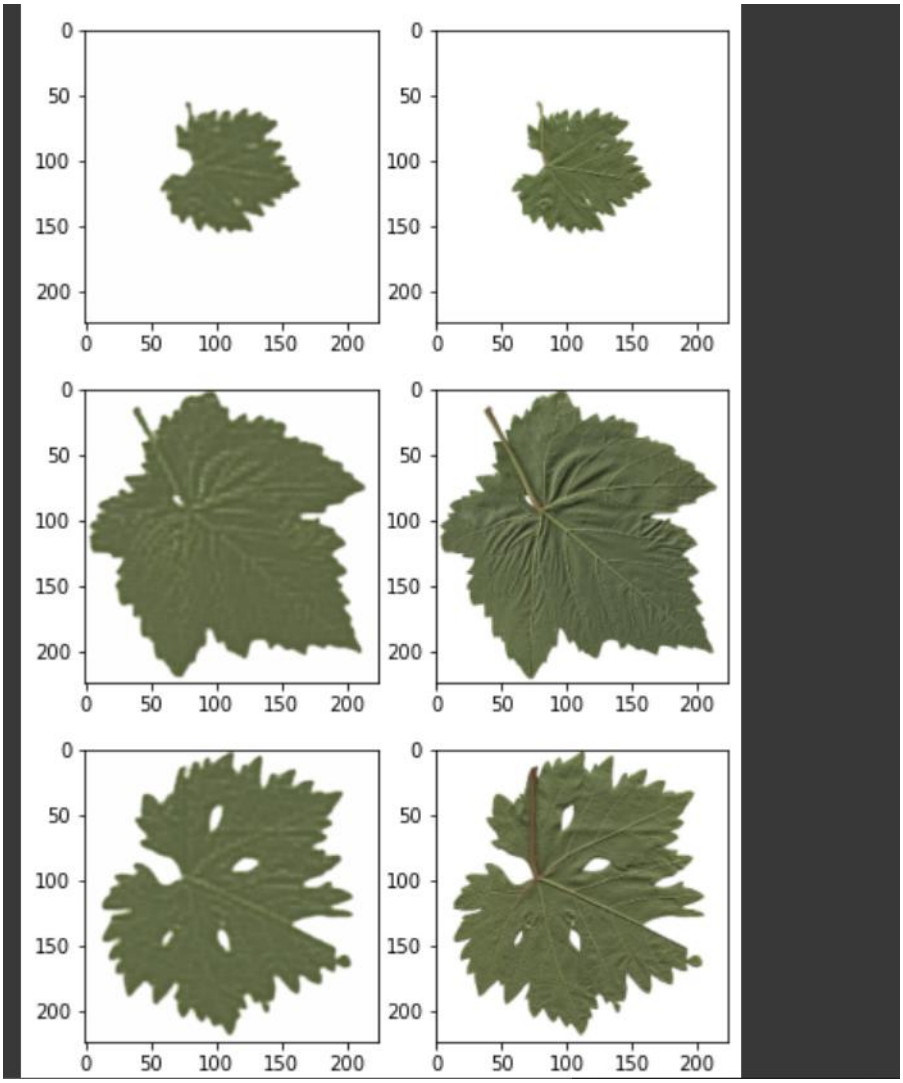
```
print('the shape of autoencoder prediction is: ', predicts.shape)
```

```
the shape of autoencoder prediction is:  (400, 224, 224, 3)
```

به این وسیله ما 400 تصویر با اندازه (224,224,3) با رنج (0,1) ساختیم.

```
for i in range(5):
    f, axarr = plt.subplots(1,2)
    axarr[0].imshow( predicts[i] )
    axarr[1].imshow( X_train[i] )
```





در اینجا تصاویر حذف نویز شده بدست آمده از اتواینکدر را با تصاویر اصلی مقایسه کردیم.

```python
def display(array1):
    n = 10
    indices = np.random.randint(len(array1), size=n)
    images1 = array1[indices, :]

    plt.figure(figsize=(10, 7))
    for i, image1 in enumerate(images1):
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(image1.reshape(224, 224,3))
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    plt.show()
```

```python
display(predicts)
```

همانطور که دیدیم، تصاویر اصلی بهتر از تصاویر بازسازی شده هستند. در ادامه دقت این تصاویر را میسنجیم.

مدل بر روی تصاویری که حذف نویز شده اند:

مدل یادگیری عمیق xception یک ماژول آغازین در شبکه‌های عصبی کانولوشن است که یک مرحله میانی بین کانولوشن منظم و عملیات کانولوشن قابل تفکیک عمقی (پیچیدگی عمقی و به دنبال آن یک پیچش نقطه‌ای) است.

