Intelligent Systems

Gatto Marco
Laino Lorenzo

28 January 2021

Index

Introduction ... 3
Theoretical Part ... 4
- Supervised Learning ... 4
- Deep Learning ... 4
- Convolutional Neural Network (CNN) ... 5
- Tensorflow ... 6
- Keras ... 6
- OpenCV ... 6
- Matplotlib ... 6
- Numpy ... 7
- SciKit-Learn ... 7

Experimental Part ... 8
- Process the data ... 8
- EfficientNet-B0 ... 10
  - Build the model ... 10
  - Compile the model ... 11
  - Train the model ... 12
  - Evaluate the model ... 14
  - Make predictions ... 14
  - Conclusion ... 17
- EfficientNet-B2 ... 20
  - Build and compile the model ... 20
  - Model training, evaluation and predictions ... 21
  - Conclusion ... 21
- EfficientNet-B4 ... 25
  - Conclusion ... 25
- Convolutional Neural Network from Scratch ... 28
  - Build the model ... 28
  - Compile the model ... 29
  - Train the model ... 29
  - Evaluate the model ... 31
  - Save the model ... 31
  - Make predictions ... 31
  - Conclusions ... 32

Introduction

The scope of this project concerns the study and the practical application of some machine learning, and in particular deep learning, techniques.

The project mainly focuses on a dataset called "rock_paper_scissors", available on: https://www.tensorflow.org/datasets/catalog/rock_paper_scissors. The dataset contains 2892 images of hands playing the rock, paper and scissor game.

The experiment is developed using the tools and resources offered by the Python Environment, and in particular: Tensorflow, Keras, OpenCV and more.

The experiment is designed to investigate the behaviours and the capabilities of different Neural Networks under different experimental conditions. In particular, the first part of the experiment uses some pre-trained Convolutional Neural Networks (CNNs) in order to classify the dataset images. The second part, instead, shows the steps of the creation from scratch of a CNN. The obtained results are compared at the end of each part.

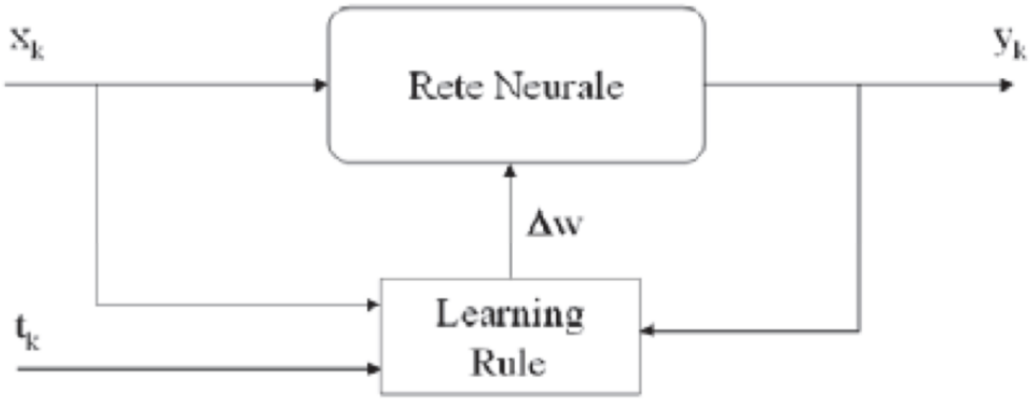
The report is divided into two parts:

1. The first one introduces the main theoretical concepts that has been applied.
2. The second one describes the steps applied during the experimental part and the obtained results.

Theoretical Part

Supervised Learning

Supervised learning tries to model the relationship between measured features of data and some label associated with the data. Once this model is determined, it can be used to apply labels to new, unknown data. This is further subdivided into classification tasks and regression tasks: in classification, the labels are discrete categories, while in regression, the labels are continuous quantities. The experiment in this project concerns a classification problem.



(Xk, tk) training sample with input-output pairs

Deep Learning

Deep learning means using a neural network with several layers of nodes between input and output. The series of features between input and output do feature identification.

A deep neural network consists of a hierarchy of layers, whereby each layer transforms the input data into a more abstract representation (e.g., edge → nose → face). The output layer combines those features to make predictions.

Backpropagation is the widely used learning algorithm for training ordinary neural networks. It aims to identify an optimal set of weights able to minimize the error when processing training data.

In order to solve the problem related to the hidden layers' node weights, the Backpropagation algorithm analyzes the error in the final layer and then looks back at how

it is distributed in the previous layers. So, the information flow goes forward and then the computed error statement is back-propagated updating weights.

In particular, the main steps followed by the algorithms are:

- The process starts with an arbitrary, but not all equal, set of weights throughout the network.
- The application of the generalized delta rule at any iterative step involves three basic phases:
    1. A training vector is presented to the network and is allowed to propagate through the layers to compute the outputs for each node.
    2. The outputs of the nodes in the output layer are then computed against their desired responses to generate the error terms.
    3. The appropriate error signal is passed to each node and the corresponding weight changes are made according to a backward pass through the network.

In a successful training session, the network error decreases with the number of iteration and the procedure converges to a stable set of weights that exhibits only small fluctuations with additional training.

Epoch is an important concept while dealing with the training of a neural network. It is the iterative processing of the training set. Usually, the learning process involves several epochs and the termination condition is formulated in terms of the minimum error accepted or in terms of the number of epochs.

Convolutional Neural Network (CNN)

Convolutional Neural Networks are very similar to ordinary Neural Networks:

- They are made up of neurons that have learnable weights and biases.
- Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity.

The difference between a CNN and an ordinary Neural Network (such as MLP) is about the connections between the layers. In CNN not all the layers are fully connected, while in ordinary Neural Network all the layers are fully connected. This difference makes the CNN the best alternative to work with images.

CNNs have a topology that organizes 3D volumes of neurons to be oriented to process images. Neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully connected manner.

Tensorflow

TensorFlow is an open-source software library released in 2015 by Google to make it easier for developers to design, build and train deep learning models.

On a high level, TensorFlow is a Python library that allows users to express arbitrary computation as a graph of data flows. Nodes in this graph represent mathematical operations, whereas edges represent data that is communicated from one node to another. Data in TensorFlow is represented as tensors, which are multidimensional arrays (representing vectors with a 1D tensor, matrices with a 2D tensor, etc.).

Although this framework for thinking about computation is valuable in many different fields, TensorFlow is primarily used for deep learning in practice and research.

Keras

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to do good research. Keras has the following key features:

- Allows the same code to run on CPU or on GPU, seamlessly.
- User-friendly API which makes it easy to quickly prototype deep learning models.
- Built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- Supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, etc. This means that Keras is appropriate for building essentially any deep learning model, from a memory network to a neural Turing machine.

OpenCV

OpenCV-Python is a library of Python bindings designed to solve computer vision problems. All the OpenCV array structures are converted to and from Numpy arrays. This makes it easier to integrate with other libraries that use Numpy such as SciPy and Matplotlib.

Matplotlib

Matplotlib is a multiplatform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. One of the Matplotlib's most important

features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish.

Numpy

NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in list type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size.

SciKit-Learn

Scikit-learn is an open-source machine learning library that supports supervised and unsupervised learning in Python. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

Experimental Part

"Rock_paper_scissors" dataset contains 2892 colour images, each with dimensions *300x300px*. The content of the images within the dataset is sampled from 3 classes:

- Rock
- Paper
- Scissors

Process the data

The deep learning Keras library provides direct access to the "Rock_paper_scissors" (Keras dataset) with relative ease, through its dataset module.

```
train_ds, test_ds = tfds.load('rock_paper_scissors', split=['train', 'test'], as_supervised=True)
```

Training Dataset (train_ds): this is the group of our dataset used to train the neural network directly. Training data refers to the dataset partition exposed to the neural network during the training.

Test Dataset (test_ds): this partition of the dataset evaluates the performance of our network after the completion of the training phase.

Since the experiment is supervised, both Training and Test Datasets are composed by images and labels. For this reason, it is necessary to define a method to split the images

from the labels.

```
def create_img_label_set(data_set, dimension):
    image_set = []
    label_set = []
    for img in data_set:
        image, label = img[0], img[1]
        res_img = cv2.resize(image, dsize=(dimension, dimension), interpolation=cv2.INTER_CUBIC)
        image_set.append(res_img)
        label_set.append(label)
    return np.asarray(image_set), np.asarray(label_set)
```

Given a dataset and a dimension value as input, the method generates two arrays containing the images and the labels respectively. Furthermore, the method uses the dimension value and the cv2 library to resize the images as *dimension x dimension*.

```
img_train, label_train = create_img_label_set(train_ds, 50)
img_test, label_test = create_img_label_set(test_ds, 50)
```

In order to obtain a faster and easier model, the images have been resized to *50x50px*.

At the end of the data processing phase, an overview of the results obtained from the previous steps is shown.

```
Labels number: 3
Total images: 2892
Train images shape: (2520, 50, 50, 3)
Train labels shape: (2520,)
Test images shape: (372, 50, 50, 3)
Test labels shape: (372,)
```

EfficientNet-B0

EfficientNet-B0 is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. It is the baseline model from which all the other EfficientNet models are scaled up. It is characterized from more than 5 million parameters and a top-1 accuracy value around 77%.

Build the model

Since EfficientNet-B0 is a well-known pre-trained model, it can be imported from the Keras library. Furthermore, it is needed to create an additional layer that is used as the new input layer of the model. The input size, corresponding to the dimension value of each image, is provided to this layer.

```
inputs = tf.keras.layers.Input(shape=(50, 50, 3))
efn_b0_model = EfficientNetB0(include_top=False, input_tensor=inputs, weights='imagenet')
```

The weights of the pre-trained model need to be frozen. In this way, it is possible to reuse all the discriminant capability learnt by the model during the previous training phases.

```
# Freeze the pretrained weights
efn_b0_model.trainable = False
```

Finally, it is needed to re-build the top layers of the model. These are the only layers that will be trained in the following phases.

```
# Rebuild top
x = tf.keras.layers.GlobalAveragePooling2D(name='avg_pool')(efn_b0_model.output)
x = tf.keras.layers.BatchNormalization()(x)

top_dropout_rate = 0.5
x = tf.keras.layers.Dropout(top_dropout_rate, name='top_dropout')(x)
outputs = tf.keras.layers.Dense(num_classes, activation='softmax', name='pred')(x)

efn_b0_model = tf.keras.Model(inputs, outputs, name='EfficientNet')
```

In particular, the introduced layers are:

- GlobalAveragePooling2D: it is the Keras representation of an average pooling layer. Average pooling is a variant of sub-sampling where the mean of the pixels that fall within the receptive field of a unit within a sub-sampling layer is taken as the output. The pool size is still set to the size of the input layer.
- BatchNormalization: it is a technique that mitigates the effect of unstable gradients within a neural network through the introduction of an additional layer that performs operations on the inputs from the previous layer. The

operations standardize and normalize the input values, after that the input values are transformed through scaling and shifting operations.

