Report

Yasmin Ahmadi

Introduction

The dataset for this project is five classes of grapevine leaves: Ak, Ala_Idris, Buzgulu, Dimnit, and Nazli. Grapevine leaves are harvested once a year as a by-product. The species of grapevine leaves are important in terms of price and taste, So it is important to be correctly classified. Therefore, using images of grapevine leaves we are trying to classify them with a high accuracy. For this purpose, images of 500 vine leaves belonging to 5 species were taken which this number is increased to 2800 using Data Augmentation. The classification was conducted with a built CNN, a pretrained MobileNetv2 CNN model, VGG16. Using them the accuracy of 90% was achieved. Then using Autoencoders some improvements were made on the initial CNN model.

Reading data from Google Drive

The dataset has to be uploaded to Google Drive. Then files from Google Drive are imported in Colab. So The first step is mounting your Google Drive by:

```
from google.colab import drive
drive.mount('/content/drive')
```

Run above two lines of code and get the authorization code by loggin into your Google account. Then, paste the authorization code and press Enter.

Importing files

Now we can import files from the Google Drive by copying their paths:

```
input_dir="/content/drive/MyDrive/Colab Notebooks/Grapevine_Leaves_Image_Dataset"
```

Importing libraries

```
import keras
%matplotlib inline
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing import image
import tensorflow as tf
```

```
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.optimizers import RMSprop,SGD,Adam
import splitfolders
import tensorflow_addons as tfa
from sklearn.model_selection import train_test_split
from tensorflow.keras import backend as K
from PIL import Image
```

calling text_dataset_from_directory will return a tf.data Dateset that yields batches of texts from the subdirectories. Batch_size and seed are 32 and 1234 respectively. Label_mode is categorical, as we have 5 different label names.

```
all_ds = tf.keras.preprocessing.image_dataset_from_directory(
    input_dir,
    seed=SEED,
    label_mode="categorical",
    image_size=image_size,
    batch_size=batch_size,
)
```

There are 500 files belonging to 5 classes.
Then we separate features from their labels:

```
X_all = []
y_all = []

for x,y in all_ds.unbatch():
  X_all.append(x)
  y_all.append(y)

X_all = np.array(X_all)
y_all = np.array(y_all)
```

Now %80 of data has to go to train and the rest to test; So we use the train_test_split function.

```
X_train_old,X_test,y_train_old,y_test = train_test_split(X_all, y_all, test_size=0.2)
```

Next we have Data Augmentation. Image rotation is one of the widely used augmentation techniques and allows the model to become invariant to the orientation of the object. ImageDataGenerator class allows you to randomly rotate images through any degree between 0 and 360 by providing an integer value in the **rotation_range** argument. When the image is rotated, some pixels will move outside the image and leave an empty area that needs to be filled in. You can fill this in different ways like a constant value or nearest pixel values, etc. This is specified in the **fill_mode** argument and the default value is **“nearest”** which simply replaces the empty area with the nearest pixel values.

Random Shifts

It may happen that the object may not always be in the center of the image. To overcome this problem we can shift the pixels of the image either horizontally or vertically; this is done by adding a certain constant value to all the pixels. ImageDataGenerator class has the argument **height_shift_range** for a vertical shift of image and **width_shift_range** for a horizontal shift of image.

If the value is a float number, that would indicate the percentage of width or height of the image to shift. Otherwise, if it is an integer value, then simply the width or height are shifted by those many pixel values.

Random Flips

Flipping images is also a great augmentation technique and it makes sense to use it with a lot of different objects. ImageDataGenerator class has parameters **horizontal_flip** and **vertical_flip** for flipping along the vertical or the horizontal axis. However, this technique should be according to the object in the image. For example, vertical flipping of a car would not be a sensible thing compared to doing it for a symmetrical object like football or something else. Having said that, I am going to flip my image in both ways just to demonstrate the effect of the augmentation.