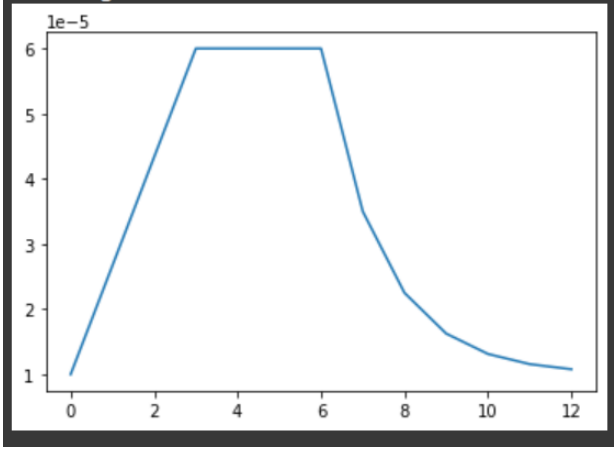
در این پروژه از مدل یادگیری عمیق xception استفاده می کنیم.

```
EPOCHS = 13
LR_START = 0.00001
LR_MAX = 0.0001 * 0.6
LR_MIN = 0.00001
LR_RAMPUP_EPOCHS = 3
LR_SUSTAIN_EPOCHS = 3
LR_EXP_DECAY = .5
```

ما تنظیمات زمانبندی نرخ متمایل را برای این مدل در بالا تنظیم کردیم.

```
def lrfn(epoch):
    if epoch < LR_RAMPUP_EPOCHS:
        lr = (LR_MAX - LR_START) / LR_RAMPUP_EPOCHS * epoch + LR_START
    elif epoch < LR_RAMPUP_EPOCHS + LR_SUSTAIN_EPOCHS:
        lr = LR_MAX
    else:
        lr = (LR_MAX - LR_MIN) * LR_EXP_DECAY**(epoch - LR_RAMPUP_EPOCHS - LR_SUSTAIN_EPOCHS) + LR_MIN
    return lr

lr_callback = tf.keras.callbacks.LearningRateScheduler(lrfn, verbose=True)

rng = [i for i in range(EPOCHS)]
y = [lrfn(x) for x in rng]
plt.plot(rng, y)
print("Learning rate schedule: {:.3g} to {:.3g} to {:.3g}".format(y[0], max(y), y[-1]))
```

```
Learning rate schedule: 1e-05 to 6e-05 to 1.08e-05
```



این LearningRateScheduler از Practical Machine Learning for Computer Vision گرفته شده است.

```
base_model = tf.keras.applications.xception.Xception( input_shape=(224,224,3), include_top=False, weights='imagenet')
base_model.trainable = True

input_shape = (224,224,3)

inputs = tf.keras.Input(shape=input_shape)
x = data_augmenter()(inputs)
x = tf.keras.applications.xception.preprocess_input(x)
x = base_model(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.5)(x)
x = layers.Flatten()(x)

outputs = layers.Dense(5,activation = 'softmax')(x)

model = tf.keras.Model(inputs, outputs)
```

```
model.summary()
```

```
Model: "model_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_10 (InputLayer)       [(None, 224, 224, 3)]     0

 sequential_5 (Sequential)   (None, 224, 224, 3)       0

 tf.math.truediv_2 (TFOpLamb  (None, 224, 224, 3)      0
 da)

 tf.math.subtract_2 (TFOpLam  (None, 224, 224, 3)      0
 bda)

 xception (Functional)       (None, 7, 7, 2048)        20861480

 global_average_pooling2d_2   (None, 2048)             0
 (GlobalAveragePooling2D)

 dropout (Dropout)           (None, 2048)              0

 flatten_2 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 5)                 10245

=================================================================
Total params: 20,871,725
Trainable params: 20,817,197
Non-trainable params: 54,528
_________________________________________________________________
```

در اینجا ما مدل اصلی خود را ساختیم:

ابتدا مدل xception از پیش آموزش دیده را از tf.keras.applications با شکل ورودی (224,224,3) دریافت کردیم. این مدل بر روی مجموعه داده imageNet که دارای کلاس 1000 است آموزش داده شده است.

ابتدا ما تمام 132 لایه xception را "قابل آموزش" تنظیم کردیم تا همه وزن ها به روز شوند. ما مدل xception پارامتر 23 متری خود را پس از افزایش داده ها در مدل، روی آن تصاویر تقویت شده تنظیم کردیم. سپس از لایه GlobalAveragePooling2D استفاده می کنیم و لایه را با پارامتر 0.5 رها می کنیم و واحدها (نرون ها) را مسطح می کنیم، در انتها این 2048 وزن را با 5 واحد به لایه softmax منتقل می کنیم.

خلاصه مدل ما در بالا نشان داده شده است.

```
denoise_train = autoencoder.predict(X_train)


y_train = y_train.astype('uint8')
y_test = y_test.astype('uint8')
y_val = y_val.astype('uint8')
```

برای قسمت فیتینگ، (x,y) را ساختیم.

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'],
)

history = model.fit(
    x=denoise_train*255.,
    y=y_train,
    validation_data=(X_val*255., y_val),
    callbacks = [lr_callback],
    epochs = EPOCHS,
    batch_size=16
)
```

```
Epoch 00001: LearningRateScheduler setting learning rate to 1e-05.
Epoch 1/13
25/25 [==============================] - 63s 2s/step - loss: 1.6404 - accuracy: 0.2050 - val_loss: 1.5978 - val_accuracy: 0.2125 - lr: 1.0000e-05

Epoch 00002: LearningRateScheduler setting learning rate to 2.6666666666666667e-05.
Epoch 2/13
25/25 [==============================] - 62s 2s/step - loss: 1.5719 - accuracy: 0.2775 - val_loss: 1.4888 - val_accuracy: 0.4125 - lr: 2.6667e-05

Epoch 00003: LearningRateScheduler setting learning rate to 4.3333333333333334e-05.
Epoch 3/13
25/25 [==============================] - 61s 2s/step - loss: 1.4155 - accuracy: 0.4575 - val_loss: 1.3606 - val_accuracy: 0.4500 - lr: 4.3333e-05

