Projet : Deep Learning

—

Classification de fruits

—

Par Enzo Di Maria, Yassir El Bsita, Arthur Couturier, José Colin & Rémi Bonrepaux

Département Sciences du Numérique - Deuxième année, Ingénierie du logiciel
2021-2022

Table des matières

Ce rapide rapport sera l'occasion pour nous de vous présenter les grands axes de notre projet : de la constitution de la base de données à l'analyse des résultats en passant par les choix de conception et d'architecture.

Il vient en complément des deux notebooks présents à la racine du répertoire GitHub, mais la trame globale ainsi que les analyses y sont déjà rédigées. C'est la raison pour laquelle ici nous nous attarderons davantage sur les choix de conception ainsi que les problèmes avec lesquels nous avons du composer.

1 Architecture du projet

Le projet a été pensé pour être le plus possible accessible à l'utilisateur. Il fallait allier précision et concision. Conséquence : les deux notebooks se montrent assez épurés en terme de code. Code qui finalement se révèle moins important que les résultats qu'il permet de produire. Les notebooks consistent donc principalement à appeler dans le bon ordre les fonctions écrites dans des fichiers situés dans des dossiers à part.

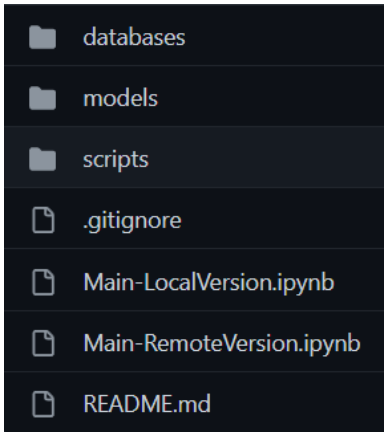

FIGURE 1 – Racine du projet

A la racine du projet, vous retrouverez les notebooks, l'un est conçu pour une utilisation via Google Colab et l'autre pour une utilisation locale. Les diverses fonctions appelées par ceux-ci sont réparties dans les dossiers *models* et *scripts*, tandis que les bases de données sont (logiquement) situées dans le dossier *databases*.

FIGURE 2 – *realistic-database* et *hybrid-database* à gauche / *simple-database* à droite

Les bases de données sont toutes fondées sur le même modèle et le même ratio : 15% des images sont dans *test*, 15% dans *validation*, et le reste (70%) dans *train*. Notez tout de même que *simple-database* se montre un peu plus complexe, puisqu'elle possède trois ensembles de test : une base de tests simples, une base de tests réalistes, et une base de tests pour le fun !

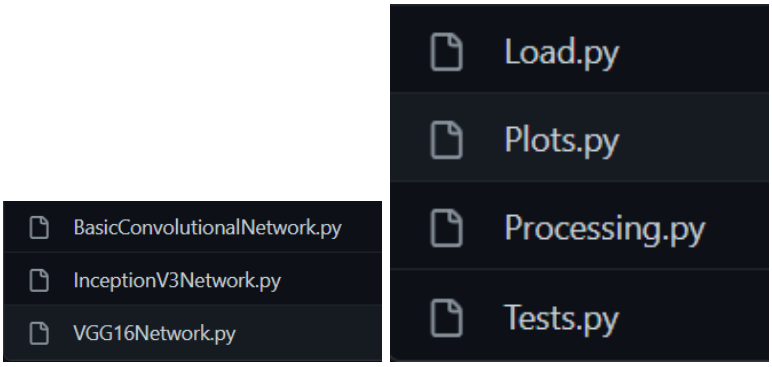

FIGURE 3 – *models* et *scripts*

Le dossier *models* contient les différents modèles que l'on a testé (ils seront détaillés dans une prochaine partie), tandis que *scripts* contient les fonctions destinées à alléger les notebooks : fonctions d'affichage,

de test, de chargement des données, de traitement des images de la base de données.

2 Constitution des bases de données

Cette partie, bien qu'en apparence triviale, fut celle qui a subi le plus de modifications. Notre intention de départ, que nous avons respecté jusqu'au bout, était de classer des images de fruits, de tirer des probabilités à partir de chaque image d'une base de données test. Dans un premier temps, nous avons opter pour une base de données constituée de 19 classes, une classe représentant un fruit. Puis, pour simplifier, nous nous sommes réduits à 10 classes. Comme l'atteste notre précédent rapport, notre intention de départ était de constituer une unique base de données dont les images étaient assez compliqués : un ou plusieurs fruits par image, arrière-plans non uniformes. Toutefois bien vite nous nous sommes heurtés à un mur. La précision, peu importe le modèle, et sans se montrer désastreuse, ne dépassait pourtant pas 65-70%.

