PLANT classificator

Mattia Spazzoli

Indice

1 PlantClassificator

Per concludere il mio percorso legato all'esame di Sistemi Digitali M, ho deciso di sviluppare un progetto che facesse uso della libreria open source Tensorflow.

L'idea che si è voluto realizzare, grazie ad una lunga tradizione familiare contadina, è quella di progettare un classificatore per piante da orto, che ricevendo in ingresso un'immagine raffigurante una foglia, fosse in grado di riconoscerne la specie. Un'altra specifica fondamentale di questo elaborato era riuscire ad utilizzare questa rete neurale attraverso un dispositivo embedded, per questo, si è deciso di utilizzare un vecchio telefono con sistema operativo Android per testarne l'efficacia. Naturalmente, per utilizzare il modello addestrato su un device di questo tipo è stato necessario lo sviluppo di una semplice applicazione, che una volta inglobato il classificatore al suo interno, ne gestisse i dati in ingresso e in uscita, rendendoli fruibili all'utente.

2 Sviluppo della rete neurale

2.1 Dataset

La rete neurale desiderata è stata realizzata attraverso l'utilizzo di un dataset di immagini già esistente e consultabile sul sito web www.kaggle.com, un portale che contiene risorse dedicate a progetti di data science.

Il nome della raccolta di immagini utilizzata è "PlantVillage Dataset" ed è rappresentata da un insieme di 50000 figure suddivise in 38 etichette. Alcune di queste, rappresentano foglie di piante da orto in uno stato di salute, mentre in altri casi, in uno stato di malattia. Lo scopo di questo dataset è quello di aumentare la ricerca nel campo della produzione di ortaggi, per studiarne e comprenderne sempre più il comportamento delle malattie infettive, che ogni anno influiscono significativamente sulla resa del raccolto, riducendola del 40% circa.

Come precedentemente introdotto però, l' obiettivo di questo progetto era quello di realizzare un classificatore in grado di riconoscere la pianta in base alla foglia fornita, per cui, si è scelto di utilizzare soltanto immagini di foglie sane, nello specifico, quelle di:

- Mela
- Mirtillo
- Mais
- Uva
- Pepe
- Patata
- Mora
- Fragola
- Pomodoro

Le immagini utilizzate sono tutte di dimensione 256x256 pixel e rappresentate in formato JPEG.

2.2 Ambiente di sviluppo e prerequisiti

La realizzazione e l'addestramento del modello Tensorflow sono avvenute tramite l'ambiente di sviluppo Jupyter, un sistema interattivo web-based per la scrittura di notebook e l'esecuzione di codice Python. Il suo utilizzo è stato possibile attraverso l'installazione di Anaconda3 sul mio pc personale, avente processore Intel Core i7 di quinta generazione ed il sistema operativo Windows 10.

Le librerie installate prima della scrittura del netbook invece sono le seguenti: Tensorflow 2.5, Matplotlib e Numpy.

2.3 Preprocessing

2.3.1 Caricamento del dataset

Sono state inserite all'interno di una cartella di nome "dataset_pv" le immagini che si volevano classificare, suddivise per tipologia di foglia in apposite sottocartelle, in cui il nome rappresentava l'etichetta corrispondente. Successivamente, per caricare il dataset all'interno del netbook, si è utilizzata la primitiva di Keras "image_dataset_from_directory", come mostrato in seguito:

```
#IMPORT DATASET WITH LABELS FROM DIRECTORY
dataset = tf.keras.preprocessing.image_dataset_from_directory(
    "dataset_pv",
    shuffle=True,
    image_size = (IMAGE_SIZE, IMAGE_SIZE),
    batch_size = BATCH_SIZE,
)
```

Nello specifico qui vengono usati batch da 32 immagini di dimensione pari a quella delle immagini del dataset. Le 9140 figure caricate sono state poi suddivise in tre sottodataset, dedicati rispettivamente a training, validation e testing, con una proporzione dell'80%, 10% e 10%.