Epoch 00004: LearningRateScheduler setting learning rate to 6e-05.
Epoch 4/13
25/25 [==============================] - 64s 3s/step - loss: 1.2085 - accuracy: 0.5650 - val_loss: 1.1819 - val_accuracy: 0.6250 - lr: 6.0000e-05

Epoch 00005: LearningRateScheduler setting learning rate to 6e-05.
Epoch 5/13
25/25 [==============================] - 62s 2s/step - loss: 0.9820 - accuracy: 0.6650 - val_loss: 1.0139 - val_accuracy: 0.6750 - lr: 6.0000e-05

Epoch 00006: LearningRateScheduler setting learning rate to 6e-05.
Epoch 6/13
25/25 [==============================] - 63s 3s/step - loss: 0.7393 - accuracy: 0.7450 - val_loss: 0.8835 - val_accuracy: 0.7000 - lr: 6.0000e-05

Epoch 00007: LearningRateScheduler setting learning rate to 6e-05.
Epoch 7/13
25/25 [==============================] - 62s 3s/step - loss: 0.6157 - accuracy: 0.7925 - val_loss: 0.8546 - val_accuracy: 0.6875 - lr: 6.0000e-05
```

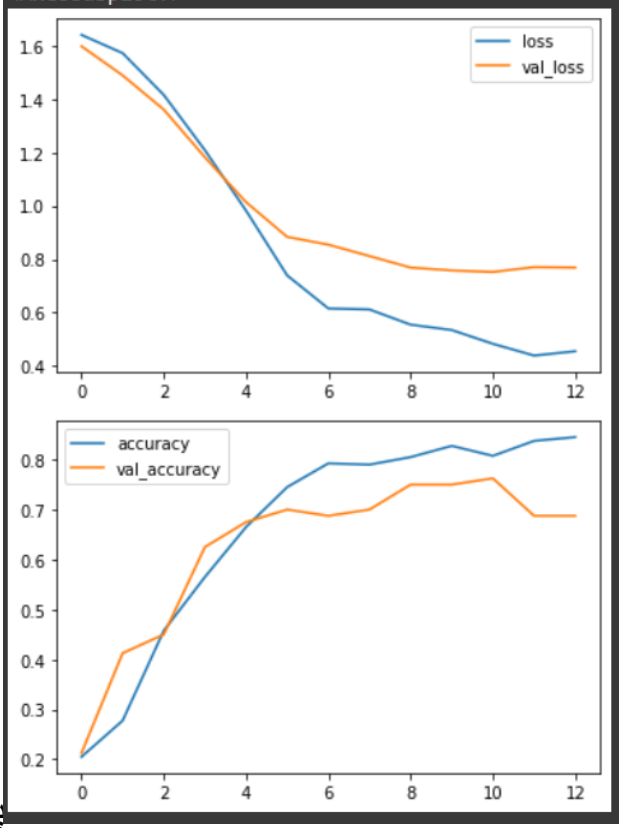
```
Epoch 00008: LearningRateScheduler setting learning rate to 3.5000000000000004e-05.
Epoch 8/13
25/25 [==============================] - 61s 2s/step - loss: 0.6119 - accuracy: 0.7900 - val_loss: 0.8118 - val_accuracy: 0.7000 - lr: 3.5000e-05

Epoch 00009: LearningRateScheduler setting learning rate to 2.25e-05.
Epoch 9/13
25/25 [==============================] - 61s 2s/step - loss: 0.5549 - accuracy: 0.8050 - val_loss: 0.7687 - val_accuracy: 0.7500 - lr: 2.2500e-05

Epoch 00010: LearningRateScheduler setting learning rate to 1.6250000000000002e-05.
Epoch 10/13
25/25 [==============================] - 62s 2s/step - loss: 0.5347 - accuracy: 0.8275 - val_loss: 0.7579 - val_accuracy: 0.7500 - lr: 1.6250e-05

Epoch 00011: LearningRateScheduler setting learning rate to 1.3125e-05.
Epoch 11/13
25/25 [==============================] - 62s 2s/step - loss: 0.4832 - accuracy: 0.8075 - val_loss: 0.7524 - val_accuracy: 0.7625 - lr: 1.3125e-05

Epoch 00012: LearningRateScheduler setting learning rate to 1.1562500000000002e-05.
Epoch 12/13
25/25 [==============================] - 61s 2s/step - loss: 0.4392 - accuracy: 0.8375 - val_loss: 0.7704 - val_accuracy: 0.6875 - lr: 1.1562e-05

Epoch 00013: LearningRateScheduler setting learning rate to 1.0781250000000001e-05.
Epoch 13/13
25/25 [==============================] - 62s 2s/step - loss: 0.4553 - accuracy: 0.8450 - val_loss: 0.7687 - val_accuracy: 0.6875 - lr: 1.0781e-05
```

سپس مدل را با بهینه ساز Adam، تابع تلفات sparse_categorical_crossentropy و دقت برای ماتریس ها compile کردیم.

پس از آن مدل را بر روی مجموعه داده training با استفاده از مجموعه داده اعتبارسنجی با اندازه دسته ای = 16 و دوره = 13 فیتینگ کردیم. نتایج نشان داده شده در بالا را به دست آوردیم.

```
history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot()
history_df.loc[:, ['accuracy', 'val_accuracy']].plot()
```

```
<AxesSubplot:>
```



پس ما 68% دقت بر روی دیتاست اعتبارسنجی و 84.5% دقت بر روی دیتاست آموزش داشتیم. ما تابع ضرر را بر اساس epoch که در بالا نشان داده شده است ترسیم کردیم.