Pour pallier à ce problème, nous avons constitué une nouvelle base de données ne comprenant que des images simples : un unique fruit sur fond blanc. Les résultats sont bien meilleurs puisqu'on compte une précision de 95% à 100%. C'est pourtant bien évident. Les images étant particulièrement simplistes et toujours construites selon le même archétype, le modèle ne peut qu'apprendre facilement la forme et la couleur des fruits.

C'est la raison pour laquelle nous avons eu l'idée de constituer une base de données hybride entre la première et la seconde. Ainsi le modèle se trouve en capacité d'apprendre la couleur du fruit et sa forme dans un contexte simple (fond blanc) mais aussi dans un contexte plus compliqué (table, panier). La différence avec la base de données "complexe" est sensible puisque la précision peut atteindre 80-85%.

Les images réalistes ont été téléchargées depuis vicos.si [1] puis traitées en local via *Processing.py*, tandis que les images simples viennent de kaggle.com [2].

3 Conception du modèle

Pour mener à bien ce projet, nous avons testé divers modèles mais la conclusion se révèle aussi simple que peu surprenante : le plus simple est le mieux. Le premier modèle que nous avons testé fut le *VGG16*.

```
class VGG16Network(tf.keras.Model):

    def __init__(self, _nbclasses, _imagesize):
        super().__init__()
        conv_base = VGG16(weights='imagenet',
                          include_top=False,
                          input_shape=(_imagesize, _imagesize, 3))
        conv_base.trainable = False
        self.vgg16 = conv_base
        self.flatten = Flatten()
        self.dense1 = Dense(256, activation='relu')
        self.dense2 = Dense(_nbclasses, activation='softmax')

    def call(self, _inputs):
        x = self.vgg16(_inputs)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)
```

FIGURE 4 – VGG16

L'idée était de partir d'un modèle pré-entraîné sur *imagenet*, de mettre *include_top* et *conv_base.trainable* à *false* pour pouvoir entraîner le modèle uniquement sur nos propres dernières couches denses, de faire

1. https://www.vicos.si/resources/fids30/
2. https://www.kaggle.com/datasets/sshikamaru/fruit-recognition

du *transfer-learning* donc. Les résultats, quand nous avions la patience de laisser le programme tourner, étaient bons mais sensiblement les mêmes que ceux que l'on aura avec notre *Basic convolutional network* pour un temps d'exécution qui dépassait allègrement les 15 minutes. Un tel temps d'exécution n'étant pas justifié par la justesse du modèle, ce dernier se révèle de fait inadéquat.

```
class InceptionV3Network(tf.keras.Model):

    def __init__(self, _nbclasses, _imagesize):
        super().__init__()
        base = InceptionV3(weights='imagenet',
                           include_top=False,
                           input_shape=(_imagesize, _imagesize, 3))
        base.trainable = False
        self.inceptionv3 = base
        self.globalAveragePooling2D = GlobalAveragePooling2D()
        self.dropout = Dropout(0.5)
        self.dense = Dense(_nbclasses, activation='softmax')

    def call(self, _inputs):
        x = self.inceptionv3(_inputs)
        x = self.globalAveragePooling2D(x)
        x = self.dropout(x)
        return self.dense(x)
```