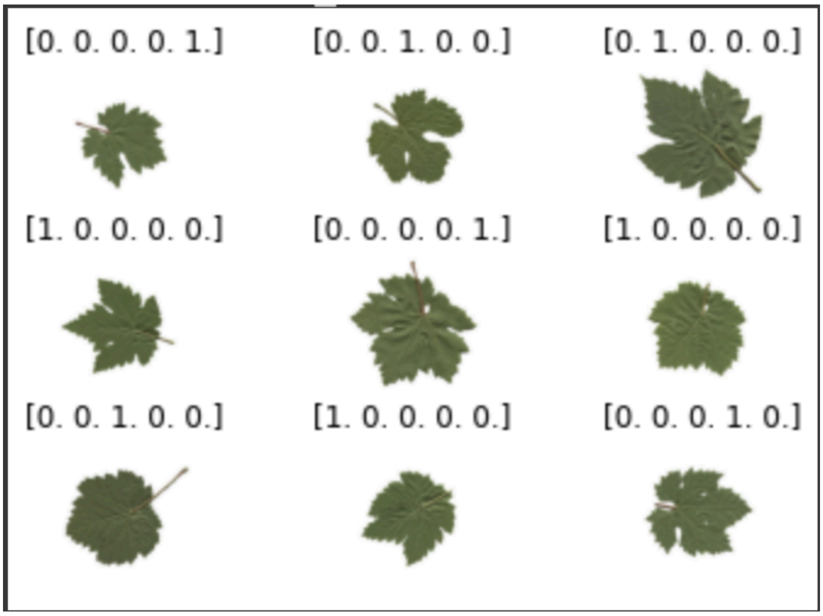
Random Zoom

The zoom augmentation either randomly zooms in on the image or zooms out of the image. ImageDataGenerator class takes in a float value for zooming in the **zoom_range** argument. You could provide a list with two values specifying the lower and the upper limit. Else, if you specify a float value, then zoom will be done in the range [1-zoom_range,1+zoom_range]. Any value smaller than 1 will zoom in on the image. Whereas any value greater than 1 will zoom out on the image.

```
train_datagen = ImageDataGenerator(width_shift_range=0.2,
                    height_shift_range=0.2,
                    zoom_range=0.2,
                    fill_mode='nearest',
```

```
                   horizontal_flip = True,
                   vertical_flip = True,
                   rescale=1./255)
```

Below we can see a picture of some leaves after Data Augmentation. The arrays correspond to different classes of leaves. If the first entry is 1, then the leaf belongs to the class 'Ak',
if the second entry is 1 → Ala_Idris
If the third entry is 1 → Buzgulu
If the forth entry is 1 → Dimnit
If the fifth entry is 1 → Nazli.



Now that we're done with Data Augmentation, it's time for Train Set Validation Split:

```
X_train,X_valid,y_train,y_valid = train_test_split(X_train, y_train, test_size=0.2)
```

Building a convolutional Neural Network model using TensorFlow:

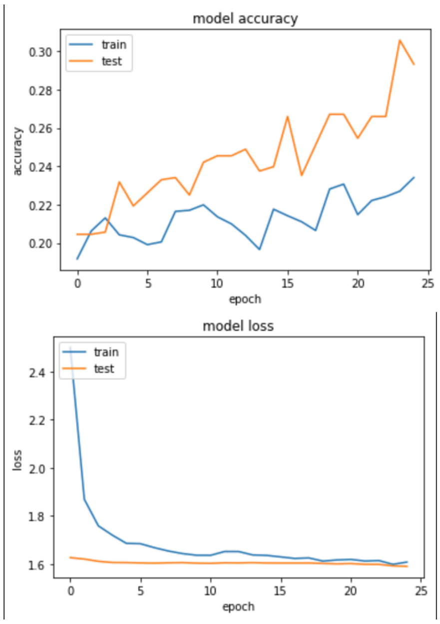
```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16,(3,3),activation = "relu" , input_shape = (img_size,img_size,3)) ,
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32,(3,3),activation = "relu") ,
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
```

```
    tf.keras.layers.Conv2D(64,(3,3),activation = "relu") ,
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64,(3,3),activation = "relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1000,activation="relu"),
    tf.keras.layers.Dropout(0.5,seed = SEED),
    tf.keras.layers.Dense(500,activation="relu"),
    tf.keras.layers.Dropout(0.5,seed = SEED),
    tf.keras.layers.Dense(400,activation ="relu"),
    tf.keras.layers.Dropout(0.5,seed = SEED),
    tf.keras.layers.Dense(300,activation="relu"),
    tf.keras.layers.Dropout(0.5,seed = SEED),
    tf.keras.layers.Dense(200,activation ="relu"),
    tf.keras.layers.Dropout(0.3,seed = SEED),
    tf.keras.layers.Dense(5,activation = "softmax")
])
```

Now fitting the model to data, with the learning rate of 0.001 :

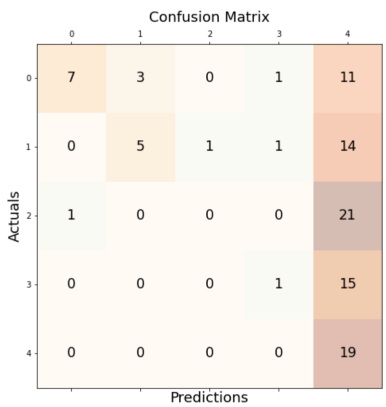
```
history = model.fit(X_train, y_train,
                    validation_data=(X_valid,y_valid),
                    steps_per_epoch=len(X_train)//batch_size,
                    epochs=25,
                    validation_steps=len(X_valid)//batch_size,
                    verbose=1)
```

Below you can see the Accuracy and Loss function of the network:



After training the model is examined by prediction on the test set. The accuracy for train set is %32.