```
base_model2 = model.layers[4]
base_model2.trainable = True

fine_tune_at = 100

for layer in base_model2.layers[:fine_tune_at]:
    layer.trainable = False
```

```
print("Number of layers in the model: ", len(base_model2.layers))
```

```
Number of layers in the model:  132
```

اینجا ما لایه ها را با استفاده از اجرای متدهای قابل آموزش بر روی لایه های مدل xception، عمل fine-tuning را انجام دادیم. در مدل xception، 132 لایه وجود دارد. حالت قابل آموزش 100 لایه اول را روی "False" و بقیه لایه ها را روی "True" تنظیم کردیم. با انجام این کار فقط وزن 32 لایه آخر به روز می شود. شما می توانید روش را در بالا مشاهده کنید.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
metrics= ['accuracy']
model.compile(loss="sparse_categorical_crossentropy",
              optimizer = optimizer,
              metrics=metrics)
```

```
Checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("save_model_xception_autoencoder.h5", save_best_only=True)
early_stopping = tf.keras.callbacks.EarlyStopping(
    patience=10,
    min_delta=0.001,
    restore_best_weights=True,
)

callbacks = [Checkpoint_cb, early_stopping]
```

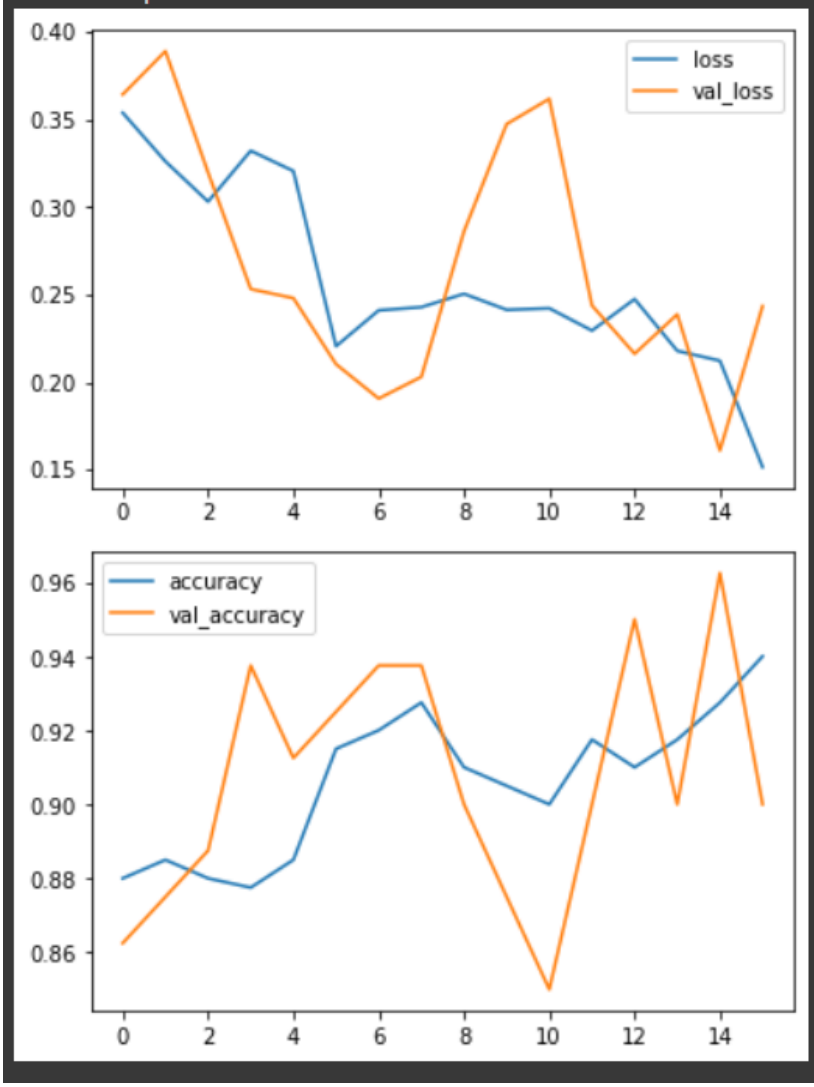
```
initial_epochs = 13
new_epochs = 15
total_epochs =  initial_epochs + new_epochs

history_fine = model.fit(x=denoise_train*255., y=y_train,validation_data=(X_val*255., y_val),
                         epochs=total_epochs,
                         initial_epoch=history.epoch[-1],
                         callbacks=callbacks)
```

```
Epoch 13/28
13/13 [==============================] - 25s 2s/step - loss: 0.3538 - accuracy: 0.8800 - val_loss: 0.3643 - val_accuracy: 0.8625
Epoch 14/28
13/13 [==============================] - 25s 2s/step - loss: 0.3261 - accuracy: 0.8850 - val_loss: 0.3890 - val_accuracy: 0.8750
Epoch 15/28
13/13 [==============================] - 25s 2s/step - loss: 0.3032 - accuracy: 0.8800 - val_loss: 0.3198 - val_accuracy: 0.8875
Epoch 16/28
13/13 [==============================] - 26s 2s/step - loss: 0.3322 - accuracy: 0.8775 - val_loss: 0.2533 - val_accuracy: 0.9375
Epoch 17/28
13/13 [==============================] - 25s 2s/step - loss: 0.3206 - accuracy: 0.8850 - val_loss: 0.2481 - val_accuracy: 0.9125
Epoch 18/28
13/13 [==============================] - 25s 2s/step - loss: 0.2207 - accuracy: 0.9150 - val_loss: 0.2104 - val_accuracy: 0.9250
Epoch 19/28
13/13 [==============================] - 25s 2s/step - loss: 0.2411 - accuracy: 0.9200 - val_loss: 0.1908 - val_accuracy: 0.9375
Epoch 20/28
13/13 [==============================] - 25s 2s/step - loss: 0.2429 - accuracy: 0.9275 - val_loss: 0.2032 - val_accuracy: 0.9375
Epoch 21/28
13/13 [==============================] - 26s 2s/step - loss: 0.2506 - accuracy: 0.9100 - val_loss: 0.2862 - val_accuracy: 0.9000
Epoch 22/28
13/13 [==============================] - 25s 2s/step - loss: 0.2414 - accuracy: 0.9050 - val_loss: 0.3474 - val_accuracy: 0.8750
Epoch 23/28
13/13 [==============================] - 25s 2s/step - loss: 0.2423 - accuracy: 0.9000 - val_loss: 0.3618 - val_accuracy: 0.8500
Epoch 24/28
13/13 [==============================] - 25s 2s/step - loss: 0.2296 - accuracy: 0.9175 - val_loss: 0.2439 - val_accuracy: 0.9000
Epoch 25/28
13/13 [==============================] - 25s 2s/step - loss: 0.2475 - accuracy: 0.9100 - val_loss: 0.2164 - val_accuracy: 0.9500
Epoch 26/28
13/13 [==============================] - 26s 2s/step - loss: 0.2180 - accuracy: 0.9175 - val_loss: 0.2388 - val_accuracy: 0.9000
Epoch 27/28
13/13 [==============================] - 25s 2s/step - loss: 0.2124 - accuracy: 0.9275 - val_loss: 0.1612 - val_accuracy: 0.9625
Epoch 28/28
13/13 [==============================] - 25s 2s/step - loss: 0.1517 - accuracy: 0.9400 - val_loss: 0.2436 - val_accuracy: 0.9000
```