- Dropout: it is a technique that works by randomly reducing the number of interconnecting neurons within a neural network. At every training step, each neuron has a chance of being left out, or rather, dropped out of the collated contributions from connected neurons. In the above piece of code, the probability of being left out equals 0.5.
- Dense: it is the Keras representation of a Fully-Connected layer. It has an embedded number of arbitrary units/neurons within. Each neuron is a perceptron. This layer represents the output layer: indeed, its number of units equals the number of different classes of the dataset. Softmax is a type of activation function that is utilized to derive the probability distribution of a set of numbers within an input vector. The output of a softmax activation function is a vector in which its set of values represents the probability of an occurrence of a class or event. The values within the vector all add up to 1.

At the end, the final model is built-up.

Compile the model

Before the training phase, the model is compiled. In particular, it is needed to specify some parameters, such as the optimizer, the loss function and the metrics.

```
# Compile
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-2)
efn_b0_model.compile(
    optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy']
)
```

The optimizer is the algorithm used to change the attributes of the neural network such as weights and learning rate to reduce the losses, and it is used to solve optimization problems by minimizing the function.

The learning rate is a factor value that determines the level of updates that are made to the values of the weights of the network. It is a type of hyperparameter.

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. In particular, this method is computationally efficient, invariant to diagonal rescaling of gradients, has little memory requirement and is well suited for problems that are large in terms of data and parameters.

The loss function is a method that quantifies 'how well' a machine learning model performs; the quantification is a cost based on a set of inputs, which are referred to as parameter values. The parameter values are used to estimate a prediction, and the 'loss' is the difference between the predictions and the actual values.

The metrics are used to measure the performances of the model. The accuracy is the ratio of number of correct predictions to the total number of input samples.

```
def build_efn_b0_model(num_classes):
    inputs = tf.keras.layers.Input(shape=(50, 50, 3))
    efn_b0_model = EfficientNetB0(include_top=False, input_tensor=inputs, weights='imagenet')

    # Freeze the pretrained weights
    efn_b0_model.trainable = False

    # Rebuild top
    x = tf.keras.layers.GlobalAveragePooling2D(name='avg_pool')(efn_b0_model.output)
    x = tf.keras.layers.BatchNormalization()(x)

    top_dropout_rate = 0.5
    x = tf.keras.layers.Dropout(top_dropout_rate, name='top_dropout')(x)
    outputs = tf.keras.layers.Dense(num_classes, activation='softmax', name='pred')(x)

    efn_b0_model = tf.keras.Model(inputs, outputs, name='EfficientNet')

    # Compile
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-2)
    efn_b0_model.compile(
        optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy']
    )
    return efn_b0_model
```

The image above shows the complete method used in this experiment in order to build-up and compile the model.

Train the model

In order to train the model, the number of epochs and the validation set should be defined.

```
epochs = 5
hist = model.fit(img_train, label_train, epochs=epochs, validation_data=(img_test, label_test),
                 verbose=2, batch_size=105)
```

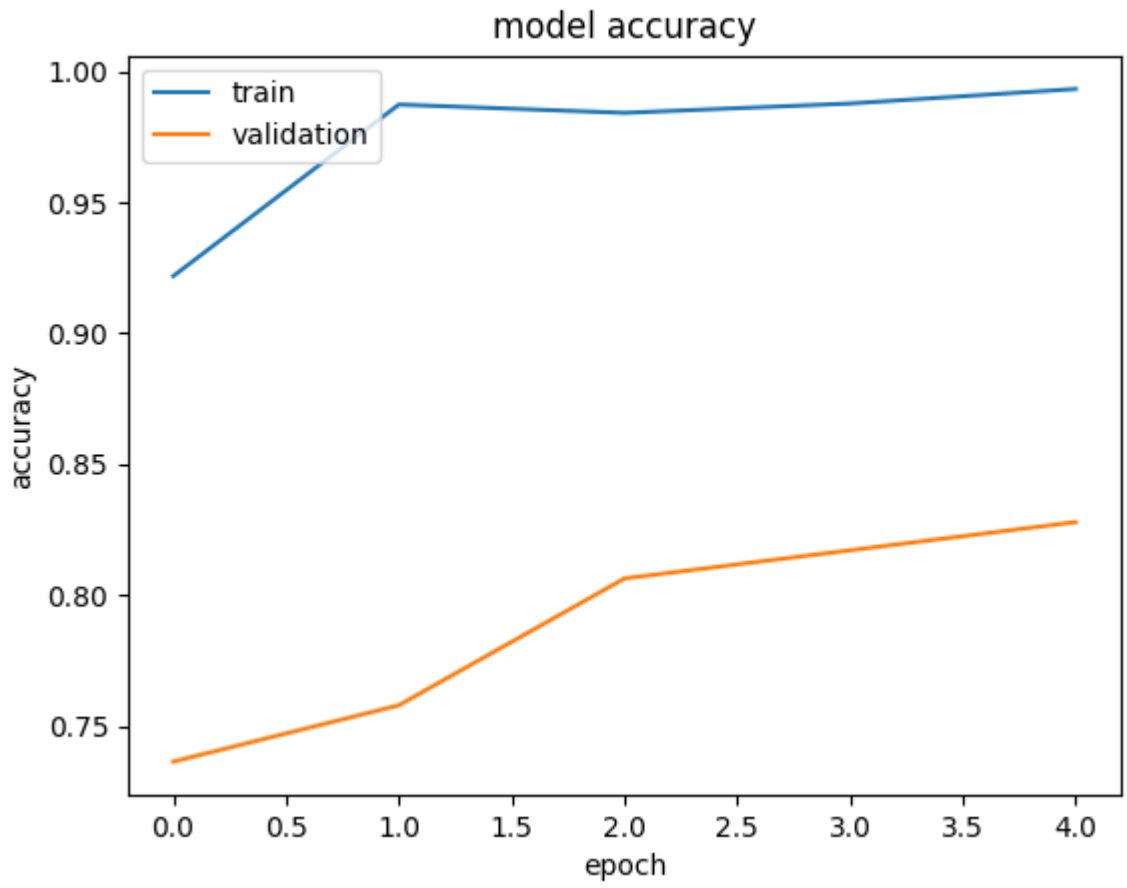
The model is trained along 5 epochs and the test set is also used as validation test. It is also specified the batch_size parameter, that represents the number of samples per gradient update. In particular, using a batch_size value equal to 105, the training set is perfectly divided into 24 batches.

An example of training execution is shown in the following image.

```
24/24 - 10s - loss: 0.2364 - accuracy: 0.9218 - val_loss: 0.5641 - val_accuracy: 0.7366
Epoch 2/5
24/24 - 5s - loss: 0.0518 - accuracy: 0.9873 - val_loss: 0.4996 - val_accuracy: 0.7581
Epoch 3/5
24/24 - 5s - loss: 0.0537 - accuracy: 0.9841 - val_loss: 0.4197 - val_accuracy: 0.8065
Epoch 4/5
24/24 - 5s - loss: 0.0528 - accuracy: 0.9877 - val_loss: 0.5242 - val_accuracy: 0.8172
Epoch 5/5
24/24 - 6s - loss: 0.0385 - accuracy: 0.9933 - val_loss: 0.5055 - val_accuracy: 0.8280
12/12 [==============================] - 1s 65ms/step - loss: 0.5055 - accuracy: 0.8280
```

It is also possible to show the training and validation accuracies with a graphical representation.

```
plot_hist(hist)
```



The plot graphically represents the model fitting execution of the previous example. In particular, it can be seen that both the training and the validation accuracies grow in a constant way.

A more detailed analysis of the results will be performed in the conclusion paragraph through the comparison of the results got from different executions.

Evaluate the model

This phase is needed to evaluate the performances of the model. In particular, the metrics used to measure the performances are the ones specified when the model is compiled. In this case, the reference metric is the accuracy, that is defined as the ratio of the number of correct predictions to the total number of input samples.

```
# Evaluate the model
accuracy = model.evaluate(img_test, label_test)
print(model.metrics_names)
print('Accuracy: %.2f%%' % (accuracy[1] * 100))
```

```
['loss', 'accuracy']
Accuracy: 82.80%
```

Since the validation set used during the training phase is the same as the test set, the value of the final accuracy equals the final value in the previous plot.

