


Projet de traitement d'images : classification supervisée de Pokémons à l'aide du deep learning

Rapport court

Marc Villette
Juin 2024

Table des matières

1 Introduction

Le but du projet est de tester différentes techniques de deep learning pour la classification supervisée d'images dont les classes correspondent à des pokémons. Le langage utilisé est Python, et les principaux outils utilisés pour les fonctions de learning sont tensorflow et keras.

Le premier modèle testé est un CNN assez simple, tandis que le second utilise une méthode de transfer learning avec ResNet-101.

1.1 Présentation du Dataset

Le dataset est constitué d'images de pokémons (trouvées sur internet, avec Google images par exemple) au format JPG réparties en classes contenant chacune 100 images. Le nombre de classes est actuellement de 20. Les images du dataset servent à la fois au training et à la validation lors de l'entraînement des modèles, généralement réparties selon la proportion 80/20 respectivement. Le dataset de test est dissocié et contient 5 images par classe, soit 100 au total pour 20 classes.

Les données pour l'entraînement et la validation sont chargées depuis le dossier "Data" avec la fonction de keras `keras.preprocessing.image_dataset_from_directory`, donc les images sont automatiquement étiquetées en fonction des sous-dossiers dans lesquels elles sont placées (un par classe, et rangés par défaut dans l'ordre alphabétique).

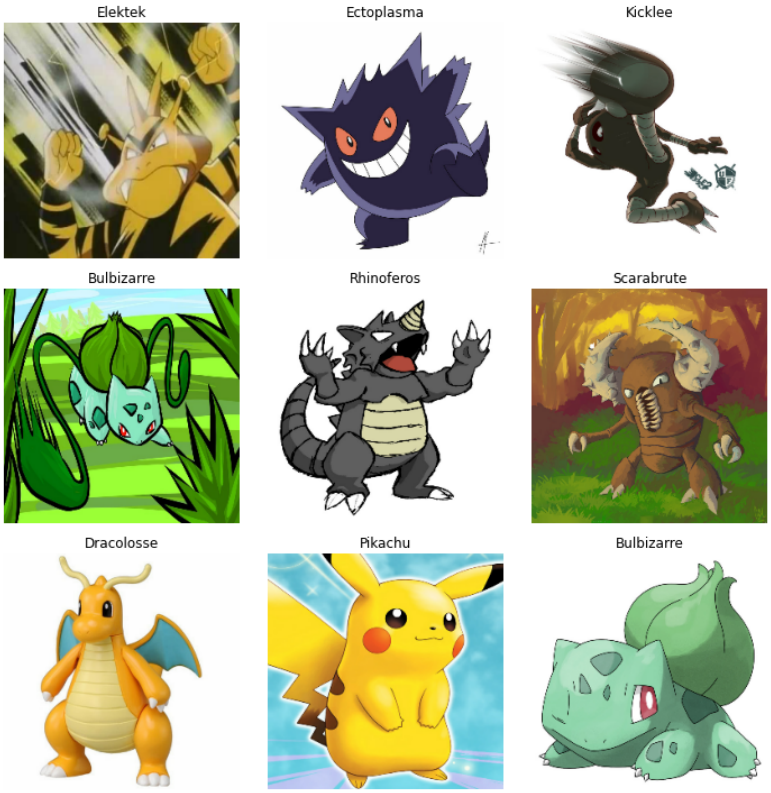

FIGURE 1.1 – *Exemple d'images du dataset*

2 Réseau de neurones à convolution

Le premier modèle testé est un réseau de neurones à convolution (CNN) assez simple codé de zéro.

2.1 Pré-traitement des données

Les images sont chargées et redimensionnées à la taille 256 x 256 pour, d'une part, avoir des images carrées, et d'autres part réduire la durée de l'apprentissage en travaillant avec des images de taille modérée.

La paramètre `validation_split` est fixé de sorte à ce que 80% des données soient utilisées pour la phase de training et 20% pour la phase de validation.