در قسمت fine-tuning، ما نرخ یادگیری Adam Optimer خود را روی 0.0001 تنظیم می کنیم و تابع early stopping call back را در قسمت فیتینگ تنظیم می کنیم، سپس مدل را با 15 دوره دیگر مطابقت می دهیم. می توانید نتیجه را همانطور که در بالا نشان داده شده است مشاهده کنید.

```
history_df = pd.DataFrame(history_fine.history)
history_df.loc[:, ['loss', 'val_loss']].plot()
history_df.loc[:, ['accuracy', 'val_accuracy']].plot()
```

```
<AxesSubplot:>
```



ما 96.25% دقت بر روی دیتاست اعتبارسنجی و 92.75% دقت بر روی دیتاست آموزش داشتیم. ما تابع ضرر را بر اساس epoch که در بالا نشان داده شده است ترسیم کردیم.

```
model.load_weights('save_model_xception_autoencoder.h5')
```

ما بهترین مدل را به وسیله model.load_weights() لود کردیم.

```
model.evaluate(X_test*255., y_test)
```

```
4/4 [==============================] - 3s 769ms/step - loss: 0.7486 - accuracy: 0.7800
[0.7486359477043152, 0.7799999713897705]
```

سپس در اینجا مدل خود را روی مجموعه آزمایشی آزمایش کردیم که اینکد نشده است.

نتیجه می گیریم که ضرر برای این ارزیابی 0.7 و دقت 78 درصد است که مطلوب نیست.

دلیل چنین نتیجه ای، اینکدینگ بد است. تصاویر اصلی به اندازه کافی خالص هستند تا آموزش را انجام دهند و ما نیازی به حذف نویز تصویر با اتواینکدر نداریم. ما بسیاری از معماری رمزگذار/رمزگشا را با کرنل های مختلف انجام دادیم، اما نتیجه به اندازه کافی رضایت بخش نبود.

مدل بر روی تصاویر اولیه:

```
base_model = tf.keras.applications.xception.Xception(input_shape=(224,224,3), include_top=False, weights='imagenet')
base_model.trainable = True

input_shape = (224,224,3)

inputs = tf.keras.Input(shape=input_shape)
x = data_augmenter()(inputs)
x = tf.keras.applications.xception.preprocess_input(x)
x = base_model(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Flatten()(x)
outputs = layers.Dense(5,activation = 'softmax')(x)

final_model = tf.keras.Model(inputs, outputs)
```

بنابراین ما به این نتیجه رسیدیم که باید حذف نویز تصویر را متوقف کنیم و آموزش را با تصاویر اصلی خود انجام دهیم (فیتینگ بدون رمزگذار خودکار). ابتدا در بالا مدل خود را مودیفای کردیم.

```
final_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'],
)

history = final_model.fit(
    x=X_train_target*255.,
    y=y_train_target,
    validation_data=(X_val*255., y_val),
    callbacks = [lr_callback],
    epochs = EPOCHS,
    batch_size=16
)
```