Make predictions

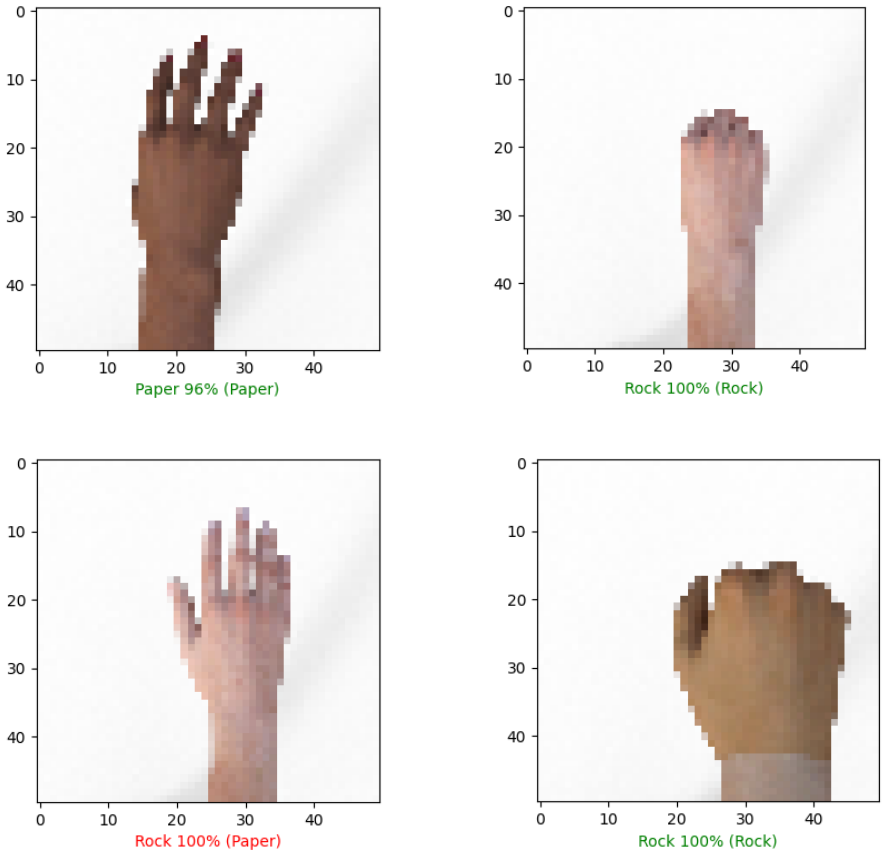
The final phase concerns the usage of the model to classify some images that have not been used during the training phase. In particular, the model is asked to predict the label of all the images contained in the test set.

```
# Make predictions
predictions = model.predict(img_test)
```

Furthermore, it is needed to create a method that shows the results of the prediction on some randomly-chosen images.

```
def plot_image(i):
    img = img_test[i]
    predicted = predictions[i]
    label = label_test[i]

    predicted_label = np.argmax(predicted)
    color = 'red' if predicted_label != label else 'green'

    plt.imshow(img, cmap=plt.cm.binary)
    plt.xlabel('{} {:2.0f}% ({})'.format(label_names[predicted_label], 100 * np.max(predicted), label_names[label]),
               color=color)


for i in range(0, 4):
    rand = np.random.randint(0, len(img_test))
    print(rand)
    plt.figure(figsize=(7, 4))
    plot_image(rand)
    plt.show()
```

This method simply choses 4 random indexes, takes from the test set the images corresponding to the chosen indexes and attaches to the images both the corresponding label and predicted values.



Since the output layer of the model uses the 'softmax' activation function, a probability value is associated with each input image and it represents the probability that the specific image is classified from the model with that specific label.

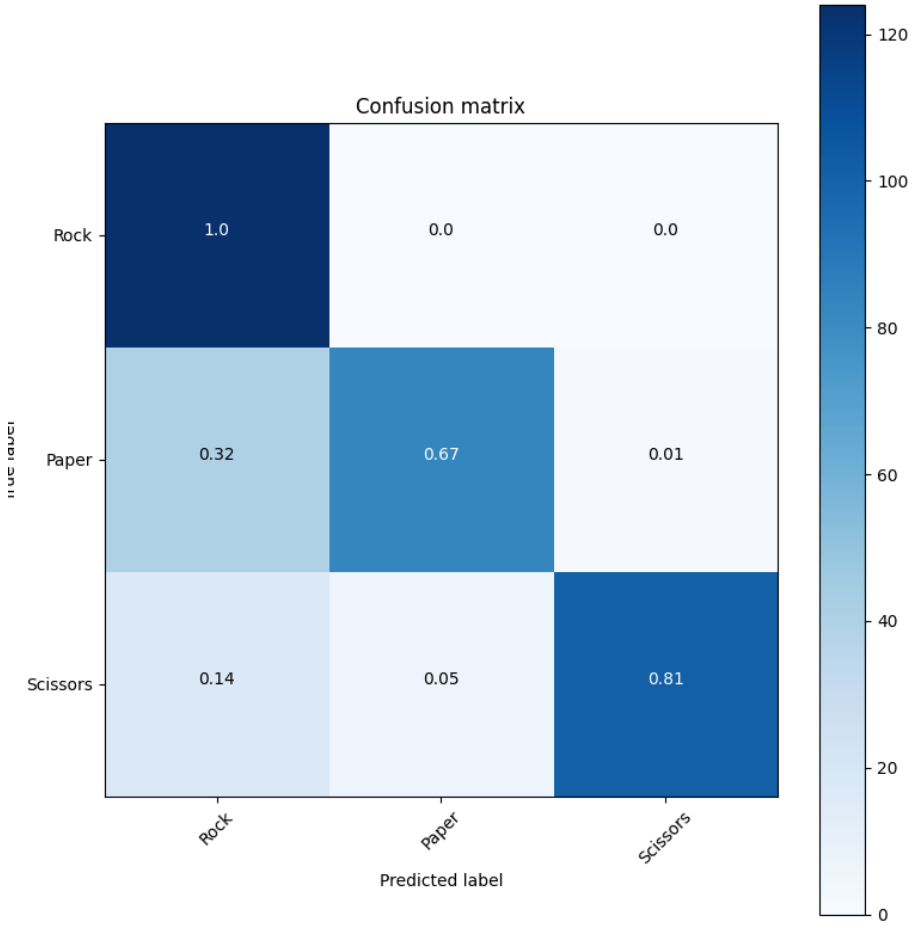
It is also computed and shown the confusion matrix concerning the labels predicted from the model. For this reason, it is used the utility provided from the library SciKit-Learn that automatically computes the confusion matrix given both the actual and the predicted labels.

```
# Use the model to predict the values from the test_images.
test_pred = np.argmax(predictions, axis=1)

# Calculate the confusion matrix using sklearn.metrics
cm = confusion_matrix(label_test, test_pred)

plot_confusion_matrix(cm)
```

Furthermore, it is needed to define a method that, given the confusion matrix, normalizes its values and then shows it as a plot.

```
def plot_confusion_matrix(cm):
    plt.figure(figsize=(8, 8))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title("Confusion matrix")
    plt.colorbar()
    tick_marks = np.arange(len(label_names))
    plt.xticks(tick_marks, label_names, rotation=45)
    plt.yticks(tick_marks, label_names)

    # Normalize the confusion matrix.
    conf_matrix = np.around(cm.astype('float') / cm.sum(axis=1)[:, np.newaxis], decimals=2)

    # Use white text if squares are dark; otherwise black.
    threshold = conf_matrix.max() / 2.

    for i, j in itertools.product(range(conf_matrix.shape[0]), range(conf_matrix.shape[1])):
        color = "white" if conf_matrix[i, j] > threshold else "black"
        plt.text(j, i, conf_matrix[i, j], horizontalalignment="center", color=color)

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

An example of the results obtained through this code is the following:



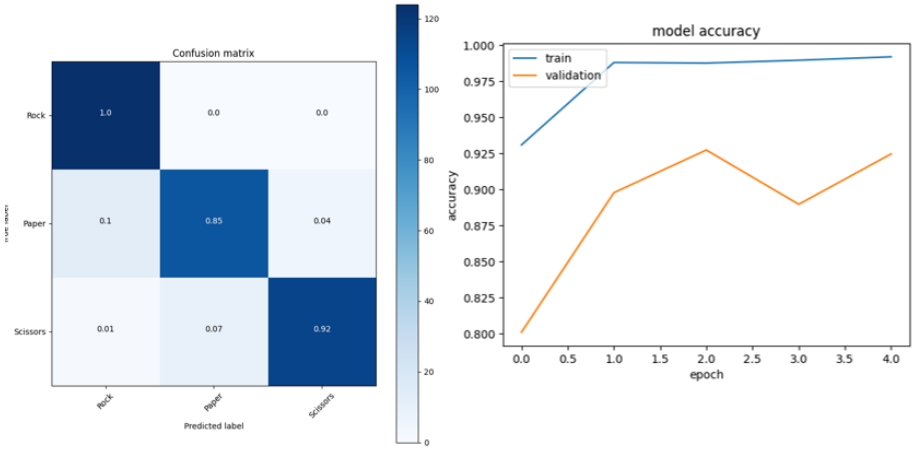
Even in this case, the image represents the confusion matrix of the model used as example in the previous steps.

Conclusion

In order to analyze the performances of EfficientNet-B0 on the "rock_paper_scissors" dataset, it is needed to evaluate the accuracy's behavior of the model in different executions. Some examples are shown in the following paragraphs.

Execution Example 1

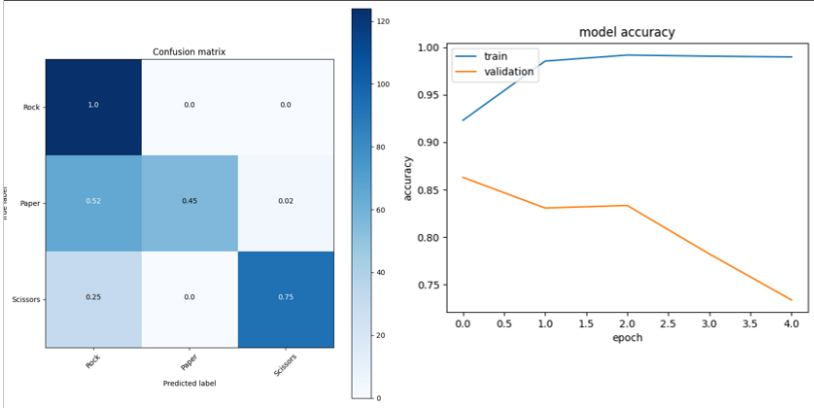
```
Epoch 1/5
24/24 - 10s - loss: 0.2009 - accuracy: 0.9310 - val_loss: 0.3760 - val_accuracy: 0.8011
Epoch 2/5
24/24 - 5s - loss: 0.0376 - accuracy: 0.9881 - val_loss: 0.3300 - val_accuracy: 0.8978
Epoch 3/5
24/24 - 5s - loss: 0.0453 - accuracy: 0.9877 - val_loss: 0.1931 - val_accuracy: 0.9274
Epoch 4/5
24/24 - 5s - loss: 0.0378 - accuracy: 0.9897 - val_loss: 0.2940 - val_accuracy: 0.8898
Epoch 5/5
24/24 - 5s - loss: 0.0478 - accuracy: 0.9921 - val_loss: 0.2054 - val_accuracy: 0.9247
12/12 [==============================] - 1s 63ms/step - loss: 0.2054 - accuracy: 0.9247
['loss', 'accuracy']
Accuracy: 92.47%
```



In this example, both the training and validation accuracies reach very good values (values > 90%). As it can be seen from the confusion matrix, the model is able to perform a very good discrimination. In particular, it never fails to recognize hands playing rock, it recognizes hands playing scissors with an accuracy value equal to 92% and hands playing scissors with a margin of error equal to 15%. The barplot graphically shows the accuracy trend of the model over the different epochs.

Execution Example 2