FIGURE 5 – InceptionV3

La structure et le principe de ce nouveau modèle est semblable au précédent : il s'agissait de prendre un modèle déjà pré-entraîné, d'en enlever la fin, et de l'entraîner sur les dernières couches que l'on aura nous même ajouté. Le temps d'exécution est rapide, encore davantage qu'avec *BasicConvolutionalNetwork*, mais les résultats sont d'une précision désastreuses (sous la barre des 50%) et les pertes sont immenses (supérieures à 2). Il y a du sur-apprentissage. Le modèle s'est montré largement inadéquat. Nous aurions pu penser qu'un tel modèle, pensé pour la détection d'objet [3] était pertinent, mais il semble qu'*imagenet* soit trop éloignée de notre sujet.

```
class BasicConvolutionalNetwork(tf.keras.Model):

    def __init__(self, _nbclasses, _imagesize):
        super().__init__()
        self.conv1 = Conv2D(32, 3, activation="relu", input_shape=[_imagesize, _imagesize, 3])
        self.conv2 = Conv2D(64, 3, activation="relu")
        self.conv3 = Conv2D(94, 3, activation="relu")
        self.conv4 = Conv2D(128, 3, activation="relu")
        self.maxpool = MaxPooling2D(pool_size=(2, 2))
        self.flatten = Flatten()
        self.dense1 = Dense(512, activation='relu')
        self.dense2 = Dense(_nbclasses, activation='softmax')

    def call(self, inputs):
        x = self.conv1(inputs)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = self.maxpool(x)
        x = self.conv3(x)
        x = self.maxpool(x)
        x = self.conv4(x)
        x = self.maxpool(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)
```

FIGURE 6 – Basic convolutional network

3. https://en.wikipedia.org/wiki/Inceptionv3

La meilleure solution est souvent la plus simple, et c'est un retour parmi les premiers TP de Deep Learning qui nous mena à cette conclusion. Nous avons repris l'essentiel d'un petit réseau convolutif inspiré de VGG16, puis nous l'avons adapté à une utilisation *multiclasse*. Il consiste en une succession de couches denses suivi d'un *MaxPooling2D*, et se conclue en un *Flatten* et deux couches denses. La fonction d'activation est alors *softmax*, cohérente pour calculer la probabilité qu'une image test appartienne aux diverses classes de la base de données.

Illustration : figure 7. On remarque là que le modèle ne doute pas que l'image illustre un poivron rouge (99.87%), mais notez que la probabilité qu'elle représente une pomme rouge ou une fraise n'est pas nulle (0.02% et 0.10%), c'est sans doute du à la couleur.