To do further analyzing, we can also create a confusion matrix:



We can also check the recall and F1score of the data:

	precision	recall	f1-score	support
0	0.88	0.32	0.47	22
1	0.62	0.24	0.34	21
2	0.00	0.00	0.00	22
3	0.33	0.06	0.11	16
4	0.24	1.00	0.38	19
accuracy			0.32	100
macro avg	0.41	0.32	0.26	100
weighted avg	0.42	0.32	0.26	100

In order to improve the accuracy, we will utilize the pre-trained VGG16 model, which is a convolutional neural network trained on 1.2 million images to classify 1000 different categories. Since the domain and task for VGG16 are similar to our domain and task, we can use its pre-trained network to do the job. We also turn the trainability of layers off since we want to utilize the weights of pertained models.

```
model2 = tf.keras.models.Sequential()
VGG = VGG16(input_shape = (224, 224, 3), # Shape of our images
            pooling="avg",
            include_top = False, # Leave out the last fully connected layer
            weights = 'imagenet')
```

We want to use the already trained parameters (weights) and not change them:

```
for layer in VGG.layers:
  layer.trainable = False
```

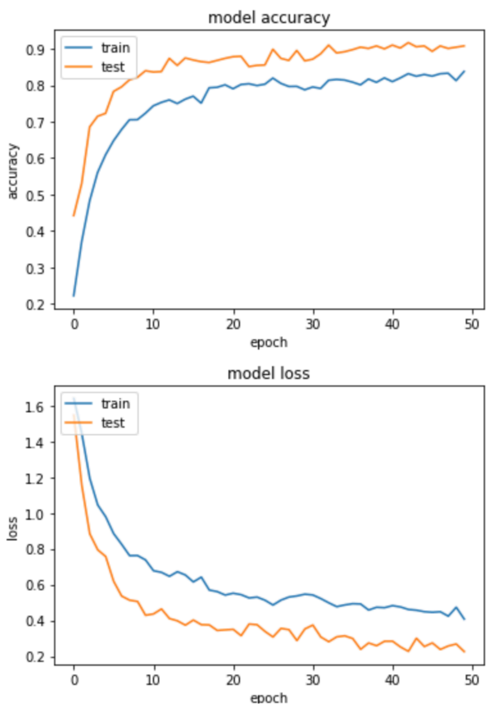
Adding some costumed dense layers to end of it:

```
model1.add(VGG)
model1.add(tf.keras.layers.Dense(256,activation="relu"))
model1.add(tf.keras.layers.Dropout(0.5))
model1.add(tf.keras.layers.Dense(256,activation="relu"))
model1.add(tf.keras.layers.Dropout(0.5))
model1.add(tf.keras.layers.Dense(128,activation="relu"))
model1.add(tf.keras.layers.Dropout(0.5))
model1.add(tf.keras.layers.Dense(5,activation="softmax"))
```

now compiling the model:

```
model1.compile(optimizer=Adam(learning_rate= 0.001), loss='categorical_crossentropy', metrics = ['acc'])
```

Below you can see the Accuracy and Loss function of the network:



After training the model is examined by prediction on the test set:

```
result = model1.evaluate(X_test, y_test)
print(dict(zip(model1.metrics_names, result)))
y_pred = model1.predict(X_test)
get_confusion_matrix(y_test, y_pred)
```

```
4/4 [==============================] - 1s 406ms/step - loss: 0.3511 - acc: 0.9000
{'loss': 0.3511497974395752, 'acc': 0.8999999761581421}
```

As can be seen from the picture above, the accuracy got increased all the way to almost 90%.