```
Epoch 1/5
24/24 - 10s - loss: 0.2353 - accuracy: 0.9230 - val_loss: 0.3592 - val_accuracy: 0.8629
Epoch 2/5
24/24 - 5s - loss: 0.0516 - accuracy: 0.9853 - val_loss: 0.4153 - val_accuracy: 0.8306
Epoch 3/5
24/24 - 5s - loss: 0.0363 - accuracy: 0.9917 - val_loss: 0.4476 - val_accuracy: 0.8333
Epoch 4/5
24/24 - 5s - loss: 0.0294 - accuracy: 0.9905 - val_loss: 0.6482 - val_accuracy: 0.7823
Epoch 5/5
24/24 - 5s - loss: 0.0232 - accuracy: 0.9897 - val_loss: 1.0241 - val_accuracy: 0.7339
12/12 [==============================] - 1s 64ms/step - loss: 1.0241 - accuracy: 0.7339
['loss', 'accuracy']
Accuracy: 73.39%
```

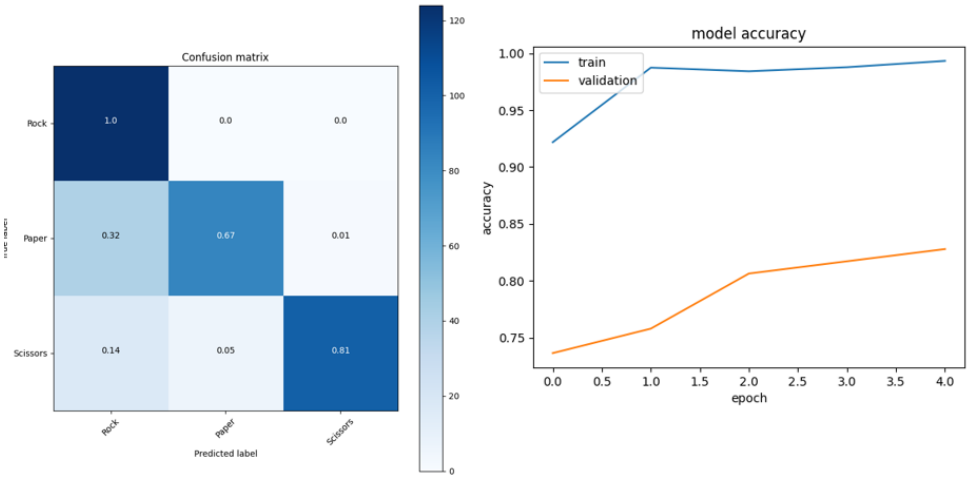


In this example, the training accuracy reaches a very good value, but the validation accuracy shows that the model bumps into some difficulties when new data are presented to it. In particular, it can also be seen that the trend of the validation accuracy over the different epochs is decreasing. Furthermore, the confusion matrix shows that the model is only able to recognize hands playing rock with a very good accuracy, it is quite able to recognize hands playing scissors (accuracy = 75%), but it shows terrible results concerning the recognition of hands playing paper.

This is a typical behavior of an overfit model, that shows high performances in training but low performances in generalization.

Execution Example 3

```
Epoch 1/5
24/24 - 10s - loss: 0.2364 - accuracy: 0.9218 - val_loss: 0.5641 - val_accuracy: 0.7366
Epoch 2/5
24/24 - 5s - loss: 0.0518 - accuracy: 0.9873 - val_loss: 0.4996 - val_accuracy: 0.7581
Epoch 3/5
24/24 - 5s - loss: 0.0537 - accuracy: 0.9841 - val_loss: 0.4197 - val_accuracy: 0.8065
Epoch 4/5
24/24 - 5s - loss: 0.0528 - accuracy: 0.9877 - val_loss: 0.5242 - val_accuracy: 0.8172
Epoch 5/5
24/24 - 6s - loss: 0.0385 - accuracy: 0.9933 - val_loss: 0.5055 - val_accuracy: 0.8280
12/12 [==============================] - 1s 65ms/step - loss: 0.5055 - accuracy: 0.8280
['loss', 'accuracy']
Accuracy: 82.80%
```



In this example, both the training and the validation accuracies show an increasing trend over the epochs. In particular, the training accuracy reaches very good results, while the accuracy value stands around the 82%. According to the confusion matrix, the model never fails to recognize hands playing rock, it reaches acceptable values of accuracy in the recognition of hands playing scissors, but is not very able to recognize hands playing paper (margin of error = 33%).

Final thoughts on the experiments' results

The model has shown the best results in the first execution example. The results can be considered acceptable in the third execution example. The results obtained in the second execution example are not acceptable.

The pre-trained EfficientNet-B0 model seems not to be very adequate to recognize the images presented as input. In particular, it quickly overfits and the differences between the various executions performed are remarkable, even with a low number of epochs.

So, it can be useful to choose a more powerful EfficientNet model, apply the same procedure to the new pre-trained model and analyze the obtained results.

EfficientNet-B2

EfficientNet-B2 is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The model is characterized from more than 9 million parameters and a top-1 accuracy value around the 80%.

Build and compile the model

Since EfficientNet-B2 is a well-known pre-trained model, it can be imported from the Keras library. Even in this case, it is needed to create a new input layer, containing the information regarding the input size.

```
    inputs = tf.keras.layers.Input(shape=(50, 50, 3))
    efn_b2_model = EfficientNetB2(include_top=False, input_tensor=inputs, weights='imagenet')
```

The other steps concerning the model building and compilation are very similar to the ones described for the experiment with EfficientNet-B2.

```
def build_efn_b2_model(num_classes):
    inputs = tf.keras.layers.Input(shape=(50, 50, 3))
    efn_b2_model = EfficientNetB2(include_top=False, input_tensor=inputs, weights='imagenet')

    # Freeze the pretrained weights
    efn_b2_model.trainable = False

    # Rebuild top
    x = tf.keras.layers.GlobalAveragePooling2D(name='avg_pool')(efn_b2_model.output)
    x = tf.keras.layers.BatchNormalization()(x)

    top_dropout_rate = 0.5
    x = tf.keras.layers.Dropout(top_dropout_rate, name='top_dropout')(x)
    outputs = tf.keras.layers.Dense(num_classes, activation='softmax', name='pred')(x)

    efn_b2_model = tf.keras.Model(inputs, outputs, name='EfficientNet')

    # Compile
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-2)
    efn_b2_model.compile(
        optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy']
    )
    return efn_b2_model
```

The image above shows the complete method used in this experiment in order to build-up and compile the model.

Model training, evaluation and predictions

Even for all these phases, the steps performed are very similar to the ones described for the experiment with EfficientNet-B0. So, only the main steps and codes are briefly reported.

In order to train the model, the number of epochs and the validation set should be defined. The model is trained along 5 epochs and the test set is also used as validation test.

```
epochs = 5
hist = model.fit(img_train, label_train, epochs=epochs, validation_data=(img_test, label_test),
                 verbose=2, batch_size=105)
```

Then, the performances of the model need to be evaluated. In particular, the metrics used to measure the performances are the ones specified when the model is compiled. In our case, the reference metric is the accuracy, that is defined as the ratio of the number of correct predictions to the total number of input samples.

```
# Evaluate the model
accuracy = model.evaluate(img_test, label_test)
print(model.metrics_names)
print('Accuracy: %.2f%%' % (accuracy[1] * 100))
```

The final phase concerns the usage of the model to classify some images that have not been used during the training phase. In particular, the model is asked to predict the label of all the images contained in the test set.

```
# Make predictions
predictions = model.predict(img_test)
```

It is also computed and shown the confusion matrix concerning the labels predicted from the model.

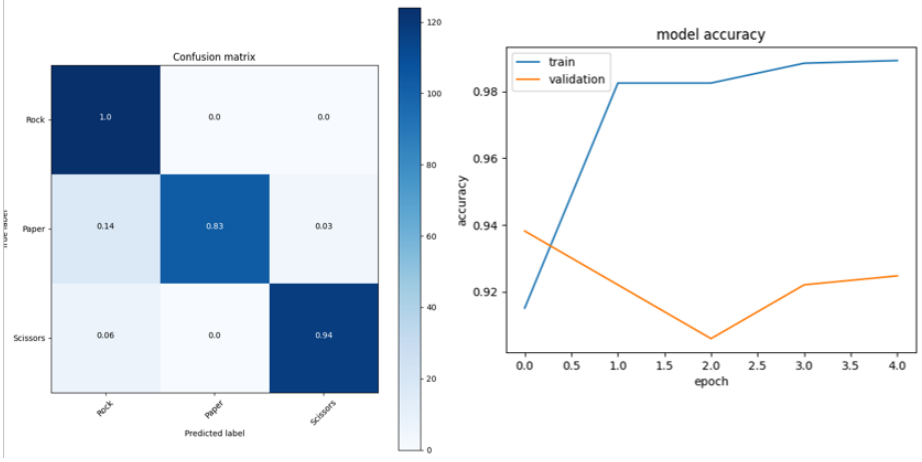
```
# Use the model to predict the values from the test_images.
test_pred = np.argmax(predictions, axis=1)