FIGURE 7 – Prédiction sur une image test

4 Analyse des résultats

4.1 *simple-database* sur des images tests simples

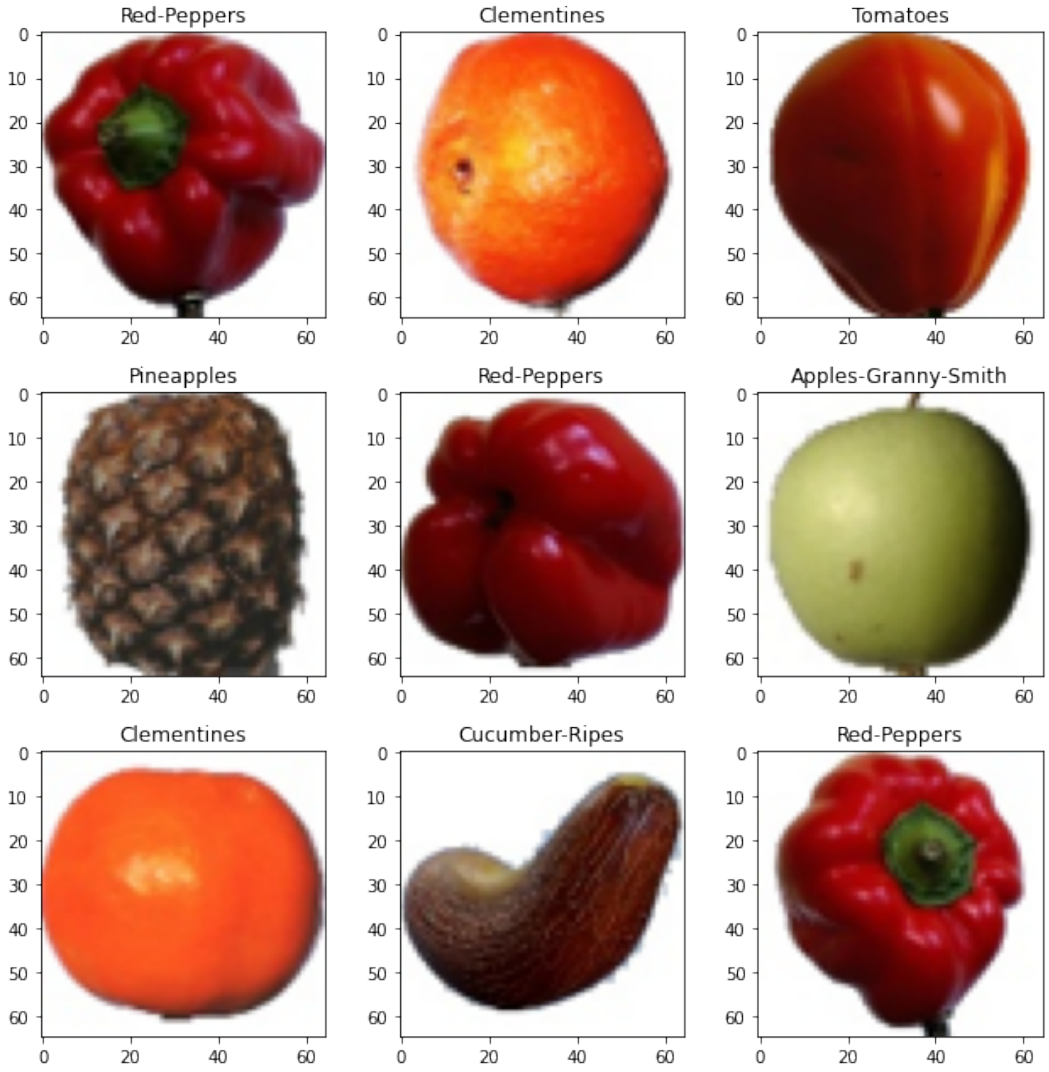

FIGURE 8 – Aperçu de la base de données

Il s'agit d'une base de données contenant des images simples : un unique fruit sur fond blanc.

```
labels1 = ['Apples-Braeburn', 'Apples-Granny-Smith', 'Apricots', 'Clementines','Corns',
           'Cucumber-Ripes', 'Green-Peppers', 'Kiwis', 'Lemons', 'Limes',
           'Mangos', 'Onions', 'Oranges', 'Peaches', 'Pineapples',
           'Red-Peppers', 'Strawberries', 'Tomatoes', 'Watermelons']
```

```
train_datagen = ImageDataGenerator(rotation_range=40,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True)
train_generator = train_datagen.flow(x_train1, y_train1)
val_generator = train_datagen.flow(x_val1, y_val1)

model1 = BasicConvolutionalNetwork(len(labels1), IMAGE_SIZE1)

model1.build(input_shape=(None, IMAGE_SIZE1, IMAGE_SIZE1, 3))
model1.summary()
model1.compile(loss='sparse_categorical_crossentropy',
               optimizer=optimizers.Adam(learning_rate=1e-4),
               metrics=['sparse_categorical_accuracy'])

history = model1.fit(train_generator,
                     validation_data=val_generator,
                     epochs=10)
```

```
2m 27.8s
```

FIGURE 9 – Test des images simples vis à vis de la base de données simple

Pour tous les entraînements, nous avons choisi de gonfler la base de données aux moyens de la classe *ImageDataGenerator*. Elle a contribué à améliorer légèrement les résultats. De plus, *sparse_ categorical_ crossentropy* et *sparse_ categorical_ accuracy* se sont montrés pertinents pour notre modèle. Enfin, un *learning_ rate* bas pour l'optimiseur a largement contribué à rendre les résultats précis et les pertes minimales.

Après un entraînement de 2m27s (15s par epochs), on évalue le modèle via les outils suivants :

— **Les courbes entraînement / validation** :