Next We will utilize the pre-trained model also used in the paper, mobilenetv2. We also add some dense layers to end of it (the last one being the activation).

```
model2 = tf.keras.models.Sequential()
mobilenetv2 = tf.keras.applications.MobileNetV2(
    input_shape=(img_size,img_size,3),
    include_top=False,
    weights='imagenet',
    pooling='avg',
)
```

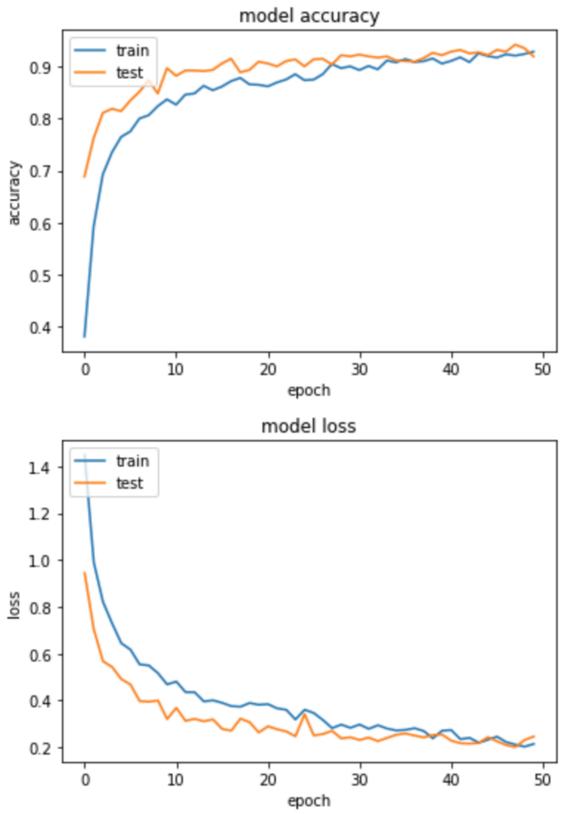
We want to use the already trained parameters (weights) and not change them:

```
for layer in mobilenetv2.layers:
  layer.trainable = False
```

Adding some costumed dense layers to end of it:

```
model2.add(mobilenetv2)
model2.add(tf.keras.layers.Dense(256,activation="relu"))
model2.add(tf.keras.layers.Dropout(0.5))
model2.add(tf.keras.layers.Dense(256,activation="relu"))
model2.add(tf.keras.layers.Dropout(0.5))
model2.add(tf.keras.layers.Dense(128,activation="relu"))
model2.add(tf.keras.layers.Dropout(0.5))
model2.add(tf.keras.layers.Dense(5,activation="softmax"))
```

Below you can see the Accuracy and Loss function of the network:



After training the model is examined by prediction on the test set:

```
result = model2.evaluate(X_test, y_test)
print(dict(zip(model2.metrics_names, result)))
y_pred = model2.predict(X_test)
get_confusion_matrix(y_test, y_pred)
```

```
4/4 [==============================] - 0s 111ms/step - loss: 0.3063 - acc: 0.8800
{'loss': 0.30625876784324646, 'acc': 0.8799999952316284}
tf.Tensor(
[[21  1  1  0  0]
 [ 1 19  5  0  0]
 [ 0  1 12  0  1]
 [ 0  0  0 19  1]
 [ 0  0  1  0 17]], shape=(5, 5), dtype=int32)
```

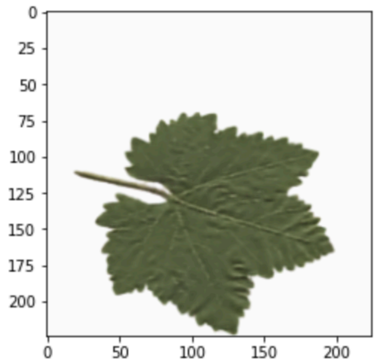
which has also improved from the accuracy of the initial CNN to %88.

Now moving on to Autoencoders, Autoencoder is an unsupervised artificial neural network that compresses the data to lower dimension and then reconstructs the input back. Autoencoder finds the representation of the data in a lower dimension by focusing more on the important features getting rid of noise and redundancy. It's based on Encoder-Decoder architecture, where encoder encodes the high-dimensional data to lower-dimension and decoder takes the lower-dimensional data and tries to reconstruct the original high-dimensional data.

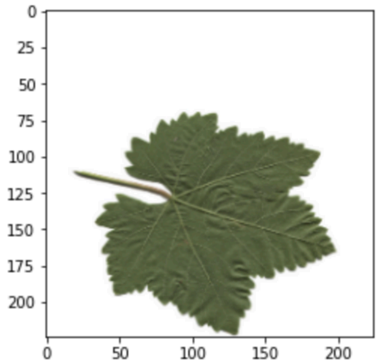
I first tried Denoising auto encoders but using Denoising auto encoders was not really helpful with this data! If the data was noisy it may have worked better.

So moving on to Dimensionality reduction Atuoencoders (which prevent overfitting), When dealing with high dimensional data, it is often useful to reduce the dimensionality by projecting the data to a lower dimensional subspace which captures the essence of the data.