Enfin, en entrée du modèle les valeurs des pixels des images sont normalisées par 255 (car elles sont au format RGB 8 bits) pour que les calculs soient ensuite effectués sur des nombres appartenant à la plage [0,1] (ce qui accélère et améliore l'entraînement du modèle).

2.2 Architecture du modèle

Le modèle est principalement constitué d'un enchaînent de blocs qui contiennent chacun une couche de convolution, suivie d'une couche de normalisation par lots (Batch Normalization) et d'une couche de pooling.

```
layers.Conv2D(64, 3, activation='relu', padding='same'),
layers.BatchNormalization(),
layers.MaxPooling2D(),
```

FIGURE 2.1 – *Bloc convolutif avec 64 filtres*

Chaque couche de convolution applique N filtres de taille 3x3 à l'image d'entrée pour en extraire des caractéristiques locales. La fonction d'activation utilisée pour pour introduire de la non-linéarité est ReLU. Le nombre de filtres est divisé par 2 (et donc décroissant) pour chaque couche de convolution suivante. Le modèle inverse avec un nombre de filtres croissant a été testé mais n'a pas donné de meilleurs résultats (entraînement sur 5 classes). Le fait de réduire successivement le nombre de filtres peut agir comme une régularisation qui peut aider à généraliser le modèle en évitant qu'il ne s'adapte trop aux données d'entraînement, ce qui est utile quand lorsqu'on travaille avec assez peu de données comme ici. Par ailleurs différents tests ont été réalisés en ajoutant ou retirant des blocs convolutifs pour trouver un bon nombre de fitres. Par exemple ajouter une couche de convolution avec 256 filtres n'améliore pas les résultats.

Ensuite on applique une couche fully connected avec 64 ou 128 neuronnes (activation ReLU), précédée d'une régularisation par dropout qui désactive aléatoirement 50 % des neurones, ce qui permet de réduire le surapprentissage.

Enfin, après une seconde régularisation par dopout, le modèle se termine par une couche dense avec un nombre de neurones égal au nombre de classes et on utilise la fonction d'activation softmax pour convertir les sorties en probabilités.

```
layers.Flatten(),
layers.Dropout(0.5),
layers.Dense(128, activation='relu'),
layers.BatchNormalization(),

layers.Dropout(0.5),
layers.Dense(num_classes, activation='softmax')
```

FIGURE 2.2 – *Dernières couches du CNN*

L'optimiseur Adam est utilisé pour mettre à jour les poids du modèle, et "Sparse Categorical Crossentropy" est la fonction de perte utilisée pour l'entraînement car les étiquettes des classes sont des nombres entiers (et non des vecteurs binaires).

Une fois le modèle entraîné on peut le sauvegarder lui et les poids avec la fonction `model.save()`, et le recharger plus tard avec `keras.models.load_model()`.

2.3 Métriques d'entraînement

Pendant l'entraînement il est intéressant d'enregistrer des métriques telles que la précision (ou accuracy, qui est la proportion de prédictions correctes par rapport au nombre total de prédictions) et les pertes (ou loss, qui mesurent à quel point les prédictions du modèle sont éloignées des vraies valeurs), à la fois pour le set d'entraînement et de validation. Ces données sont stockées dans l'objet `History` retourné par la fonction `model.fit()`. On peut ensuite les extraire pour visualiser les courbes d'accuracy et de loss avec matplotlib.