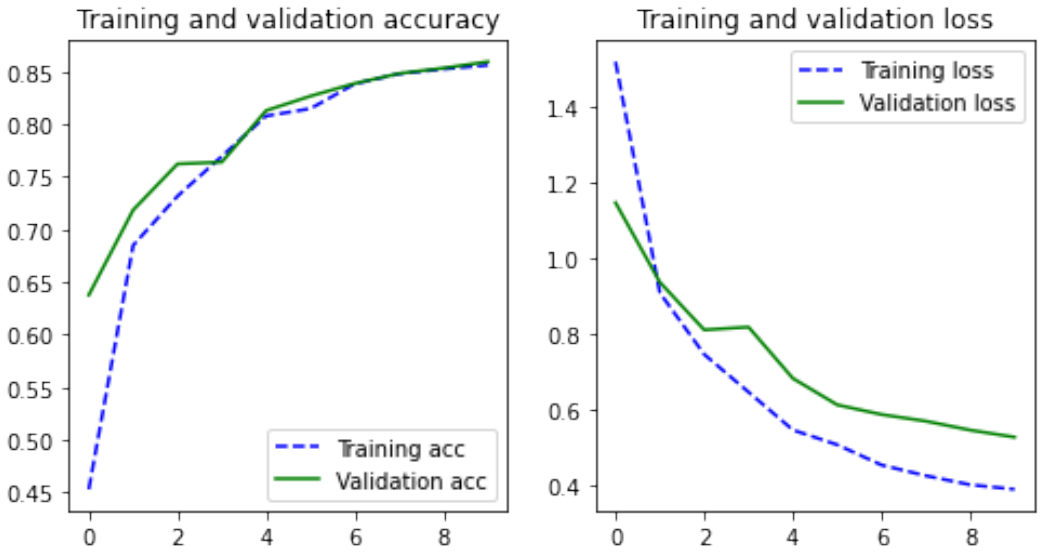

FIGURE 10 –

On note que dans les deux graphiques, les courbes se superposent assez bien. Cela veut dire que l'apprentissage de la base de données par le modèle est bon. C'est encourageant pour la suite des mesures.

— **La précision et le pertes du modèle** : LOSS : 0.10, ACCURACY : 97.04%
Ce sont des données excellentes. Preuve que le modèle est adéquat pour détecter la nature d'images simples. Un tel résultat n'est pourtant pas surprenant dans la mesure où les images sont très simplistes.

— **Matrice de confusion** :

```
(1) 'Apples-Braeburn', (2) 'Apples-Granny-Smith', (3) 'Apricots', (4) 'Clementines', (5) 'Corns',
(6) 'Cucumber-Ripes', (7) 'Green-Peppers', (8) 'Kiwis', (9) 'Lemons', (10) 'Limes',
(11) 'Mangos', (12) 'Onions', (13) 'Oranges', (14) 'Peaches', (15) 'Pineapples',
(16) 'Red-Peppers', (17) 'Strawberries', (18) 'Tomatoes', (19) 'Watermelons'

    [réel/estimé] (1) (2) (3) (4) (5) (6) (7) (8) (9)(10)(11)(12)(13)(14)(15)(16)(17)(18)(19)
    (1)        [[ 69   0   0   0   0   0   0   0   0   0   0   0   0   3   0   0   0   0   0]
    (2)         [  0  67   0   0   0   0   0   0   0   5   0   0   0   0   0   0   0   0   0]
    (3)         [  0   0  51   0   0   0   0   0   0   0   0   0  21   0   0   0   0   0   0]
    (4)         [  0   0   0  72   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
    (5)         [  0   0   0   0  64   0   0   0   2   0   0   0   0   0   0   0   0   0   0]
    (6)         [  0   0   0   0   0  53   0   0   0   0   0   0   0   0   0   0   3   0   0]
    (7)         [  0   0   0   0   0   0  64   0   0   0   0   0   0   0   0   0   0   0   0]
    (8)         [  0   0   0   0   0   0   0  68   0   0   0   0   0   0   0   0   0   0   0]
    (9)         [  0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0   0   0]
    (10)        [  0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0   0]
    (11)        [  0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0   0   0   0]
    (12)        [  0   0   0   0   0   0   0   5   0   0   0  59   0   0   0   0   0   0   0]
    (13)        [  0   0   0   0   0   0   0   0   0   0   0   0  70   0   0   0   0   0   0]
    (14)        [  0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0   0]
    (15)        [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0   0   0]
    (16)        [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  98   0   0   0]
    (17)        [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  72   0   0]
    (18)        [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 108   0]
    (19)        [  0   0   0   0   0   0   2   0   0   0   0   0   0   0   0   0   0   0  70]]
```

FIGURE 11 – Matrice de confusion

On remarque à cette occasion des valeurs importantes sur la diagonale, preuve que l'estimation est juste. On remarque toutefois des confusions entre les abricots et les oranges, sans doute en raison de leur couleur et de leur forme communes.