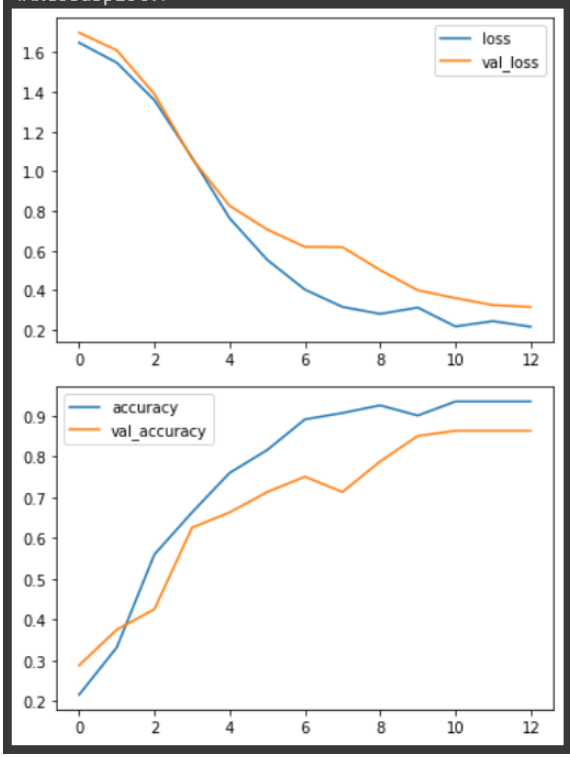
```
Epoch 00001: LearningRateScheduler setting learning rate to 1e-05.
Epoch 1/13
20/20 [==============================] - 52s 3s/step - loss: 1.6480 - accuracy: 0.2156 - val_loss: 1.6984 - val_accuracy: 0.2875 - lr: 1.0000e-05

Epoch 00002: LearningRateScheduler setting learning rate to 2.6666666666666667e-05.
Epoch 2/13
20/20 [==============================] - 49s 2s/step - loss: 1.5474 - accuracy: 0.3313 - val_loss: 1.6098 - val_accuracy: 0.3750 - lr: 2.6667e-05

Epoch 00003: LearningRateScheduler setting learning rate to 4.3333333333333334e-05.
Epoch 3/13
20/20 [==============================] - 50s 2s/step - loss: 1.3581 - accuracy: 0.5594 - val_loss: 1.3899 - val_accuracy: 0.4250 - lr: 4.3333e-05

Epoch 00004: LearningRateScheduler setting learning rate to 6e-05.
Epoch 4/13
20/20 [==============================] - 53s 3s/step - loss: 1.0691 - accuracy: 0.6625 - val_loss: 1.0634 - val_accuracy: 0.6250 - lr: 6.0000e-05

Epoch 00005: LearningRateScheduler setting learning rate to 6e-05.
Epoch 5/13
20/20 [==============================] - 51s 3s/step - loss: 0.7631 - accuracy: 0.7594 - val_loss: 0.8259 - val_accuracy: 0.6625 - lr: 6.0000e-05

Epoch 00006: LearningRateScheduler setting learning rate to 6e-05.
Epoch 6/13
20/20 [==============================] - 49s 2s/step - loss: 0.5532 - accuracy: 0.8156 - val_loss: 0.7065 - val_accuracy: 0.7125 - lr: 6.0000e-05

Epoch 00007: LearningRateScheduler setting learning rate to 6e-05.
Epoch 7/13
20/20 [==============================] - 48s 2s/step - loss: 0.4037 - accuracy: 0.8906 - val_loss: 0.6187 - val_accuracy: 0.7500 - lr: 6.0000e-05

Epoch 00008: LearningRateScheduler setting learning rate to 3.5000000000000004e-05.
Epoch 8/13
20/20 [==============================] - 49s 2s/step - loss: 0.3161 - accuracy: 0.9062 - val_loss: 0.6176 - val_accuracy: 0.7125 - lr: 3.5000e-05
```

```
Epoch 00009: LearningRateScheduler setting learning rate to 2.25e-05.
Epoch 9/13
20/20 [==============================] - 49s 2s/step - loss: 0.2810 - accuracy: 0.9250 - val_loss: 0.5023 - val_accuracy: 0.7875 - lr: 2.2500e-05

Epoch 00010: LearningRateScheduler setting learning rate to 1.6250000000000002e-05.
Epoch 10/13
20/20 [==============================] - 48s 2s/step - loss: 0.3128 - accuracy: 0.9000 - val_loss: 0.4004 - val_accuracy: 0.8500 - lr: 1.6250e-05

Epoch 00011: LearningRateScheduler setting learning rate to 1.3125e-05.
Epoch 11/13
20/20 [==============================] - 49s 2s/step - loss: 0.2177 - accuracy: 0.9344 - val_loss: 0.3605 - val_accuracy: 0.8625 - lr: 1.3125e-05

Epoch 00012: LearningRateScheduler setting learning rate to 1.1562500000000002e-05.
Epoch 12/13
20/20 [==============================] - 49s 2s/step - loss: 0.2444 - accuracy: 0.9344 - val_loss: 0.3252 - val_accuracy: 0.8625 - lr: 1.1562e-05

Epoch 00013: LearningRateScheduler setting learning rate to 1.0781250000000001e-05.
Epoch 13/13
20/20 [==============================] - 48s 2s/step - loss: 0.2162 - accuracy: 0.9344 - val_loss: 0.3160 - val_accuracy: 0.8625 - lr: 1.0781e-05
```

این دفعه مدل را بر روی تصاویر اصلی که حذف نویز نشده اند، فیت کردیم. لایه های dropout را باز کردیم.

```
history_df = pd.DataFrame(history.history)

history_df.loc[:, ['loss', 'val_loss']].plot()
history_df.loc[:, ['accuracy', 'val_accuracy']].plot()
```

```
<AxesSubplot:>
```