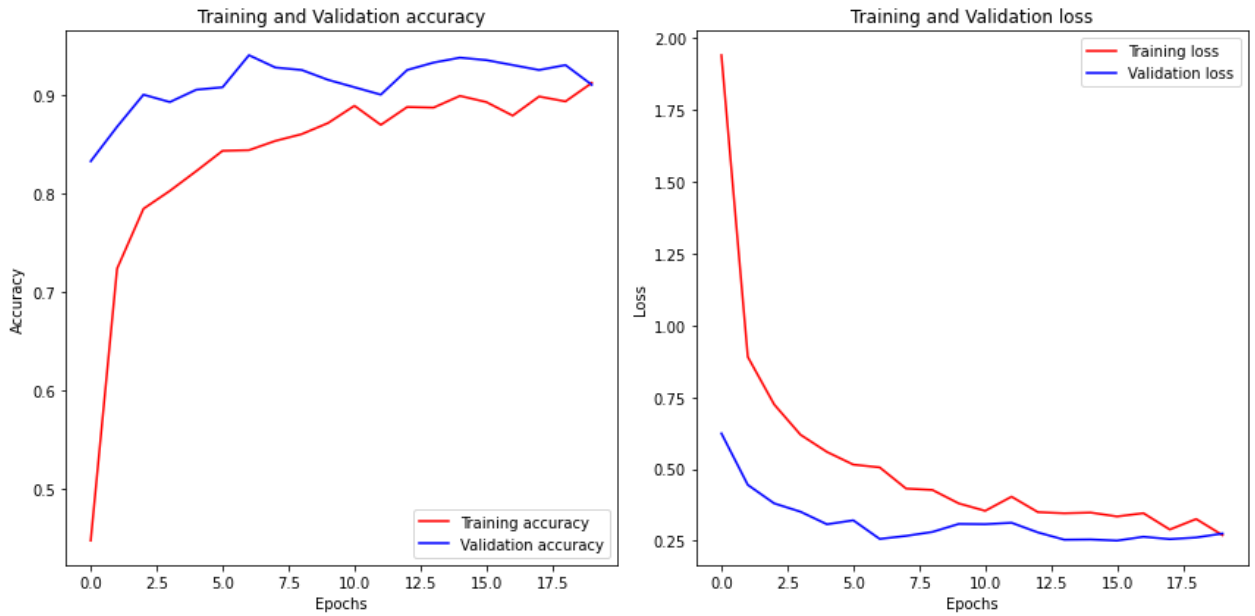

FIGURE 2.3 – *Graphiques avec les courbes de précisions (à gauche) et de pertes (à droite)*

Ces données sont importantes pour évaluer le modèle et trouver d'éventuels problèmes. Par exemple une précision d'entraînement forte mais accompagnée d'une précision de validation qui stagne et n'arrive pas aux mêmes valeurs peut indiquer un surentraînement.

2.4 Test

J'ai créé plusieurs fonctions permettant de faciliter la phase de test. Une première fonction, `preprocess_image`, permet de charger et pré-traiter une image dont le chemin d'accès est fourni. Une deuxième fonction nommée `predict_image` prend en entrée l'image pré-traitée pour retourner une prédiction de classe ainsi que la probabilité de cette prédiction (la fonction `model.predict()` de keras retourne un vecteur de 20 probabilités, une pour chaque classe, dont je prends le maximum pour avoir la classe la plus probable).

```
In [14]: predict_image(my_model5,img1,class_names,1)
1/1 [==============================] - 7s 7s/step
C'est un Pikachu ! (78.4%)
Out[14]: ('Pikachu', 0.7839813)
```

FIGURE 2.4 – *Exemple d'une prédiction (correcte)*

Une troisième fonction, `evaluate_model`, permet de de faire des prédictions pour l'ensemble d'un dossier et retourne le ratio de prédictions correctes sur le nombre total de prédictions/d'images du dossier.

3 Transfer Learning

Le transfert learning est une technique où un modèle pré-entraîné sur une tâche ou un set de données est réutilisé pour une autre tâche. Cette approche permet de transférer des connaissances déjà acquises par le modèle pré-entraîné pour travailler sur un autre problème.