4.2 *simple-database* sur des images tests réalistes

```
EVALUATION DU MODELE basic_convolutional_network
MATRICE DE CONFUSION
[[ 1  0  0  0  0  0  0  0  0  0  0  0  0  0 17 11  7  0]
 [ 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  1  0  3  0  0  1 27  0  0  0 12  4  0  0  0  0  0]
 [ 5 10  0  0  0  4  0  4 11  5  0  0  0  0  0  3  0  6]
 [16  0  1  0  0  0  0  4  0  0  0 13 25  0  0  0  1  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [19  0  0  0  0  0  0 13  0  0  0  6 22  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  5  0  1  0  4  0  0 18  0 29  0  3]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [21  0  0  0  1  0  0  0  0  0  0  0  0  2 34 24  2  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
40/40 [==============================] - 0s 8ms/step - lo
LOSS     : 16.10
ACCURACY : 16.62%
```

FIGURE 12 – Performance de l'apprentissage sur une base de tests réalistes

La matrice de confusion porte bien son nom ! Les valeurs remarquables ne sont que rarement sur la diagonale, la précision est risible, les pertes sont immenses. Le résultat précédent, bien qu'encourageant, est loin d'être suffisant, car notre solution manque cruellement d'adaptabilité. Étudions donc l'efficacité d'une base de données purement réaliste, et voyons si nous pouvons mieux faire.

4.3 *realistic-database* sur des images réalistes

FIGURE 13 – Aperçu de la base de données

```
labels2 = ['apples', 'bananas', 'coconuts', 'grapes','lemons',
           'limes', 'mangos', 'oranges', 'pineapples', 'tomatoes']
EVALUATION DU MODELE basic_convolutional_network_1
MATRICE DE CONFUSION
[[20  0  0  1  0  0  0  0  3 24]
 [ 0 38  0  0 22  9  0  0  3  0]
 [ 6  0 21 16  0  0  3  0  2  0]
 [ 3  0  0 45  0 12  0  0  0  0]
 [ 0 28  0  0 15  0  0  5  0  0]
 [ 0  0  0  0  0 48  0  0  0  0]
 [ 0  0  0  0  4  0 35 19  0  2]
 [ 0  4  0  0  9  0  2 45  0  0]
 [ 0  0  2 20  0  0  0  0 38  0]
 [16  0  0  0  0  0  0  0  0 68]]
59/59 [==============================] - 0s 8ms/step - loss: 0.8
LOSS     : 0.87
ACCURACY : 63.44%
```

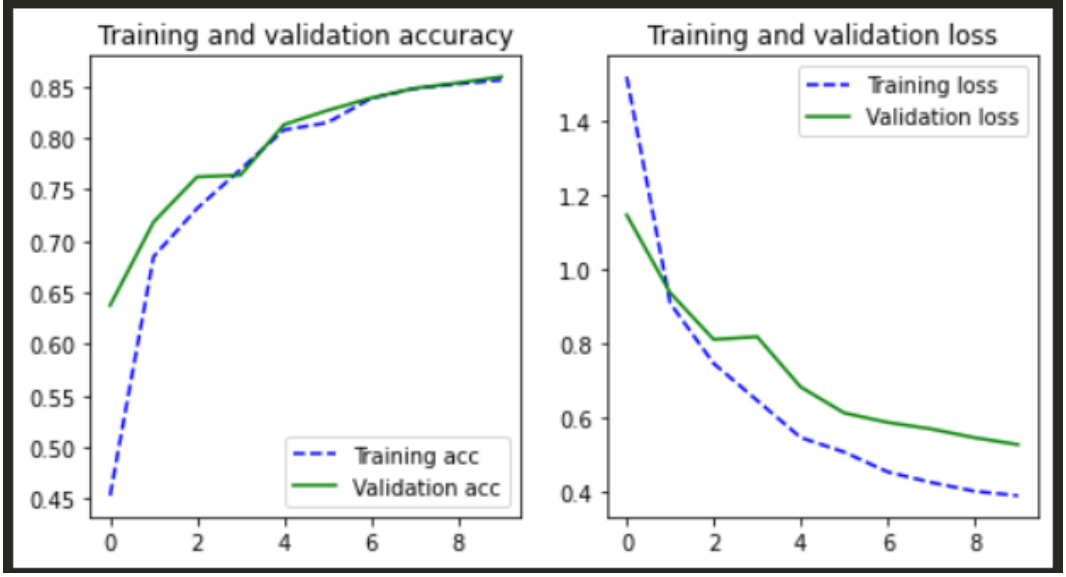

FIGURE 14 – Performance de l'apprentissage sur une base de données réaliste

Notre base de données compte désormais 10 classes, mais l'entraînement est strictement le même. La précision est de 63.44%, contre 10% si la prédiction était complètement aléatoire. C'est encourageant, car bien plus convaincant que les prédictions faites à la partie précédente. On note cependant des confusions entre pommes et tomates, entre bananes et citrons, entre noix de coco et raisins, entre raisins et citrons vert, entre mangues et oranges, et entre ananas et raisins. On remarque qu'il s'agit souvent de fruits qui ont la même couleur, mais le trop grand nombre d'informations contenu par les images ne doit pas aider à la détection non plus.