# Calculate the confusion matrix using sklearn.metrics
cm = confusion_matrix(label_test, test_pred)

plot_confusion_matrix(cm)
```

Conclusion

In order to analyze the performances of EfficientNet-B2 on the "rock_paper_scissors" dataset, it is needed to evaluate the accuracy's behavior of the model in different executions. Some examples are shown in the following paragraphs.

Execution Example 1

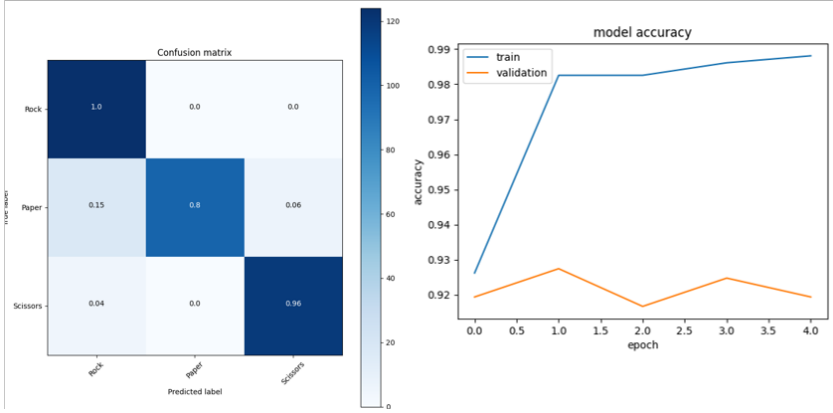
```
Epoch 1/5
24/24 - 15s - loss: 0.3058 - accuracy: 0.9151 - val_loss: 0.1935 - val_accuracy: 0.9382
Epoch 2/5
24/24 - 8s - loss: 0.0936 - accuracy: 0.9825 - val_loss: 0.3391 - val_accuracy: 0.9220
Epoch 3/5
24/24 - 8s - loss: 0.0694 - accuracy: 0.9825 - val_loss: 0.4141 - val_accuracy: 0.9059
Epoch 4/5
24/24 - 8s - loss: 0.0665 - accuracy: 0.9885 - val_loss: 0.4948 - val_accuracy: 0.9220
Epoch 5/5
24/24 - 8s - loss: 0.0417 - accuracy: 0.9893 - val_loss: 0.3490 - val_accuracy: 0.9247
12/12 [==============================] - 1s 104ms/step - loss: 0.3490 - accuracy: 0.9247
['loss', 'accuracy']
Accuracy: 92.47%
```



In this example, both the training and validation accuracies reach very good values (values > 90%). As it can be seen from the confusion matrix, the model is able to perform a very good discrimination. In particular, it never fails to recognize hands playing rock, it recognizes hands playing scissors with an accuracy value equal to 94% and hands playing scissors with a margin of error equal to 17%. The barplot graphically shows the accuracy trend of the model over the different epochs.

Execution Example 2

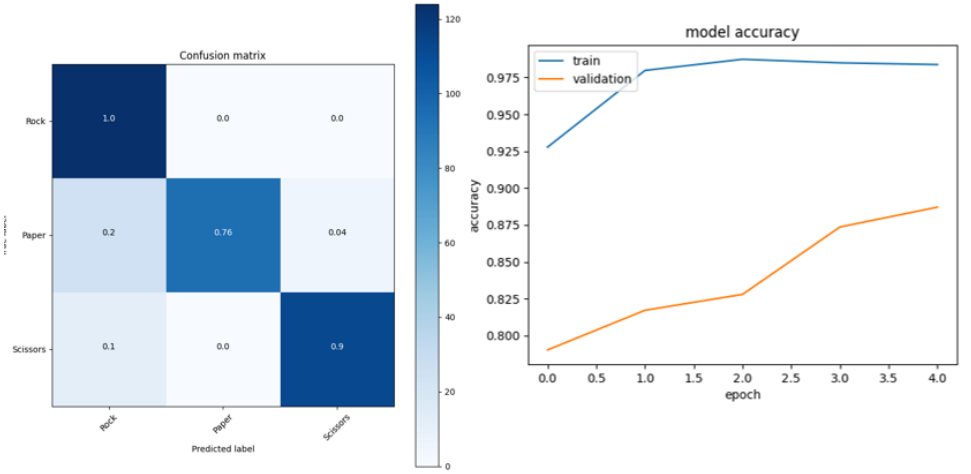
```
Epoch 1/5
24/24 - 15s - loss: 0.2337 - accuracy: 0.9262 - val_loss: 0.2510 - val_accuracy: 0.9194
Epoch 2/5
24/24 - 8s - loss: 0.0843 - accuracy: 0.9825 - val_loss: 0.2201 - val_accuracy: 0.9274
Epoch 3/5
24/24 - 8s - loss: 0.0761 - accuracy: 0.9825 - val_loss: 0.3065 - val_accuracy: 0.9167
Epoch 4/5
24/24 - 8s - loss: 0.0575 - accuracy: 0.9861 - val_loss: 0.4190 - val_accuracy: 0.9247
Epoch 5/5
24/24 - 8s - loss: 0.0475 - accuracy: 0.9881 - val_loss: 0.4709 - val_accuracy: 0.9194
12/12 [==============================] - 1s 109ms/step - loss: 0.4709 - accuracy: 0.9194
['loss', 'accuracy']
Accuracy: 91.94%
```



Even in this example, both the training and validation accuracies reach values that are over the 90%. Also, the confusion matrix shows that the discrimination power of the model is still very good. In particular, with respect to the previous experiment, the model shows the same accuracy in the recognition of hands playing rock and very slight differences in the recognition of hands playing both scissors and paper. The accuracy trend of the model over the different epochs is shown to be stable around the value 92%.

Execution Example 3

```
Epoch 1/5
24/24 - 15s - loss: 0.2568 - accuracy: 0.9278 - val_loss: 0.6778 - val_accuracy: 0.7903
Epoch 2/5
24/24 - 8s - loss: 0.0910 - accuracy: 0.9798 - val_loss: 0.5842 - val_accuracy: 0.8172
Epoch 3/5
24/24 - 8s - loss: 0.0523 - accuracy: 0.9873 - val_loss: 0.6706 - val_accuracy: 0.8280
Epoch 4/5
24/24 - 8s - loss: 0.0630 - accuracy: 0.9849 - val_loss: 0.6573 - val_accuracy: 0.8737
Epoch 5/5
24/24 - 8s - loss: 0.0671 - accuracy: 0.9837 - val_loss: 0.6591 - val_accuracy: 0.8871
12/12 [==============================] - 1s 106ms/step - loss: 0.6591 - accuracy: 0.8871
['loss', 'accuracy']
Accuracy: 88.71%
```



In this example, the training accuracy reaches an almost stable value of 98%, while the validation accuracy shows an increasing trend over the epochs and its final value stands around the 88%. According to the confusion matrix, the model never fails to recognize hands playing rock, it reaches very good values of accuracy in the recognition of hands playing scissors and acceptable values in the recognition of hands playing paper (accuracy = 76%).<

Final thoughts on the experiments' results

The model has shown the best results in both the first and the second execution examples. The results can be considered very good in the third execution example.

The three examples performed can prove that the pre-trained model EfficientNet-B2 reaches a discriminant power that is considerably stronger in average with respect to EfficientNet-B0. In particular, among various execution examples, it has been reported as third experiment example the one in which the worst results have been reached.

According to the results obtained with both the previous pre-trained models, it can be interesting to investigate the results that can be reached with a more powerful EfficientNet model.

EfficientNet-B4

EfficientNet-B4 is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The model is characterized from more than 19 million parameters and a top-1 accuracy value around the 83%.

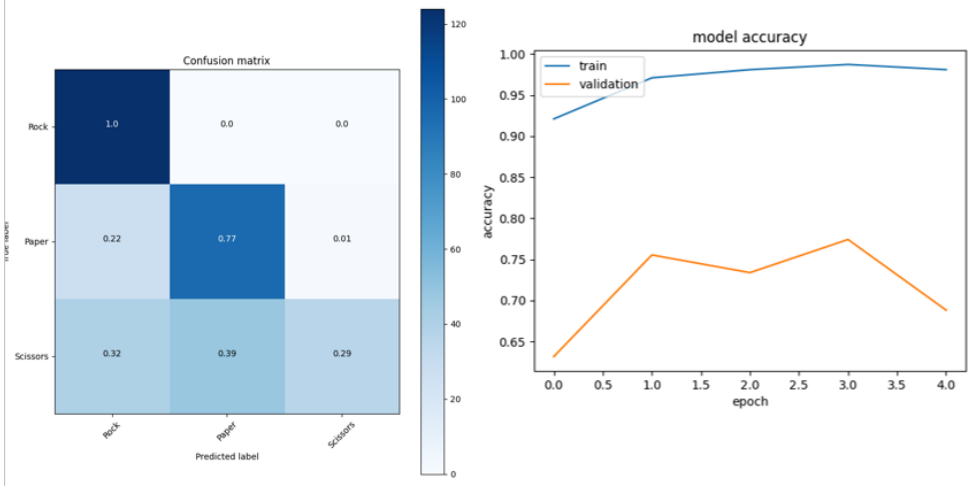
Since it has been followed the same procedure also for this experiment, only the experiment conclusions will be reported. Furthermore, this choice also allows to avoid the report to be too long.

Conclusion

In order to analyze the performances of EfficientNet-B4 on the "rock_paper_scissors" dataset, it is needed to evaluate the accuracy's behavior of the model in different executions. Some examples are shown in the following paragraphs.

Execution Example 1