ما 93.4% دقت بر روی دیتاست اعتبارسنجی و 86.25%دقت بر روی دیتاست آموزش داشتیم. ما تابع ضرر را بر اساس epoch که در بالا نشان داده شده است ترسیم کردیم.

```
base_model2 = final_model.layers[4]
base_model2.trainable = True

fine_tune_at = 90

for layer in base_model2.layers[:fine_tune_at]:
    layer.trainable = False
```

مانند قبل fine-tuning را انجام میدهیم فقط این بار 42 لایه آخر را آموزش دادیم.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
metrics= ['accuracy']
final_model.compile(loss="sparse_categorical_crossentropy",
              optimizer = optimizer,
              metrics=metrics)

Checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("save_model_xception_without_autoencoder.h5", save_best_only=True)
early_stopping = tf.keras.callbacks.EarlyStopping(
    patience=10,
    min_delta=0.001,
    restore_best_weights=True,
)

callbacks = [Checkpoint_cb, early_stopping]
```

بهینه ساز، تابع ضرر و کال بک ها را مانند دفعه قبلی ست میکنیم.

```
initial_epochs = 13
new_epochs = 30
total_epochs =  initial_epochs + new_epochs

history_fine = final_model.fit(x=X_train_target*255., y=y_train_target,validation_data=(X_val*255., y_val), epochs=total_epochs,
                               initial_epoch=history.epoch[-1],
                               callbacks=callbacks)
```

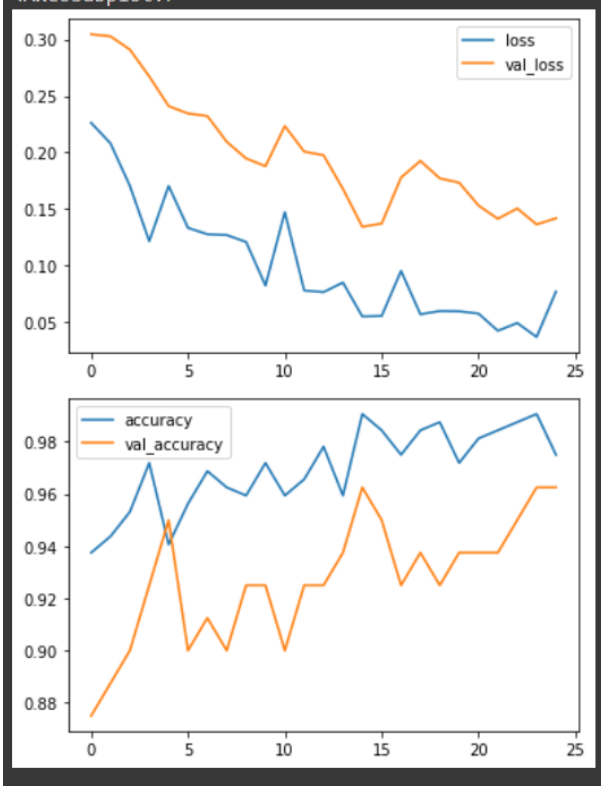
```
Epoch 13/43
10/10 [==============================] - 23s 2s/step - loss: 0.2262 - accuracy: 0.9375 - val_loss: 0.3046 - val_accuracy: 0.8750
/opt/homebrew/Caskroom/miniforge/base/envs/env1/lib/python3.10/site-packages/keras/engine/functional.py:1410: CustomMaskWarning: Custom mask layers require a config and
  layer_config = serialize_layer_fn(layer)
Epoch 14/43
10/10 [==============================] - 23s 2s/step - loss: 0.2080 - accuracy: 0.9438 - val_loss: 0.3027 - val_accuracy: 0.8875
Epoch 15/43
10/10 [==============================] - 23s 2s/step - loss: 0.1705 - accuracy: 0.9531 - val_loss: 0.2910 - val_accuracy: 0.9000
Epoch 16/43
10/10 [==============================] - 23s 2s/step - loss: 0.1216 - accuracy: 0.9719 - val_loss: 0.2673 - val_accuracy: 0.9250
Epoch 17/43
10/10 [==============================] - 23s 2s/step - loss: 0.1705 - accuracy: 0.9406 - val_loss: 0.2411 - val_accuracy: 0.9500
Epoch 18/43
10/10 [==============================] - 23s 2s/step - loss: 0.1334 - accuracy: 0.9563 - val_loss: 0.2345 - val_accuracy: 0.9000
Epoch 19/43
10/10 [==============================] - 23s 2s/step - loss: 0.1278 - accuracy: 0.9688 - val_loss: 0.2323 - val_accuracy: 0.9125
Epoch 20/43
10/10 [==============================] - 23s 2s/step - loss: 0.1271 - accuracy: 0.9625 - val_loss: 0.2095 - val_accuracy: 0.9000
Epoch 21/43
10/10 [==============================] - 23s 2s/step - loss: 0.1210 - accuracy: 0.9594 - val_loss: 0.1948 - val_accuracy: 0.9250
Epoch 22/43
10/10 [==============================] - 23s 2s/step - loss: 0.0824 - accuracy: 0.9719 - val_loss: 0.1879 - val_accuracy: 0.9250
Epoch 23/43
10/10 [==============================] - 23s 2s/step - loss: 0.1473 - accuracy: 0.9594 - val_loss: 0.2234 - val_accuracy: 0.9000
Epoch 24/43
10/10 [==============================] - 23s 2s/step - loss: 0.0781 - accuracy: 0.9656 - val_loss: 0.2008 - val_accuracy: 0.9250
Epoch 25/43
10/10 [==============================] - 23s 2s/step - loss: 0.0766 - accuracy: 0.9781 - val_loss: 0.1976 - val_accuracy: 0.9250
Epoch 26/43
10/10 [==============================] - 23s 2s/step - loss: 0.0851 - accuracy: 0.9594 - val_loss: 0.1679 - val_accuracy: 0.9375
Epoch 27/43
10/10 [==============================] - 23s 2s/step - loss: 0.0551 - accuracy: 0.9906 - val_loss: 0.1344 - val_accuracy: 0.9625
```