3.1 ResNet-101

Les modèles que j'ai choisi d'utiliser ici sont ResNet-50 puis ResNet-101 (nommés ainsi car possédant respectivement 50 et 101 couches), des CNN particuliers car utilisant des connexions résiduelles. Je les ai choisis car ces modèles sont assez populaires dans le domaine de la classification de données. Ici j'ai voulu tiré parti du pré-entraînement de ResNet101 sur ImageNet, qui est une grande base de données contenant des millions d'images annotées appartenant à des milliers de classes, pour l'appliquer à la classification des pokémons. On charge donc ResNet101 avec les poids `"imagenet"`.

```
# Chargement du modèle pré-entraîné ResNet101 sans la couche de classification ImageNet
base_model = ResNet101(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
```

FIGURE 3.1 – *Chargement du modèle pré-entraîné ResNet101*

ResNet prend en entrée des images de dimensions 224 x 224 pixels, pour cette méthode c'est donc à cette taille que sont redimensionnées les images du dataset chargée
Ensuite j'ai ajouté de nouvelles couches de classification à la fin de l'architecture du modèle pour qu'il soit adapté à la tâche qui m'intéresse :

```
# Ajout de nouvelles couches de classification adaptées aux classes Pokémon
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.5)(x) # test
x = Dense(1024)(x)
x = BatchNormalization()(x)  # Ajout de la couche BatchNormalization
x = Activation('relu')(x)
x = Dropout(0.5)(x)
predictions = Dense(num_classes, activation='softmax')(x)

model = keras.models.Model(inputs=base_model.input, outputs=predictions)
```

FIGURE 3.2 – *Couches ajoutées à ResNet pour la classification de mes données*

3.2 Data augmentation

Pour limiter le surapprentissage j'ai ensuite décidé d'appliquer une augmentation de données (data augmentation) : le principe est d'augmenter artificiellement le nombre de données en générant des versions légèrement modifiées des images d'entraînement d'origine. Cette technique permet au modèle de mieux généraliser lors de l'apprentissage et donc de gagner robustesse.

J'ai sélectionné plusieurs transformations qui sont appliquées aléatoirement aux images d'origine pour les modifier dont une rotation dans une plage de 0 à 90 degrés, un retournement horizontal ou encore un zoom avant/arrière dans une plage de 80 à 120% par exemple. La fonction *ImageDataGenerator* de keras configure ces transformations et génère des lots d'images modifiées.

```
# Création d'un générateur d'images avec augmentation de données
train_data_generator = ImageDataGenerator(
    rotation_range=90,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    shear_range=0.2,
    horizontal_flip=True,
    validation_split=0.2
)
```

FIGURE 3.3 – *Générateur d'images pour l'augmentation de données*

Enfin, la fonction *flow_from_directory* permet de charger les images du dataset (pour l'entraînement et la validation) en appliquant les transformations. Le paramètre `shuffle=True` est utilisé pour mélanger les images du set d'entraînement de manière aléatoire à chaque epoch, ce qui aide à améliorer la généralisation du modèle.

FIGURE 3.4 – *Exemple d'images modifiées par la data augmentation*

3.3 Fine-tuning

Une autre approche (complémentaire) consiste à effectuer un fine-tuning de ResNet en l'entraînant partiellement avec mes données (les images de pokémons). Pour ce faire on peut dégeler quelques couches supérieures et leur permettre de s'adapter à l'ensemble de données, tout en maintenant les couches inférieures plus génériques de ResNet inchangées.

```
#Dégeler certaines couches pour le fine-tuning
for layer in base_model.layers[:10]:
    layer.trainable = True
```

FIGURE 3.5 – *Dégèle des 10 premières couches de ResNet*

En pratique, avec 20 classes, j'ai noté une légère amélioration de la précision lors de l'entraînement et des résultats lors de la phase de test, mais au détriment d'un temps d'apprentissage beaucoup plus long (car on entraîne plus de couches du réseau). Il faut donc essayer de trouver un bon compromis entre le nombre de couches à dégeler et les gains obtenus grâce à cette approche.