```
Epoch 1/5
24/24 - 22s - loss: 0.2715 - accuracy: 0.9210 - val_loss: 0.9832 - val_accuracy: 0.6317
Epoch 2/5
24/24 - 13s - loss: 0.1344 - accuracy: 0.9710 - val_loss: 0.5301 - val_accuracy: 0.7554
Epoch 3/5
24/24 - 13s - loss: 0.0690 - accuracy: 0.9810 - val_loss: 0.4878 - val_accuracy: 0.7339
Epoch 4/5
24/24 - 13s - loss: 0.0532 - accuracy: 0.9873 - val_loss: 0.4434 - val_accuracy: 0.7742
Epoch 5/5
24/24 - 13s - loss: 0.1101 - accuracy: 0.9810 - val_loss: 0.7151 - val_accuracy: 0.6882
12/12 [==============================] - 2s 177ms/step - loss: 0.7151 - accuracy: 0.6882
['loss', 'accuracy']
Accuracy: 68.82%
```

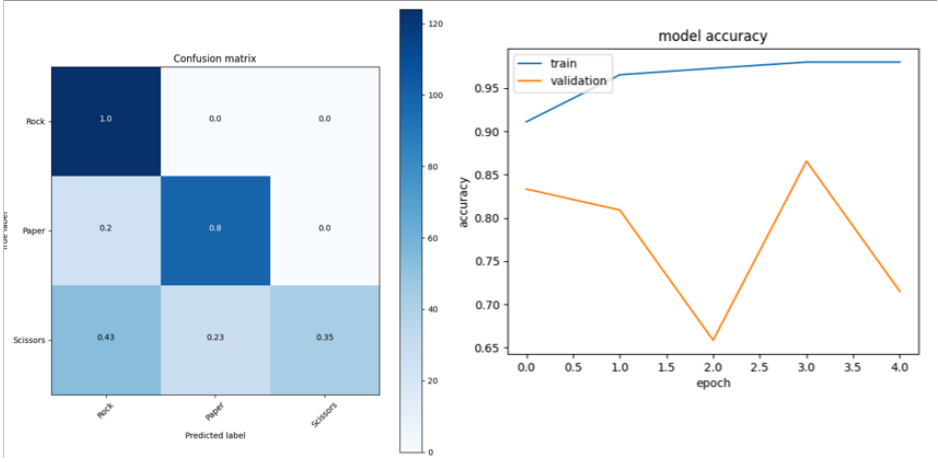


In this example, the training accuracy reaches a very good value, but the validation accuracy shows that the model bumps into some difficulties when new data are presented to it. In particular, the trend of the accuracy over the epochs does not show a continuous and incremental learning growth. Furthermore, the confusion matrix shows that the model is only able to recognize hands playing rock with a very good accuracy, it is quite able to recognize hands playing paper (accuracy = 77%), but it shows terrible results concerning

the recognition of hands playing scissors. In particular, the probability of the model to truly recognize hands playing scissors is even lower than 1/3.

Execution Example 2

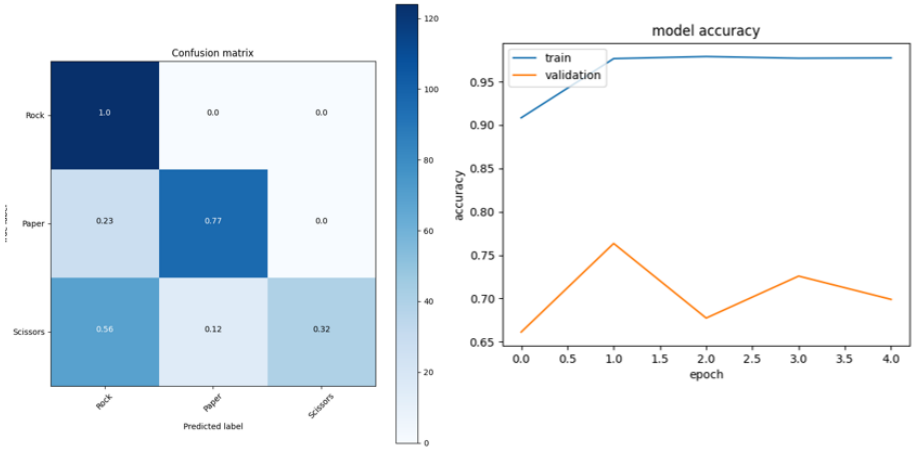
```
Epoch 1/5
24/24 - 22s - loss: 0.3579 - accuracy: 0.9111 - val_loss: 0.3470 - val_accuracy: 0.8333
Epoch 2/5
24/24 - 13s - loss: 0.1724 - accuracy: 0.9655 - val_loss: 0.4187 - val_accuracy: 0.8091
Epoch 3/5
24/24 - 13s - loss: 0.1050 - accuracy: 0.9730 - val_loss: 0.9329 - val_accuracy: 0.6586
Epoch 4/5
24/24 - 13s - loss: 0.0930 - accuracy: 0.9802 - val_loss: 0.4082 - val_accuracy: 0.8656
Epoch 5/5
24/24 - 13s - loss: 0.1035 - accuracy: 0.9802 - val_loss: 0.6734 - val_accuracy: 0.7151
12/12 [==============================] - 2s 180ms/step - loss: 0.6734 - accuracy: 0.7151
['loss', 'accuracy']
Accuracy: 71.51%
```



Even in this example, the training accuracy reaches a very good value, but the validation accuracy shows that the model bumps into some difficulties when new data are presented to it. In particular, it can also be seen that the trend of the validation accuracy over the different epochs continuously changes. According to the confusion matrix, the model never fails in the recognition of hands playing rock, it reaches good values in the recognition of hands playing paper, but most of the times it fails in the recognition of hands playing scissors (accuracy = 35%).

Execution Example 3

```
Epoch 1/5
24/24 - 22s - loss: 0.3347 - accuracy: 0.9083 - val_loss: 0.8198 - val_accuracy: 0.6613
Epoch 2/5
24/24 - 13s - loss: 0.1115 - accuracy: 0.9766 - val_loss: 0.5445 - val_accuracy: 0.7634
Epoch 3/5
24/24 - 12s - loss: 0.0968 - accuracy: 0.9790 - val_loss: 0.7681 - val_accuracy: 0.6774
Epoch 4/5
24/24 - 12s - loss: 0.1061 - accuracy: 0.9770 - val_loss: 0.6603 - val_accuracy: 0.7258
Epoch 5/5
24/24 - 12s - loss: 0.1036 - accuracy: 0.9774 - val_loss: 0.9464 - val_accuracy: 0.6989
12/12 [==============================] - 2s 180ms/step - loss: 0.9464 - accuracy: 0.6989
['loss', 'accuracy']
Accuracy: 69.89%
```



Even in this example, the training accuracy reaches a very good value, but the validation accuracy never reaches very high values. In particular, the barplot shows that the trend of the validation accuracy over the epochs continuously changes and never goes over the 80%. The confusion matrix shows a model behavior that is similar to the one presented in the previous experiment examples. In particular, even in this case, the model's ability in the recognition of hands playing scissors is lower than 1/3.

Final thoughts on the experiments' results

The model has shown the best results in the second execution example. However, the results can be considered not acceptable in all of the execution examples presented.

The pre-trained EfficientNet-B4 model seems not to be very adequate to recognize the images presented as input. In particular, it quickly overfits and the accuracy values reached in all the execution examples are never very high. In particular, with respect to the other pre-trained models considered in the previous experiments, EfficientNet-B4 has shown a strong difficulty in the recognition of hands playing scissors. Furthermore, the model shows the typical overfitting behavior in all the execution examples reported.

So, the use of a more powerful EfficientNet model has given worse results. In particular, the EfficientNet-B4 model has given accuracy values that are 20% lower in average with respect to the EfficientNet-B2 model. These results can be probably blamed to an excessive model complexity.

Convolutional Neural Network from Scratch

For sake of completeness, and in order to increase the difficulty coefficient of the experiment, it has been chosen to create a convolutional neural network from scratch, train the neural network using the "rock_paper_scissors" dataset, evaluate the performances that it is able to reach and finally compare these results with the ones obtained in the previous experiment examples using a pre-trained model.

Build the model

The library Keras allows the creation of neural networks through the usage of tensors. A tensor is a multi-dimensional array that only contains values with a uniform type. All the tensors are unchanging: the value of a tensor can never be modified after its creation. Furthermore, all the tensors are characterized by a rank, that represents the number of axes that are needed in order to represent the values contained in the tensor.

In order to create a convolutional neural network, the Sequential class is used. Sequential groups a linear stack of layers into a model and provides training and inference features on this model. All the other layers only need to be sequentially added to the sequential model.

```
# Build the model
model = keras.Sequential()
model.add(keras.layers.Conv2D(50, (3, 3), input_shape=(50, 50, 3)))
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(3, 3)))

model.add(keras.layers.Conv2D(25, (3, 3)))
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(3, 3)))

model.add(keras.layers.Conv2D(50, (3, 3)))
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))

model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(50))
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(len(label_names), activation='softmax'))
```

The introduced layers are:

- Conv2D: this layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. A convolution is a mathematical term that describes a dot product multiplication between two sets of elements. Within deep learning, the convolution operation acts on the filters/kernels and image data array within the convolutional layer. This layer requires both the dimensionalities of the output space and of the kernel

window. When it is used to represent the input layer, it is also needed to specify the dimensionality of the images presented to the model.

- Activation: this layer applies an activation function to an output. An activation function is a mathematical operation that transforms the result or signals of neurons into a normalized output. The purpose of an activation function as a component of a neural network is to introduce non-linearity within the network. The inclusion of an activation function enables the neural network to have greater representational power and solve complex functions. ReLU is an activation function that transforms the values according to the formula y=max(0, x). In particular, it clamps down any negative values from the neuron to 0, and positive values remain unchanged.
- MaxPooling2D: it is the Keras representation of a max pooling layer. It downsamples the input representation by taking the maximum value over the window defined by pool_size for each dimension along the features' axis.
- Flatten: this layer takes an input shape and flattens the input image data into a one-dimensional array.
- The layers Dense and Dropout are the same explained in the "Build the model" section of the EfficientNet-B0 experiment.

The output layer is represented with a Dense layer in which the number of units equals the number of different labels of the dataset. Furthermore, in order to characterize the outputs of the model also with a probability value, a softmax activation function is used in this layer.

Compile the model

Before the training phase, the model is compiled. In particular, it is needed to specify some parameters, such as the loss function, the optimizer and the metrics.

```
# Compile the model
model.compile(loss='sparse_categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

The loss function used is the sparse categorical crossentropy. It computes the crossentropy loss between the labels and predictions and is used when there are two or more label classes.

RMSprop is an optimization algorithm that maintains a moving (discounted) average of the square of gradients and then divides the gradient by the root of this average.

As usual, the accuracy is the metric used to measure the performances of the model.

Train the model

In order to train the model, the number of epochs and the validation set should be defined.

```
# Train the model
hist = model.fit(img_train, label_train, validation_data=(img_test, label_test), epochs=10, batch_size=105)
```

In particular, the model is trained along 10 epochs and the test set is also used as validation set. Even in this case, it is used a batch_size value equal to 105, so that the training set is perfectly divided into 24 batches containing 105 images each.