```
Epoch 28/43
10/10 [==============================] - 23s 2s/step - loss: 0.0555 - accuracy: 0.9844 - val_loss: 0.1372 - val_accuracy: 0.9500
Epoch 29/43
10/10 [==============================] - 23s 2s/step - loss: 0.0954 - accuracy: 0.9750 - val_loss: 0.1779 - val_accuracy: 0.9250
Epoch 30/43
10/10 [==============================] - 23s 2s/step - loss: 0.0571 - accuracy: 0.9844 - val_loss: 0.1926 - val_accuracy: 0.9375
Epoch 31/43
10/10 [==============================] - 23s 2s/step - loss: 0.0598 - accuracy: 0.9875 - val_loss: 0.1772 - val_accuracy: 0.9250
Epoch 32/43
10/10 [==============================] - 23s 2s/step - loss: 0.0596 - accuracy: 0.9719 - val_loss: 0.1734 - val_accuracy: 0.9375
Epoch 33/43
10/10 [==============================] - 23s 2s/step - loss: 0.0576 - accuracy: 0.9812 - val_loss: 0.1531 - val_accuracy: 0.9375
Epoch 34/43
10/10 [==============================] - 23s 2s/step - loss: 0.0424 - accuracy: 0.9844 - val_loss: 0.1414 - val_accuracy: 0.9375
Epoch 35/43
10/10 [==============================] - 23s 2s/step - loss: 0.0493 - accuracy: 0.9875 - val_loss: 0.1505 - val_accuracy: 0.9500
Epoch 36/43
10/10 [==============================] - 23s 2s/step - loss: 0.0369 - accuracy: 0.9906 - val_loss: 0.1364 - val_accuracy: 0.9625
Epoch 37/43
10/10 [==============================] - 23s 2s/step - loss: 0.0771 - accuracy: 0.9750 - val_loss: 0.1419 - val_accuracy: 0.9625
```

برای بیش از 30 epoch، فیت شد.

```
history_df = pd.DataFrame(history_fine.history)
history_df.loc[:, ['loss', 'val_loss']].plot()
history_df.loc[:, ['accuracy', 'val_accuracy']].plot()
```

```
<AxesSubplot:>
```



ما 99% دقت بر روی دیتاست اعتبارسنجی و 96.25% دقت بر روی دیتاست آموزش داشتیم. ما تابع ضرر را بر اساس epoch که در بالا نشان داده شده است ترسیم کردیم.

```
model.load_weights('save_model_xception_without_autoencoder.h5')
```

بهترین مدل را در بالا لود کردیم.

```
[ ]  model.evaluate(X_test*255., y_test)

    4/4 [==============================] - 4s 762ms/step - loss: 0.1282 - accuracy: 0.9400
    [0.1282070279121399, 0.9399999976158142]
```

مدل خود را بر روی دیتاست تست در بالا ارزیابی کردیم.

پس دقت نهایی بر روی دیتاست تست 94% است که قابل قبول است.

ارزیابی:

```
test_predicts = final_model.predict(X_test*255.)

y_predict = np.argmax(test_predicts,axis=1)

class_names

test_predicts

y_test
```

در ابتدا در بالا ما کلاس ها را برای مجموعه داده تست پیش بینی کردیم.

```
plt.figure(figsize=(18,12))

for i in range(25):
    plt.subplot(5,5,i+1)
    plt.imshow(X_test[i])
    plt.title(f"Prediction: {class_names[y_test[i]]}")
    plt.axis("off")
plt.tight_layout()
plt.show()
```

برای دیدن چندی از پیش بینی های مدلمان.

```
from sklearn import metrics
print("Test set Accuracy: ", metrics.accuracy_score(y_test, y_predict))

from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
print (classification_report(y_test, y_predict))
cm = confusion_matrix(y_test, y_predict)
fig, ax = plt.subplots(figsize=(4,4))
sns.heatmap(cm, annot=True)
plt.show()

from sklearn.metrics import f1_score
print('f1 is: ',f1_score(y_test, y_predict, average='macro'))
```

اینجا ما ماتریس سردرگمی را برای پیش بینی خود داریم.

نتایج به گونه ای واضح است که می توانید ببینید که مدل در مجموعه داده آزمایشی در کجا به خوبی پیش بینی نمی کند است.

```
data = [['model with autoencoder', 78], ['model without autoencoder', 94], ['model of the article', 97.6]]

df = pd.DataFrame(data, columns=['model', 'accuracy(%)'])

df
```

سپس در پایان می‌توانیم دیتافریمی از دقت مدل‌های خود در مقایسه با دقت مقاله (به درصد) داشته باشیم.

نتیجه می گیریم که مدل اصلی ما 94 درصد دقت در مجموعه داده های تست و 99 درصد دقت در مجموعه آموزشی دارد که در مقایسه با مقاله با دقت 97.6 درصد خوب است.