Ce résultat est toutefois le maximum que l'on puisse tirer de ce modèle avec cette base de données, en témoignent les courbes entraînement / validation qui se superposent assez bien. Il faut donc repenser une fois encore la base de données. Cette fois-ci, nous allons construire une base hybride entre la première et la seconde, elle comptera 10 classes également, et voyons ce que cela vaut.

4.4 *hybrid-database* sur des images réalistes

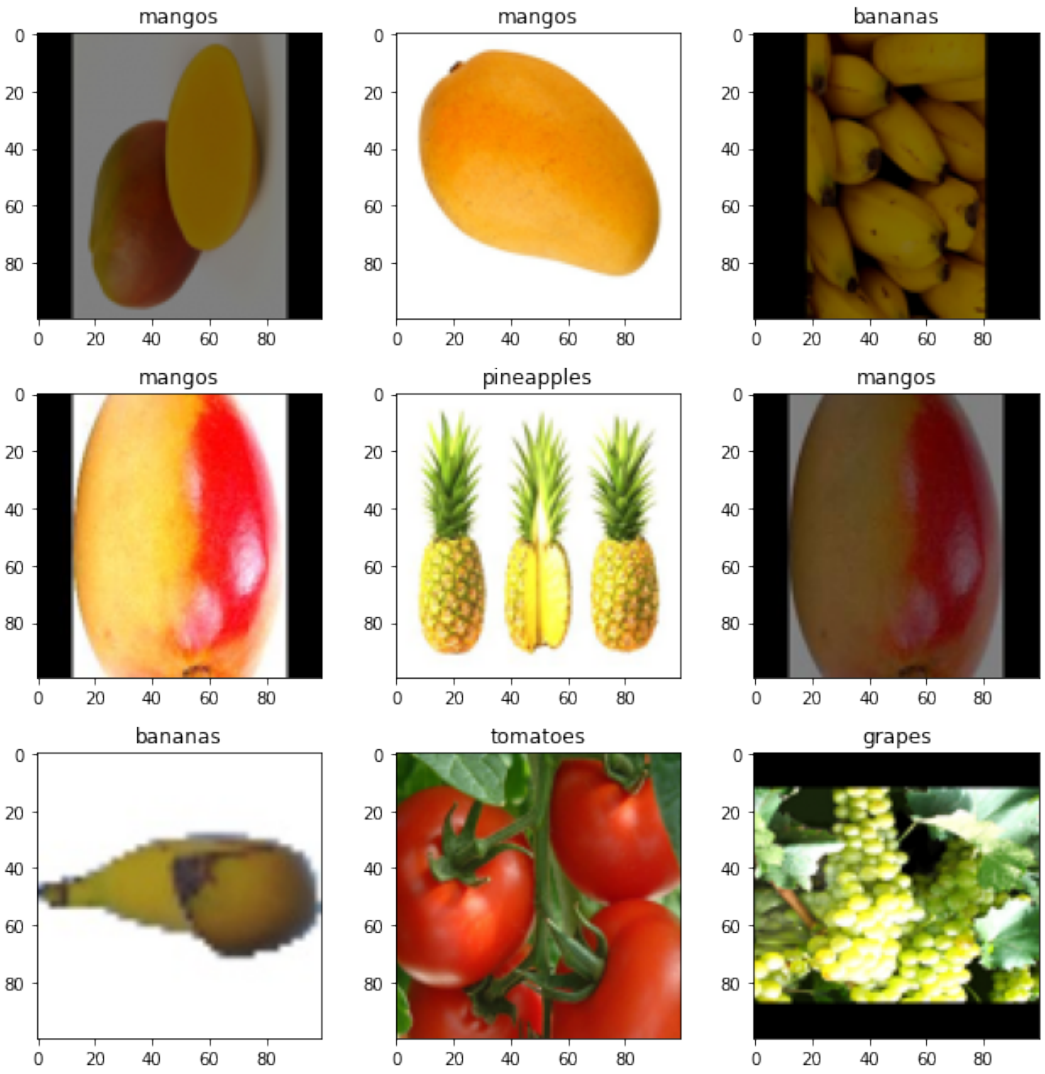

FIGURE 15 – Aperçu de la base de données

```
EVALUATION DU MODELE basic_convolutional_network_2
MATRICE DE CONFUSION
[[74  0  0  0  0 12  0  0  0  0]
 [ 0 60  0  0 27  0  0  0  0  0]
 [ 0  0 65  6  0  0  0  0 13  0]
 [ 0  7  0 60  0 17  0  0 12  0]
 [ 0  3  0  0 79  0  0  0  0  0]
 [ 0  0  0  0  0 73  0  0  9  0]
 [ 4  0  0  0  0  0 82  0  0  0]
 [ 0  0  0  0  8  0  0 90  0  0]
 [ 0  0 10  0  0  0  0  0 89  0]
 [12  0  0  0  0  0  0  0  0 86]]
90/90 [==============================] - 1s 14ms/step - loss: 0.
LOSS     : 0.52
ACCURACY : 84.41%
```

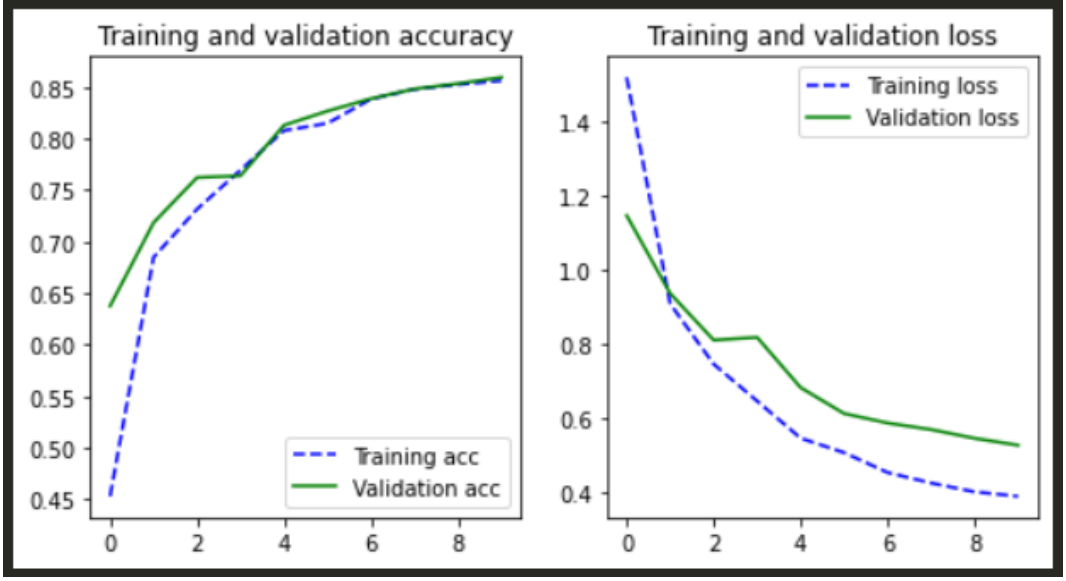

FIGURE 16 – Performance de l'apprentissage sur une base de données hybride

Les résultats sont immédiatement meilleurs ! La précision est de 84.41% et globalement la matrice de confusion possède ses poids forts sur sa diagonale. On note tout de même encore quelques erreurs, la principale étant, encore une fois, entre citrons et bananes. Ce sont les deux seuls fruits à être jaunes. On remarque d'ailleurs que la confusion concernant ces deux fruits est toujours aussi intense avec cette base de données (27) qu'avec la précédente (22). Cela veut dire qu'il aurait fallu ajouter davantage de photos permettant au modèle de discriminer ces deux fruits via leur forme. Ce résultat est toutefois très satisfaisant, car bien loin devant une prédiction aléatoire (10%) ainsi que de la prédiction précédente (63.44%). Enfin, les courbes montrent que l'apprentissage fut adéquat, ce résultat est donc le maximum que l'on puisse obtenir.

5 Bilan

BD	BD test simple / BD simple	BD test réaliste / BD simple	BD test réaliste / BD réaliste	BD test hybride / BD hybride
Précision	97.04%	16.62%	63.44%	84.41%
Pertes	0.10	16.10	0.87	0.52
Adaptabilité	Faible	Faible	Moyenne	Forte

FIGURE 17 – Bilan des performances du modèle *Basic convolutional network*