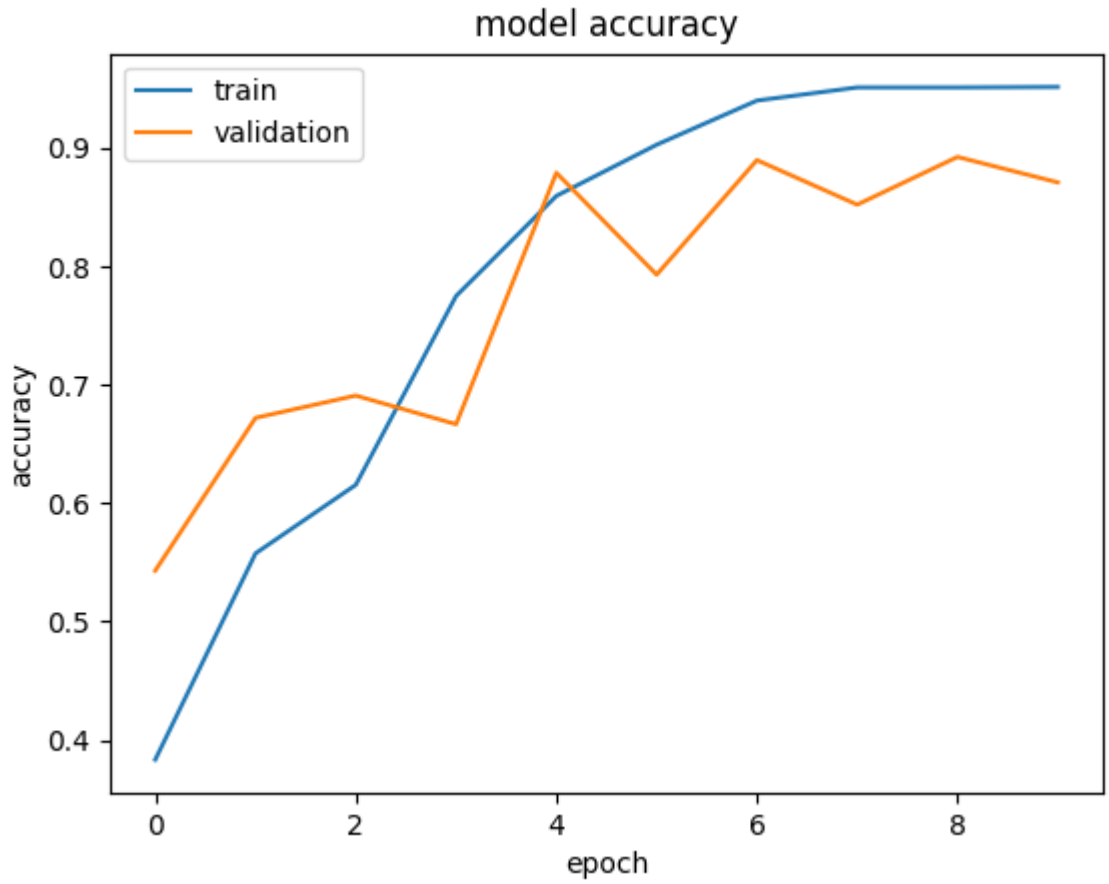
The training execution is shown in the following image.

```
Epoch 1/10
24/24 [==============================] - 3s 98ms/step - loss: 19.1105 - accuracy: 0.3507 - val_loss: 0.9275 - val_accuracy: 0.5430
Epoch 2/10
24/24 [==============================] - 2s 82ms/step - loss: 1.0099 - accuracy: 0.5141 - val_loss: 0.7343 - val_accuracy: 0.6720
Epoch 3/10
24/24 [==============================] - 2s 82ms/step - loss: 0.8893 - accuracy: 0.6028 - val_loss: 0.7017 - val_accuracy: 0.6909
Epoch 4/10
24/24 [==============================] - 2s 82ms/step - loss: 0.5271 - accuracy: 0.7634 - val_loss: 0.8031 - val_accuracy: 0.6667
Epoch 5/10
24/24 [==============================] - 2s 82ms/step - loss: 0.3337 - accuracy: 0.8556 - val_loss: 0.5313 - val_accuracy: 0.8790
Epoch 6/10
24/24 [==============================] - 2s 83ms/step - loss: 0.2389 - accuracy: 0.9072 - val_loss: 0.6089 - val_accuracy: 0.7930
Epoch 7/10
24/24 [==============================] - 2s 82ms/step - loss: 0.2908 - accuracy: 0.9063 - val_loss: 0.5743 - val_accuracy: 0.8898
Epoch 8/10
24/24 [==============================] - 2s 82ms/step - loss: 0.1356 - accuracy: 0.9555 - val_loss: 0.7122 - val_accuracy: 0.8522
Epoch 9/10
24/24 [==============================] - 2s 82ms/step - loss: 0.1791 - accuracy: 0.9372 - val_loss: 0.5186 - val_accuracy: 0.8925
Epoch 10/10
24/24 [==============================] - 2s 82ms/step - loss: 0.1542 - accuracy: 0.9538 - val_loss: 0.6324 - val_accuracy: 0.8710
```

It is also possible to show the training and validation accuracies with a graphical representation.

```
plot_hist(hist)
```

The method used to show the plot is the same explained in the 'Train the model' section of the EfficientNet-B0 experiment.



The plot graphically represents the model fitting execution of the previous example. In particular, it can be seen that the training accuracy grows in a constant way, while the validation accuracy is characterized by some trend inversions.

A more detailed analysis of the results will be performed in the conclusion paragraph.

Evaluate the model

This phase is needed to evaluate the performances of the model.

```
# Evaluate the model
accuracy = model.evaluate(img_test, label_test)
print(model.metrics_names)
print('Accuracy: %.2f%%' % (accuracy[1] * 100))
```

```
['loss', 'accuracy']
Accuracy: 87.10%
```

Since the validation set used during the training phase is the same as the test set, the value of the final accuracy equals the final value in the previous plot.

Save the model

Once the model is trained and evaluated, it has been chosen to save it. In this way, it is possible to import the trained model into other projects and reuse it as a pre-trained one. In particular, both the architecture and the weights have been saved into two different files, so that they can be easily moved and reused.

```
# Save the model
json_file = model.to_json()
with open(f'{proj_dir_path}\\trained_models\\rps_10_epochs_cnn_model.json', 'w') as file:
    file.write(json_file)

# Serialize weights to HDF5
model.save_weights(f'{proj_dir_path}\\trained_models\\rps_10_epochs_cnn_model_weights.hdf5')
```

The variable 'proj_dir_path' contains the absolute path of the project directory. So, both the model's architecture and its weights are saved into two different files of a specific directory.

Make predictions

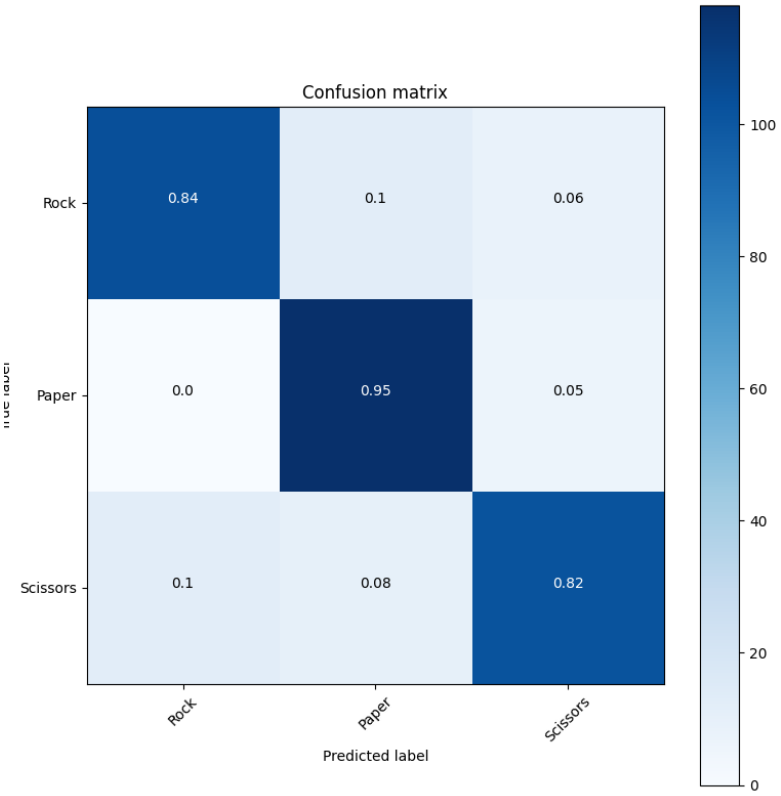
The final phase concerns the usage of the model to classify some images that have not been used during the training phase. In particular, the model is asked to predict the label of all the images contained in the test set.

However, it is also needed a preliminary step, that involves the trained model to be loaded from the file in which it has been previously saved.

```
# Load json and create model
file = open(f'{proj_dir_path}\\trained_models\\rps_10_epochs_cnn_model.json', 'r')
json_model = file.read()
file.close()
loaded_model = keras.models.model_from_json(json_model)

# Load weights
loaded_model.load_weights(f'{proj_dir_path}\\trained_models\\rps_10_epochs_cnn_model_weights.hdf5')

predictions = loaded_model.predict(img_test)
```

It is also computed and shown the confusion matrix concerning the labels predicted from the model.

```
# Use the model to predict the values from the test_images.
test_pred = np.argmax(predictions, axis=1)

# Calculate the confusion matrix using sklearn.metrics
cm = confusion_matrix(label_test, test_pred)

plot_confusion_matrix(cm)
```

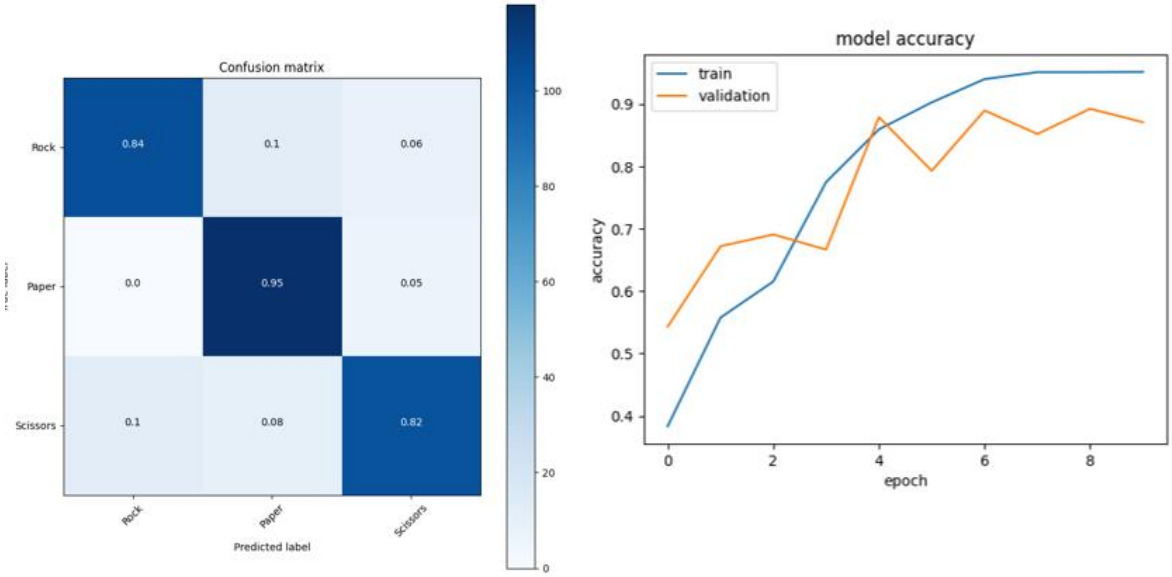


Conclusions

In order to analyze the performances of the experimental convolutional neural network on the "rock_paper_scissors" dataset, it is needed to evaluate the accuracy's behavior of the model in different executions. In particular, two tests have been performed for this purpose: the first one involves the training of the model along 10 epochs, while the second one involves a training phase of 15 epochs.