2.3.2 Resizing, Rescaling, Flip and Rotation

Prima di applicare le vere e proprie funzioni di preprocessing sui dati, ne viene fatto un prefetch così da velocizzare le operazioni di training successive, evitando il loro caricamento ad ogni epoca.

```
#FETCH SOME FUNCTIONS TO USE THEM IN FOLLOW EPOCHS
train_ds=train_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
val_ds=val_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
test_ds=test_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
```

In seguito, tramite la primitiva "Sequential" di Keras, vengono definiti due layers, che verranno applicati ai dati in ingresso prima che essi vengano analizzati dalla rete neurale. Il primo, è stato denominato "resize_and_rescale" ed ha il fine di portare tutte le immagini di ingresso ad una dimensione desiderata e di scalare i valori RGB dei tensori in un intervallo definito tra 0 ed 1.

```
#SCALE RGB TENSOR TO 0-1 VALUES
resize_and_rescale= tf.keras.Sequential([layers.experimental.preprocessing.Resizing(IMAGE_SIZE, IMAGE_SIZE),
                                         layers.experimental.preprocessing.Rescaling(1.0/255)])
```

Il secondo invece, è stato chiamato “data_augmentation” ed è la sequenza di due livelli, che tramite la rotazione ed il capovolgimento delle immagini di training vuole rendere la predizione delle label più robusta.

```
#FLIP AND ROTATE RANDOMLY IMAGES
data_augmentation=tf.keras.Sequential([layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
                                       layers.experimental.preprocessing.RandomRotation(0.2)])
```

2.4 Architettura e addestramento

Di seguito viene illustrata la definizione dell’architettura della rete neurale convoluzionale che, applicata ripetitivamente ai dati di training e con pesi sempre meglio calibrati, ha centrato l’obiettivo di addestrare il modello di riconoscimento automatico desiderato.

Essa è composta da sei livelli convoluzionali e sei di pooling, alternati tra loro e con rango rispettivamente a 3 e 2 pixel. Infine, tramite le funzioni “Dense” di Keras viene applicata la funzione di attivazione aggiornata e ricalcolate le probabilità di output delle dieci labels.

```
#CNN ARCHITECTURE CREATE
input_shape=(BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
n_classes = len(class_names)
model = models.Sequential([
    #Applies preprocessing layers
    resize_and_rescale,

    #Applies CNN layers
    layers.Conv2D(32,(3,3), activation='relu', input_shape=input_shape), #Convolutional layer
    layers.MaxPooling2D((2,2)), #Pooling layer
    layers.Conv2D(64, kernel_size=(3,3), activation='relu'),  #Convolutional layer
    layers.MaxPooling2D((2,2)), #Pooling layer
    layers.Conv2D(64, kernel_size=(3,3), activation='relu'),  #Convolutional layer
    layers.MaxPooling2D((2,2)), #Pooling layer
    layers.Conv2D(64,(3,3), activation='relu'),  #Convolutional layer
    layers.MaxPooling2D((2,2)), #Pooling layer
    layers.Conv2D(64,(3,3), activation='relu'),  #Convolutional layer
    layers.MaxPooling2D((2,2)), #Pooling layer
    layers.Conv2D(64,(3,3), activation='relu'),  #Convolutional layer
    layers.MaxPooling2D((2,2)), #Pooling layer

    #Flat neurons
    layers.Flatten(),
    #Applies the rectified linear unit activation functio
    layers.Dense(64, activation="relu"),
    #Converts a vector of values to a probability distribution
    layers.Dense(n_classes, activation='softmax')
])
```

Il modello è stato compilato con una funzione di loss appartenente a Keras di nome “SparseCategoricalCrossentropy” e con metrica l’accuratezza, successivamente, è stato poi sottoposto a 200 epoche di training, processando ogni volta 228 batch da 32 immagini ed impiegando circa 30 ore per giungere a termine.

```
#CNN ARCHITECTURE COMPILE WITH A LOOS FUNCTION AND AN OPTIMIZER
#(Optimizer= Adam; Loos function= SparseCategoricalCrossentropy)
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

```
#CNN RUN WITH TRAIN DATASET
history = model.fit(
    train_ds,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    verbose=1,
    validation_data=val_ds
)
```

2.5 Risultati

Una volta finita la fase di training, si è voluto analizzare l'accuratezza del modello appena addestrato e lo si è fatto attraverso la funzione "evaluate" di tensorflow, che ha restituito ottimi risultati, i seguenti:

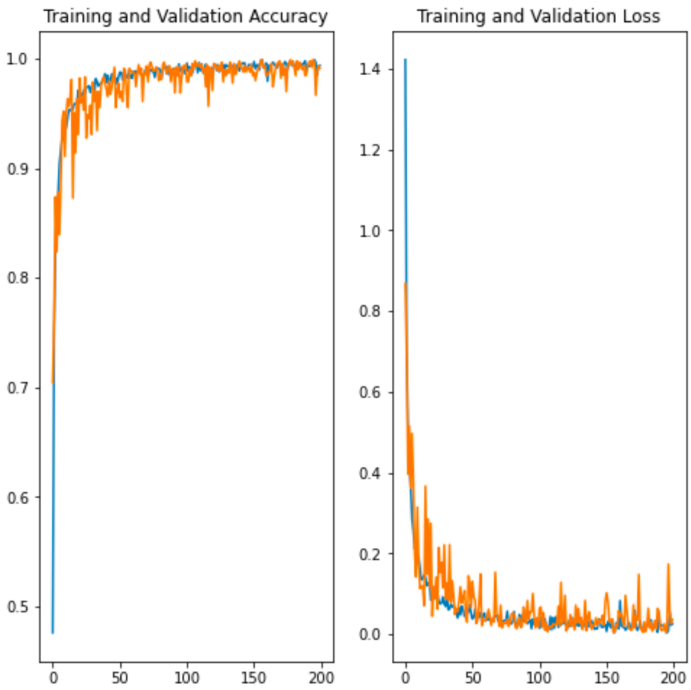
```
    #EVALUATE TRAINED CNN WITH TEST DATASET
    #[loss, accuracy]
    scores = model.evaluate(test_ds)
    scores
```

```
30/30 [==============================] - 43s 581ms/step - loss: 0.0080 - accuracy: 0.9969

[0.007953745312988758, 0.996874988079071]
```

Inoltre, per verificare lo storico dei valori di accuracy e di loss, si sono stampati due grafici che ne rappresentano il valore ad ogni epoca di addestramento, facendone percepire il netto miglioramento dei risultati con l'avanzare dell'allenamento. Visto l'appiattimento della curva, ormai definitivo, si è deciso di terminare l'addestramento a 200 epoche.

In figura, si possono notare le tracce dei valori di training, in colore azzurro, e quelle dei valori di validation, in color arancio:



2.6 Il Modello TensorflowLite

Come descritto nella sezione iniziale, giunti a questo punto, è stato utile pensare alla seconda parte del progetto, ovvero, alla realizzazione di una applicazione Android su cui importare il modello appena addestrato. L'obiettivo era quindi quello di trasformarlo in un formato TensorflowLite, apposito per l'esecuzione su dispositivi a microprocessore, e per farlo, si sono utilizzate le seguenti istruzioni di Tensorflow:

```
tf_lite_converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model=tf_lite_converter.convert()
```

Si è ottenuto inoltre, tramite le seguenti chiamate, la dimensione dei dati che il modello richiede in input e rilascia in output, in quanto una volta giunti nelle classi Java di Android Studio, il modello tflite non sarà più ispezionabile.

```
interpreter = tf.lite.Interpreter(model_path = TF_LITE_MODEL_FILE_NAME)
input_details = interpreter.get_input_details()
output_details= interpreter.get_output_details()
print("Input Shape:", input_details[0]['shape'])
print("Input Type:", input_details[0]['dtype'])

print("Output Shape:", output_details[0]['shape'])
print("Output Type:", output_details[0]['dtype'])
```

```
Input Shape: [  1 256 256   3]
Input Type: <class 'numpy.float32'>
Output Shape: [ 1 10]
Output Type: <class 'numpy.float32'>
```

3 Sviluppo dell'applicazione Android

L'applicazione Android sviluppata ha un comportamento semplice; dopo l'accensione, l'utente può decidere se acquisire un'immagine tramite la fotocamera del telefono, per poi ottenere il nome della pianta corrispondente alla foglia fotografata, oppure, visualizzare l'elenco di quelle classificabili dal modello addestrato.

3.1 Ambiente di sviluppo e configurazione del progetto

Questa seconda parte del progetto è stato sviluppata attraverso l'utilizzo di Android Studio, l'IDE apposito per questo tipo di applicazioni, e al suo interno sono state scritte in Java, le classi contenenti la logica delle Activity, mentre in XML, le loro interfacce grafiche.

Oltre a questi file, ne sono stati prodotti altri due di rilevante importanza, l'AndroidManifest.xml ed il build.gradle. Il primo è un documento in cui le Activity sono definite e coordinate tra di loro, mentre il secondo, serve a registrare le configurazioni principali per la compilazione.

Di seguito sono riportate quelle utilizzate nel nostro caso:

```
defaultConfig {
    applicationId "com.example.plantdetector"
    minSdk 26
    targetSdk 31
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}
```

```
dependencies {
    implementation 'androidx.appcompat:appcompat:1.4.0'
    implementation 'com.google.android.material:material:1.4.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.2'
    implementation 'org.tensorflow:tensorflow-lite-support:0.3.0'
    implementation 'org.tensorflow:tensorflow-lite-metadata:0.3.0'
    implementation 'org.tensorflow:tensorflow-lite-task-vision:0.3.0'
    implementation 'org.tensorflow:tensorflow-lite-task-text:0.3.0'
    implementation 'org.tensorflow:tensorflow-lite-task-audio:0.3.0'
}
```

3.2 Welcome Activity

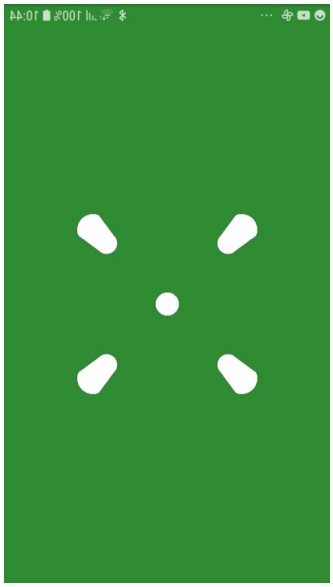
Questa Activity ha un comportamento ed un'interfaccia grafica semplicissima, in quanto il suo fine unico è quello di apparire all'avvio dell'applicazione, aspettare un secondo e poi lasciare spazio alla MainActivity in cui saranno presenti le attività principali dell'applicazione. L'attesa del secondo iniziale è realizzata attraverso la definizione di un nuovo Handler Looper

all'interno della classe Java apposita che, una volta aspettato il tempo impostato, esegue la startActivity() che passerà il controllo all'Activity successiva.

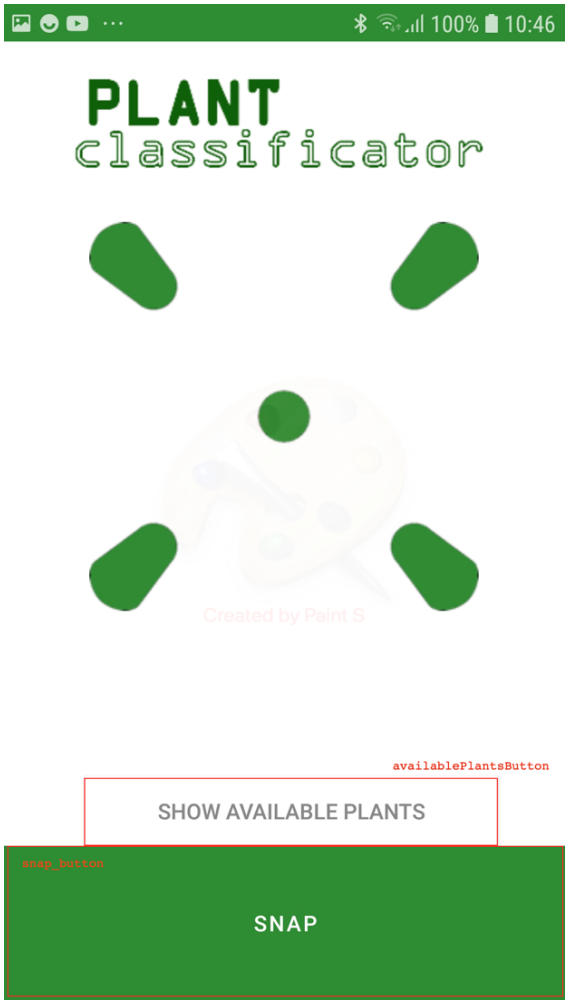
```java
public class WelcomeActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_welcome);

        //Wait 1 seconds
        Intent i = new Intent( packageContext: this, MainActivity.class);
        final Handler handler = new Handler(Looper.getMainLooper());
        handler.postDelayed(new Runnable() {
            @Override
            public void run() {
                //Start MainActivity
                startActivity(i);
            }
        }, delayMillis: 1000);
    }
}
```



3.3 Main Activity



Questa Activity rappresenta il comportamento centrale dell'applicazione.

Al primo accesso, dopo l'installazione dell'app, verrà mostrata a video una permission che richiede all'utente il consenso per accedere alla fotocamera, successivamente l'esecuzione dell'activity procede fornendo le due opzioni sotto descritte:

- Attraverso il Button "Snap" è possibile accedere alla fotocamera
- Attraverso il Button "Show available plants" vengono mostrate le piante che il classifcatore può riconoscere

Nel secondo caso il sistema sarà reindirizzato alla "AvailablePlantsActivity" che verrà mostrata successivamente, mentre nel primo, l'utente potrà accedere alla fotocamera per scattare la fotografia alla foglia desiderata.

In entrambi i casi il comportamento è dato da una funzione di nome OnClickListener() che una volta rilevato il click sul bottone ne specifica le azioni da svolgere. Per il bottone "Show available plants" viene semplicemente attivata l'activity corrispondente, mentre nel caso del bottone "Snap", la logica è più complessa e verrà illustrata nel prossimo paragrafo.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ActivityCompat.requestPermissions( activity: this,new String[]{Manifest.permission.CAMERA},  requestCode: 0);

    button = (Button) findViewById(R.id.snap_button);
    button.setOnClickListener(MainActivity.this);

    plantsList =(Button) findViewById(R.id.availablePlantsButton);
    plantsList.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v){
            startActivity(new Intent( packageContext: MainActivity.this, AvailablePlantsActivity.class));
        }
    });
```

3.3.1 Inferenza sul modello TensorflowLite

All'interno del metodo onClick() corrispondente al bottone "Snap" sono presenti le parti di codice corrispondenti alle seguenti azioni:

- Salvataggio dell'immagine acquisita dalla fotocamera in formato Bitmap.
- Processamento dell'immagine appena salvata attraverso l'utilizzo di un ImageProcessor, con il quale essa viene portata alla dimensione desiderata.
- Preparazione dei dati di input e predisposizione del vettore di output per il modello TensorflowLite.

```
// Create Buffer di input
TensorBuffer inputFeature0 = TensorBuffer.createFixedSize(new int[]{1, 256, 256, 3}, DataType.FLOAT32);
inputFeature0.loadBuffer(tensorImage.getBuffer());

// Create Buffer di output
float[][] result = new float[1][10];
```

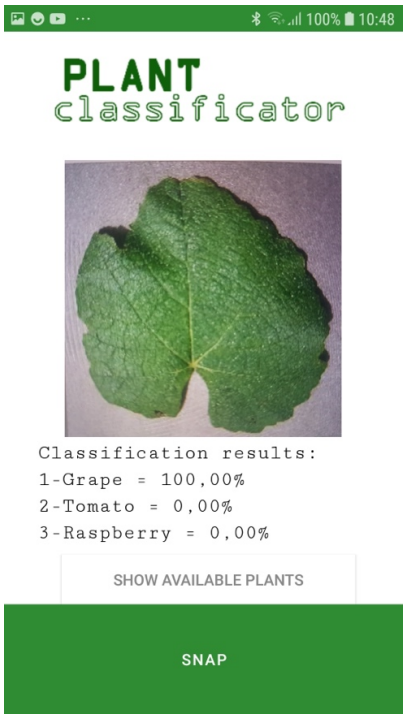
- Caricamento del modello TensorflowLite ed inferenza su di esso con il dato di input appena preparato.

```
MappedByteBuffer tfliteModel
        = FileUtil.loadMappedFile( context: this,
         filePath: "tf_lite_model.tflite");
InterpreterApi tflite = new InterpreterFactory().create(
        tfliteModel, new InterpreterApi.Options());

// Run inference
if (null != tflite) {
    tflite.run(inputFeature0.getBuffer(), result);
}
```

- Abbinamento delle probabilità in uscita dal modello con le dieci label possibili, caricate attraverso un file di testo di nome labels.txt presente nella cartella assets del progetto.

- Setting della leafImageView con l'immagine appena acquisita e dell'outputTextView con i tre migliori risultati restituiti dal modello.



La figura a sinistra è il risultato del percorso appena appena descritto, in cui l'utente ha fotografato una foglia di vite e l'applicazione ha restituito a video i tre migliori risultati prodotti dal modello TensorflowLite caricato.

3.4 Available Plants Activity

Infine, a destra, viene raffigurata l'interfaccia grafica dell'activity a cui viene passato il controllo dell'applicazione nel caso in cui l'utente clicchi il bottone "Show available plants" all'interno della Main Activity. Essa semplicemente mostra l'elenco delle piante che il classificatore è in grado di riconoscere, questa lista rappresenta puntualmente il file labels.txt presente all'interno del progetto.