Using Dimensionality reduction Atuoencoders we can do encoding, so data is reduced, then we can give the reduced data to the model again and do training.



Above you can see the autoencoded (reduced) leaf which looks almost like the original picture below:



so the reducing is done well as it’s preserving the instances while reducing the dimensionality.

The model we choose to give the reduced data to, is the very first CNN (as the pretrained CNNs already have high accuracies, the built CNN is the one needing improvements)
So the model is rebuilt here:

```
model5 = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16,(3,3),activation = "relu" , input_shape = (56,56,32)) ,
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32,(3,3),activation = "relu") ,
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64,(3,3),activation = "relu") ,
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64,(3,3),activation = "relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
```

```
    tf.keras.layers.Dense(1000,activation="relu"),
    tf.keras.layers.Dropout(0.5,seed = SEED),
    tf.keras.layers.Dense(500,activation="relu"),
    tf.keras.layers.Dropout(0.5,seed = SEED),
    tf.keras.layers.Dense(400,activation ="relu"),
    tf.keras.layers.Dropout(0.5,seed = SEED),
    tf.keras.layers.Dense(300,activation="relu"),
    tf.keras.layers.Dropout(0.5,seed = SEED),
    tf.keras.layers.Dense(200,activation ="relu"),
    tf.keras.layers.Dropout(0.3,seed = SEED),
    tf.keras.layers.Dense(5,activation = "softmax")
])
```

And then instead of training on the regular data it is trained on the reduces X:

```
history5 = model5.fit(X_reduced_train, y_train,
                validation_data=(X_reduced_valid,y_valid),
                steps_per_epoch=len(X_reduced_train)//batch_size,
                epochs=50,
                validation_steps=len(X_reduced_valid)//batch_size,
                verbose=1)
```

And the prediction is now so much better on the test data with the help of autoencoders. Using Autoencoders the accuracy increased more than double! As can be seen below:

```
result = model5.evaluate(X_reduced_test, y_test)
print(dict(zip(model5.metrics_names, result)))
y_pred = model5.predict(X_reduced_test)
get_confusion_matrix(y_test, y_pred)
```

```
4/4 [==============================] - 0s 5ms/step - loss: 2.3791 - acc: 0.6600
{'loss': 2.3791327476501465, 'acc': 0.6600000262260437}
4/4 [==============================] - 0s 4ms/step
tf.Tensor(
[[14  0  2  0  0]
 [ 1  9  2  4  4]
 [ 1  0 17  4  0]
 [ 0  0  5 11  6]
 [ 1  0  1  3 15]], shape=(5, 5), dtype=int32)
```

Lastly 10Fold Cross Validation is used.
The accuracy of the first fold using the first CNN:

```
288/288 [==============================] - 2s 8ms/step - loss: 1.5964 - acc: 0.2476 - val_loss: 1.5597 - val_acc: 0.3300
Epoch 15/20
288/288 [==============================] - 2s 8ms/step - loss: 1.5728 - acc: 0.2757 - val_loss: 1.5259 - val_acc: 0.3900
Epoch 16/20
288/288 [==============================] - 2s 8ms/step - loss: 1.5443 - acc: 0.2844 - val_loss: 1.4991 - val_acc: 0.3800
Epoch 17/20
288/288 [==============================] - 2s 8ms/step - loss: 1.5280 - acc: 0.3014 - val_loss: 1.5042 - val_acc: 0.3300
Epoch 18/20
288/288 [==============================] - 2s 8ms/step - loss: 1.5240 - acc: 0.3014 - val_loss: 1.4803 - val_acc: 0.3400
Epoch 19/20
288/288 [==============================] - 2s 8ms/step - loss: 1.4840 - acc: 0.3326 - val_loss: 1.4482 - val_acc: 0.3600
Epoch 20/20
288/288 [==============================] - 2s 8ms/step - loss: 1.4680 - acc: 0.3389 - val_loss: 1.4219 - val_acc: 0.4000
<keras.callbacks.History object at 0x000001EB04051E50>
4/4 [==============================] - 0s 15ms/step - loss: 1.4081 - acc: 0.3900
{'loss': 1.408110499382019, 'acc': 0.38999998569488525}
4/4 [==============================] - 0s 3ms/step
tf.Tensor(
[[ 5  8  0  1  4]
 [ 2 10  0  0  8]
 [ 0  0  1  1 11]
 [ 2  3  2  0 18]
 [ 0  0  1  0 23]], shape=(5, 5), dtype=int32)
```