Test 1

```
Epoch 1/10
24/24 [==============================] - 3s 98ms/step - loss: 19.1105 - accuracy: 0.3507 - val_loss: 0.9275 - val_accuracy: 0.5430
Epoch 2/10
24/24 [==============================] - 2s 82ms/step - loss: 1.0099 - accuracy: 0.5141 - val_loss: 0.7343 - val_accuracy: 0.6720
Epoch 3/10
24/24 [==============================] - 2s 82ms/step - loss: 0.8893 - accuracy: 0.6028 - val_loss: 0.7017 - val_accuracy: 0.6909
Epoch 4/10
24/24 [==============================] - 2s 82ms/step - loss: 0.5271 - accuracy: 0.7634 - val_loss: 0.8031 - val_accuracy: 0.6667
Epoch 5/10
24/24 [==============================] - 2s 82ms/step - loss: 0.3337 - accuracy: 0.8556 - val_loss: 0.5313 - val_accuracy: 0.8790
Epoch 6/10
24/24 [==============================] - 2s 83ms/step - loss: 0.2389 - accuracy: 0.9072 - val_loss: 0.6089 - val_accuracy: 0.7930
Epoch 7/10
24/24 [==============================] - 2s 82ms/step - loss: 0.2908 - accuracy: 0.9063 - val_loss: 0.5743 - val_accuracy: 0.8898
Epoch 8/10
24/24 [==============================] - 2s 82ms/step - loss: 0.1356 - accuracy: 0.9555 - val_loss: 0.7122 - val_accuracy: 0.8522
Epoch 9/10
24/24 [==============================] - 2s 82ms/step - loss: 0.1791 - accuracy: 0.9372 - val_loss: 0.5186 - val_accuracy: 0.8925
Epoch 10/10
24/24 [==============================] - 2s 82ms/step - loss: 0.1542 - accuracy: 0.9538 - val_loss: 0.6324 - val_accuracy: 0.8710
['loss', 'accuracy']
Accuracy: 87.10%
```

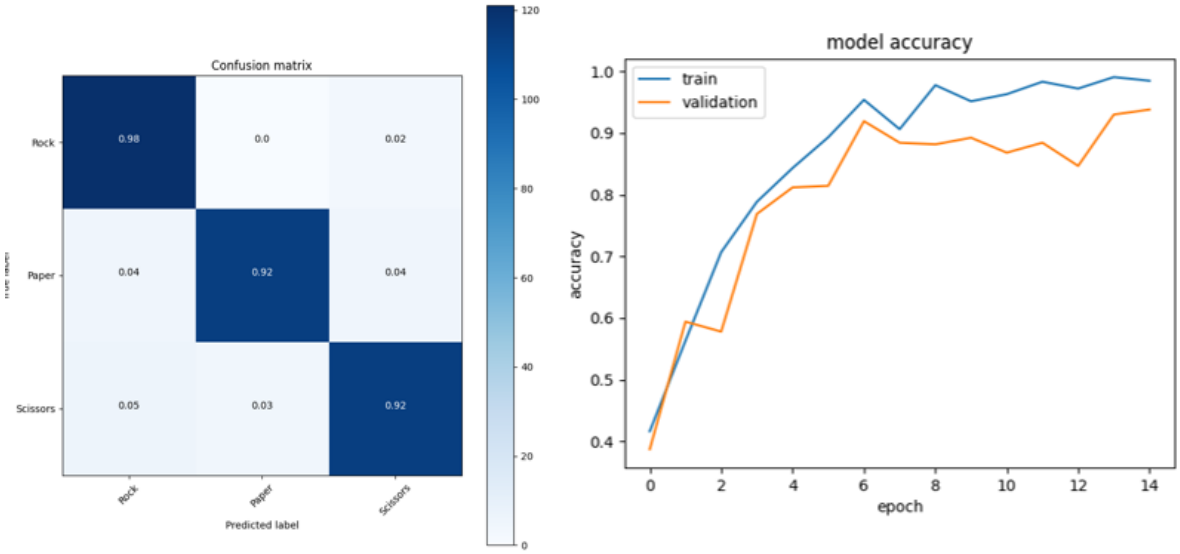


In this test, the model is trained along 10 epochs.

Both the training and validation accuracies reach very good values. As it can be seen from the confusion matrix, the model is able to perform a good discrimination. In particular, the model is able to recognize hands playing paper with a very good accuracy (value = 95%) and to recognize hands playing both rock and scissors with accuracy values that respectively equal 82% and 84%. The barplot graphically shows the accuracy trend of the model over the different epochs. In particular, it highlights a constant growth of the training accuracy and a globally increasing trend for the validation accuracy.

Test 2

```
Epoch 1/15
24/24 [==============================] - 3s 97ms/step - loss: 11.0653 - accuracy: 0.3761 - val_loss: 1.0202 - val_accuracy: 0.3871
Epoch 2/15
24/24 [==============================] - 2s 82ms/step - loss: 1.0753 - accuracy: 0.5231 - val_loss: 0.8162 - val_accuracy: 0.5941
Epoch 3/15
24/24 [==============================] - 2s 82ms/step - loss: 0.7730 - accuracy: 0.6667 - val_loss: 0.8014 - val_accuracy: 0.5780
Epoch 4/15
24/24 [==============================] - 2s 81ms/step - loss: 0.9979 - accuracy: 0.7275 - val_loss: 0.4935 - val_accuracy: 0.7688
Epoch 5/15
24/24 [==============================] - 2s 82ms/step - loss: 0.3926 - accuracy: 0.8460 - val_loss: 0.4343 - val_accuracy: 0.8118
Epoch 6/15
24/24 [==============================] - 2s 81ms/step - loss: 0.2960 - accuracy: 0.9118 - val_loss: 0.3778 - val_accuracy: 0.8145
Epoch 7/15
24/24 [==============================] - 2s 81ms/step - loss: 0.1648 - accuracy: 0.9442 - val_loss: 0.2204 - val_accuracy: 0.9194
Epoch 8/15
24/24 [==============================] - 2s 81ms/step - loss: 0.2679 - accuracy: 0.9261 - val_loss: 0.2603 - val_accuracy: 0.8844
Epoch 9/15
24/24 [==============================] - 2s 81ms/step - loss: 0.0745 - accuracy: 0.9762 - val_loss: 0.4777 - val_accuracy: 0.8817
Epoch 10/15
24/24 [==============================] - 2s 81ms/step - loss: 0.1015 - accuracy: 0.9563 - val_loss: 0.2169 - val_accuracy: 0.8925
Epoch 11/15
24/24 [==============================] - 2s 81ms/step - loss: 0.1238 - accuracy: 0.9615 - val_loss: 0.3942 - val_accuracy: 0.8683
Epoch 12/15
24/24 [==============================] - 2s 81ms/step - loss: 0.0380 - accuracy: 0.9871 - val_loss: 0.4529 - val_accuracy: 0.8844
Epoch 13/15
24/24 [==============================] - 2s 82ms/step - loss: 0.0347 - accuracy: 0.9883 - val_loss: 0.4605 - val_accuracy: 0.8468
Epoch 14/15
24/24 [==============================] - 2s 81ms/step - loss: 0.0348 - accuracy: 0.9891 - val_loss: 0.2384 - val_accuracy: 0.9301
Epoch 15/15
24/24 [==============================] - 2s 82ms/step - loss: 0.0371 - accuracy: 0.9882 - val_loss: 0.1087 - val_accuracy: 0.9382
['loss', 'accuracy']
Accuracy: 93.82%
```



In this test, the model is trained along 15 epochs.

Both the training and validation accuracies reach very good values (accuracy > 90%). As it can be seen from the confusion matrix, the model is able to perform a very good discrimination, with accuracy values that are always greater than 90%. In particular, the best result is reached in the recognition of hands playing rock, where the accuracy value equals 98%. The barplot shows that both the training and validation accuracies do not follow a monotonic trend, even if they reach very good values at the end of the last epoch.

Final thoughts on the experiment results

The results obtained from a convolutional neural network built from scratch are comparable to the best results obtained using a pre-trained model, that is EfficientNet-B2. In particular, since the weights of the CNN are initialized with random values, it requires a higher number of epochs in order to reach the same results of a pre-trained model.

The creation of a CNN from scratch allows to obtain results that are more focused on the classification of the images that are more interesting for the purposes of the

experiment. However, this approach is more time-consuming and it requires more technical skills from the implementational point of view.

On the other hand, the use of a pre-trained model allows to obtain quicker results with the same level of accuracy. Furthermore, they are also equipped with the related documentation and suggestions of use. However, a pre-trained model has to be taken as-is and there is a poor number of possible customizations that can be applied to it.

Given all these considerations and all the experimental attempts performed, probably the best solution in this case is EfficientNet-B2, because it is ready to use and it reaches very good accuracy results.

Bibliography

"Fundamentals of Deep Learning", Nikhil Buduma
"Python Data Science Handbook", Jake VanderPlas
https://www.tensorflow.org/overview
https://keras.io/api/applications/
https://docs.opencv.org/master/d7/da8/tutorial_table_of_content_imgproc.html
https://numpy.org/doc/stable/
https://matplotlib.org/
https://scikit-learn.org/stable/getting_started.html
https://www.tensorflow.org/datasets/catalog/rock_paper_scissors
https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet